Fabio Kon
Anne-Marie Kermarrec (Eds.)

LNCS 7049

Middleware 2011

ACM/IFIP/USENIX 12th International Middleware Conference
Lisbon, Portugal, December 2011
Proceedings

Springer

Lecture Notes in Computer Science 7049

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Fabio Kon Anne-Marie Kermarrec (Eds.)

Middleware 2011

ACM/IFIP/USENIX
12th International Middleware Conference
Lisbon, Portugal, December 12-16, 2011
Proceedings

Springer

Volume Editors

Fabio Kon
University of São Paulo, Department of Computer Science
Rua do Matão, 1010, 05508-090, São Paulo, SP, Brazil
E-mail: fabio.kon@ime.usp.br

Anne-Marie Kermarrec
INRIA-Rennes
Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
E-mail: anne-marie.kermarrec@inria.fr

Cover photo: "Torre de Belém (night)", © 2007 Andrew Fecheyr (andruby),
made available under an Attribution 2.0 Generic (CC BY 2.0) license

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-25820-6 e-ISBN 978-3-642-25821-3
DOI 10.1007/978-3-642-25821-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: Applied for

CR Subject Classification (1998): C.2, H.4, D.2, H.3, H.5, K.6.5

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Welcome to the 12th edition of the ACM/IFIP/USENIX International Middleware Conference.

The importance of middleware software and systems keeps growing in a world where distribution and heterogeneity are the norm. Middleware abstractions are present everywhere, e.g., from data centers to networks of mobile devices, from multi-core architectures to social networks, bridging the gap between many areas including programming languages, distributed algorithms, networks, and databases.

Among the 125 initial submissions (from 28 countries) to Middleware 2011, 22 research papers and 2 industry papers were selected for inclusion in the technical program and the proceedings of the conference, resulting in an acceptance rate of 19%. All papers were reviewed by at least three reviewers and some of them had four or six reviews. After a discussion period and a rigid selection process, in which some good papers had to be cut out, we selected the 24 papers that appear now in the proceedings. Five of those papers passed through a shepherding process to make sure that minor problems detected by some reviewers were resolved before the papers were published. The papers were judged according to their originality, presentation quality, and relevance to the conference. The accepted papers cover a wide range of topics with a slight bias toward papers related to cloud computing and reliability, which are obviously hot topics at the moment. Many submissions were related to emerging cloud computing, data centers and server farms, as well as scalability and performance for system issues. We thank the Industry Track Chairs Dilma da Silva and Jan de Meer for their help with the selection of industry papers.

We are grateful to Rachid Guerraoui from EPFL for being the keynote speaker of Middleware 2011. He related one of the most successful middleware stories, namely, software transactional memory (STM) systems. An invited paper, included in the proceedings, co-authored by Vincent Gramoli and Rachid Guerraoui, conveys the message that STM systems should not only simplify a programmer's life but also be flexible enough to enable skilled programmers to take the most from this abstraction to ensure STM a promising future.

We were also delighted to grant the 10-Year Best Paper Award to Peter Druschel and Antony Rowstron for their Middleware 2001 paper "Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems." The 10-Year Best Paper Award seeks to reward those papers from the Middleware conference that was held exactly 10 years ago based on their impact on the academy and industry in the past 10 years. This tradition started with the 10th Middleware in 2009 in Urbana-Champaign, USA, and had its second edition in 2010 in Bangalore, India; Middleware 2011 in Lisbon, Portugal, hosted the third edition. This paper, published in 2001, is cited more than 4,000 times

according to Google Scholar and describes probably the most well-known structured peer-to-peer overlay platform. This represents one of the most influential papers in the past decade in distributed systems and we are honored that the Middleware conference was the one to give the authors this award.

We would like to express our deepest thanks to the authors of submitted papers, to the Program Committee members for their work in reviewing the papers and their enthusiasm in the discussions, to Paulo Ferreira and Luís Veiga the General Chairs and their team and, finally, to the members of the Steering Committee for their efforts toward making Middleware one of the major venues in distributed systems.

We hope you enjoy the papers in this volume of *Lecture Notes in Computer Science*.

December 2011

Anne-Marie Kermarrec
Fabio Kon

Organization

Middleware 2011 was organized under the joint sponsorship of the Association for Computing Machinery (ACM), the International Federation for Information Processing (IFIP), and USENIX.

Organizing Committee

General Chairs

Paulo Ferreira	INESC-ID Lisboa / IST, Portugal
Luís Veiga	INESC-ID Lisboa / IST, Portugal

Program Committee Chairs

Anne-Marie Kermarrec	INRIA, France
Fabio Kon	IME, University of São Paulo, Brazil

Industrial Track Chairs

Dilma da Silva	IBM T.J. Watson Research Center, USA
Jan de Meer	SmartSpaceLab, Germany

Workshops Chair

Peter Pietzuch	Imperial College, UK

Doctoral Symposium Chair

David Eyers	University of Otago, New Zealand

Posters and Demos Chair

Sonia Ben Mokhtar	CNRS Lyon, France

Local Arrangements Chair

João Barreto	INESC-ID Lisboa / IST, Portugal

Publicity Chair

Carlos Ribeiro	INESC-ID Lisboa / IST, Portugal

Financial Chair

João Nuno Silva	INESC-ID Lisboa / IST, Portugal

Registration Chair

João Leitão	INESC-ID Lisboa / IST, Portugal

Conference Web Chairs

Nuno Carvalho	INESC-ID Lisboa / IST, Portugal
Edgar Marques	INESC-ID Lisboa / IST, Portugal

Proceedings Chairs

João Garcia	INESC-ID Lisboa / IST, Portugal
Luís Veiga	INESC-ID Lisboa / IST, Portugal

Steering Committee

Gordon Blair	Lancaster University, UK (Chair)
Jean Bacon	University of Cambridge, UK
Guruduth Banavar	IBM, USA
Roy Campbell	University of Illinois at Urbana-Champaign, USA
Brian Cooper	Google, USA
Jan De Meer	SmartSpaceLab, Germany
Fred Douglis	IBM Research, USA
Indranil Gupta	University of Illinois at Urbana-Champaign, USA
Valérie Issarny	INRIA, France
Hans-Arno Jacobsen	University of Toronto, Canada
Wouter Joosen	KUL-DistriNet, Belgium
Cecilia Mascolo	University of Cambridge, UK
Elie Najm	ENST, France

Program Committee

Jean Bacon	University of Cambridge, UK
Ken Birman	Cornell University, USA
Gordon Blair	University of Lancaster, UK
Christian Cachin	IBM Research - Zurich, Switzerland
Roy Campbell	University of Illinois at Urbana-Champaign, USA
Renato Cerqueira	PUC-Rio, Brazil
Ranveer Chandra	Microsoft Research Redmond, USA
Lucy Cherkasova	HP Labs, USA
Brian F. Cooper	Google, USA
Paolo Costa	Imperial College, UK
Fábio Costa	UFG, Brazil
Geoff Coulson	University of Lancaster, UK
Koustuv Dasgupta	Xerox, India
Fred Douglis	EMC, USA
Frank Eliassen	University of Oslo, Norway
Markus Endler	PUC-Rio, Brazil
Patrick Eugster	Purdue University, USA
David Eyers	University of Otago, New Zealand
Paulo Ferreira	INESC-ID Lisboa / IST, Portugal
Davide Frey	INRIA, France
Vincent Gramoli	EPFL, Switzerland
Gang Huang	Peking University, China
Valérie Issarny	INRIA, France

Hans-Arno Jacobsen	University of Toronto, Canada
Wouter Joosen	KUL-DistriNet, Belgium
Flavio Junqueira	Yahoo Research, Barcelona
Anne-Marie Kermarrec	INRIA, France
Steve Ko	SUNY at Buffalo, USA
Fabio Kon	IME, University of São Paulo, Brazil
Cecilia Mascolo	University of Cambridge, UK
Jan De Meer	SmartSpaceLab, Germany
Erwan Le Merrer	Technicolor Research, France
Dejan Milojicic	HP Labs, USA
Klara Nahrstedt	University of Illinois at Urbana-Champaign, USA
Gian Pietro Picco	University of Trento, Italy
Peter Pietzuch	Imperial College, UK
Oriana Riva	Microsoft Research, USA
Étienne Rivière	University of Neuchâtel, Switzerland
Antony Rowstron	Microsoft Research Cambridge, UK
Rick Schantz	BBN Technologies, USA
Douglas Schmidt	SEI-CMU, USA
Dilma da Silva	IBM T.J. Watson Research Center, USA
Maarten van Steen	VU University Amsterdam, The Netherlands
Francois Taiani	Lancaster University, UK
Peter Triantafillou	University of Patras, Greece
Nalini Venkatasubramanian	University of California at Irvine, USA
Roman Vitenberg	University of Olso, Norway
Marko Vukolic	Eurecom, France
Ben Zhao	University of California Santa Barbara, USA

Referees

Gustavo L.B. Baptista	Nikos Ntarmos	Gilles Straub
Christos Efstratiou	Partha Pal	Guido Urdaneta
Nebil Ben Mabrouk	Animesh Pathak	José Valerio
Francesco Gadaleta	Aaron Paulos	Luís Veiga
Matthew Gillen	Hangwei Qian	Abhishek Verma
Stefan Guna	Kiran K. Rachuri	José Viterbo Filho
Leila Jalali	Olaf Resch	Jack Xing
Kyungbaek Kim	Kurt Rohloff	Maysam Yabandeh
Bert Lagaisse	Mo Sadoghi	Chunyang Ye
Ilias Leontiadis	Gerhard Schimpf	Xiujuan Yi
Walter Mascarenhas	Vinay Setty	Young Yoon
Giuliano Mega	Reza Sherafat	Kaiwen Zhang
Hein Meling	Francisco Silva e Silva	
Kianoosh Mokhtarian	Alessandro Sorniotti	

Sponsoring Institutions

International Federation for Information Processing
http://www.ifip.org

Association for Computing Machinery
http://www.acm.org

Advanced Computing Systems Association
http://www.usenix.org

Instituto Superior Técnico
http://www.ist.utl.pt

INESC-ID Lisboa
http://www.inesc-id.pt

Corporate Sponsors

BBN Technologies
http://www.bbn.com

Hewlett-Packard Company
www.hp.com

Innovation Makers
http://www.innovmakers.com

Table of Contents

Invited Paper

Social Networks

Storage and Performance Management

Green Computing and Resource Management

Notification and Streaming

Replication and Caching

Security and Interoperability

Run-Time (Re)configuration and Inspection

Industry

Democratizing Transactional Programming

Vincent Gramoli and Rachid Guerraoui

EPFL
Switzerland

Abstract. The transaction abstraction is arguably one of the most appealing middleware paradigms. It lies typically between the programmer of a concurrent or distributed application on the one hand, and the operating system with the underlying network on the other hand. It encapsulates the complex internals of failure recovery and concurrency control, significantly simplifying thereby the life of a non-expert programmer.

Yet, some programmers are indeed experts and, for those, the transaction abstraction turns out to be inherently restrictive in its classic form. We argue for a genuine democratization of the paradigm, with different transactional semantics to be used by different programmers and composed within the same application.

1 A Brief History of Transaction

The transaction abstraction is in essence a middleware paradigm: it allows multiple processes running on one or more processors (machines) to interact. The transaction abstraction lies typically between the programmer of concurrent and distributed applications and the operating system. It encapsulates complex concurrency control and failure recovery mechanisms behind a simple user interface.

The transaction abstraction is very old. It dates back to the 70's when it was proposed as a means to ensure the *consistency* of shared data [1], determined with respect to a sequential behavior. To formalize this notion of consistency, the *serializability* definition recast the consistency of an execution of transactions in terms of its equivalence to a sequential execution of transactions [2]: concurrent accesses have to behave as if they were executing sequentially—in other words, they must be *atomic*. Since that definition, researchers have derived other variants, like opacity [3], applicable to different transactional contexts.

Formerly used in databases, the transaction abstraction was adapted for the first time as a language construct in the form of *guards* and *actions* [4] in particular to address issues like *robustness* to hardware failures. The programmability of transactions has subsequently been studied in distributed systems in various forms, e.g., Argus [5], Eden [6] and ACS [7]. During that period, the first hardware support for such a transactional construct was invented to introduce parallelism in functional languages by providing synchronization on multiple memory words [8].

Later, transactional memory was proposed for concurrent programming especially to remedy the existing difficulties of programming with locks, e.g., priority

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 1–19, 2011.

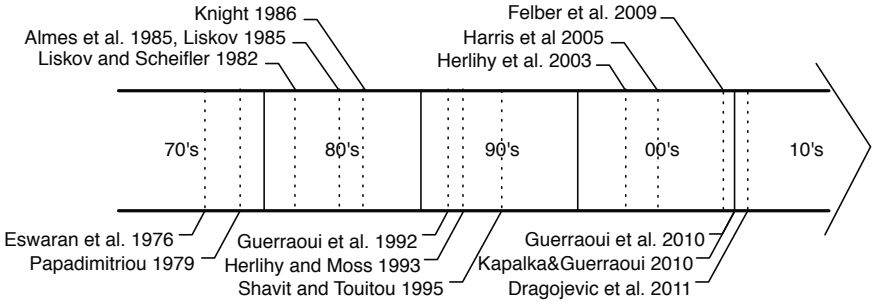


Fig. 1. A brief history of transactions

inversion, lock-convoying and deadlocks [9]. Since the advent of multicore architectures, the very notion of transaction memory has became an active topic of research[1]. Hardware implementations of such transactional systems [9] were generally limited by specific constraints and the programmer could only abstract away from these limitations using unbounded hardware transactions, a complex solution that most industrials are no longer exploring. Instead, a more hybrid tendency was adopted by implementing a best-effort hardware component that needs to be complemented by *software transactions* [10,11,12,13].

Software transactions were originally designed as a portable solution to execute a set of shared memory accesses fixed prior to execution [14]. Later software transactions were applied to a dynamic variant of this model, in which the control flow of the transaction was not predetermined [15]. Besides improvements stemming from the usage of timely information [16,17], new software transactions were derived to access more complex objects [18,19]. Nowadays, software transactions are even used in concurrent programs for the sake of coordinated failure recovery [20]. Despite these promising results, early investigations on the performance of software transactions have suggested their confinement to a research toy by questioning their ability to leverage multicore architectures [21].

Software transactions have finally won their spurs by outperforming sequential applications with only few cores. The result of [22] shows that an STM with manually instrumented benchmarks and explicit privatization outperforms sequential code by up to 29 times on SPARC with 64 concurrent threads and by up to 9 times on x86 with 16 concurrent threads. Even though the software overheads, induced by compiler instrumentation and transparent privatization, do not prevent transactions from outperforming sequential code, performance remains one of the major issue of transactions. Basically, an expert will never be able to extract as much concurrency from classic transactions than from synchronization primitives under the hood.

[1] A bibliography of the topic can be found at `http://www.cs.wisc.edu/trans-memory/biblio/list.html`.

Not surprisingly, researchers have kept exploring possible relaxations of the classic model since the early stage of transactions. Nesting models exploited *commutativity* of high level operations to favor concurrency [23,24]. In short, commutativity applies to operations whose order does not impact the transaction outcome. Such techniques would typically require the programmer to identify operations that can commute statically or to introduce code breakpoints. Others were dedicated to improve performance of typical contention hotspots: relaxed transactions were proposed for aggregate fields of database systems [25,26] on the one hand, and for search structures of multicore programs [15,27] on the other hand. A large majority of these relaxations rely however on complex code refactoring to improve performance and only a few, like [27], preserve both the sequential code and composition, most of them remaining non-exploitable by novice programmers.

To summarize, the transaction is an old appealing abstraction that has been the main topic of many practical and theoretical achievements in research, however, it has never been widely adopted in practice. Despite their genericness, transactions failed to be unified across distinct usages. Instead, transactions have always been tuned differently for different purposes, enforcing their incompatibility. An example is the recent adoption of transactions by IBM in their BlueGene/Q processor. This choice has been made in order to obtain the fastest supercomputer ever, yet only a very limited set of applications, which are supercomputing applications, will benefit from these highly tuned transactions. The incompatibility between distinct transactions breaks the appeal of the abstraction itself and prevents it from being used by the masses.

2 The Inherent Appeal of Transactions

The transaction paradigm is appealing for its simplicity as it preserves sequential code by hiding synchronization internals and its ability to promote concurrent code composition.

Algorithm 1. An implementation of a linked list operation with transactions

```
1:  tx-contains(val)_p:
2:    int result;
3:    node *prev, *next;
4:    transaction {
5:      curr = set → head;
6:      next = curr → next;
7:      while next → val < val do
8:        curr = next;
9:        next = curr → next;
10:     result = (next → val == val);
11:   }
12:   return result;
```

2.1 Preserving Sequentiality

Transactions preserve the sequential code in that their usage does not alter the sequential code, besides segmenting it into several transactions. More precisely, the regions of sequential code that must remain atomic in a concurrent context are simply delimited, typically by a transaction{...} block or similar tx-begin/tx-commit delimiters, as depicted in Algorithm 1—the existing data organization appears unchanged (Algorithm 2 (left)).

Programming with transactions shifts the inherent complexity of concurrent programming to the implementation of the transaction abstraction which must be done once for all. Thanks to transactions, writing a concurrent application follows a divide-and-conquer strategy where experts have the complex task of writing a live and safe transaction system with an unsophisticated interface so that the novice has simply to write a transaction-based application, namely delimit regions of sequential code.

Algorithm 2. The linked list node

```
1: Transactional structure node:
2:     intptr_t val;
3:     struct node * next;
4:     // Metadata management is implicit
5: Lock-based structure node_lk:
6:     intptr_t val;
7:     struct node_lk * next;
8:     volatile pthread_spinlock_t lock;
```

On the one hand, traditional synchronization techniques require generally the programmer to first re-factorize the sequential code. Using lock-free techniques, the programmer would typically need to use subtle mechanisms, like logical deletion, to prevent inconsistent memory deallocations, yet the memory management would not even be guaranteed to be simple, and may require additional re-engineering [28]. Using lock-based techniques, the programmer must explicitly declare and initialize all locks before protecting memory accesses as depicted in Algorithm 2; the programmer may even need to use a logical deletion technique as well as an additional validation phase to guarantee consistency [29].

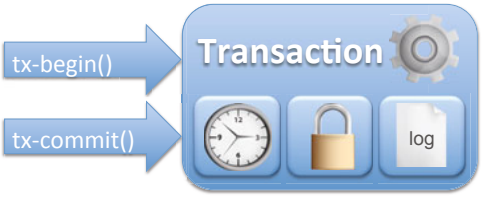


Fig. 2. The transaction abstraction hides complex synchronization mechanisms behind a simple interface

On the other hand, the transaction abstraction hides both synchronization internals and metadata management. If locks are internally used, they are declared and initialized transparently by the transaction system. Moreover, as the transaction system wraps memory accesses, a simple reference counting can keep track of the status of transactions accessing a particular location, before freeing the memory.

Despite its apparent simplicity and as depicted in Figure 2, the transaction system internally hides complex synchronization mechanisms. For example a single transactional system can exploit (i) time, to associate timestamps to values and guarantee that all values read belong to the same snapshot the transaction is acting upon; (ii) locks, for concurrent transactions to detect conflicts when accessing common data; and (iii) logs, to record operations that will be re-executed at commit time, or rolled back at abort time.

2.2 Enabling Composition

Transactions are also appealing for they allow concurrent programs to be reused in a modular fashion. More specifically, transactions allow Bob to compose existing transactional operations developed by Alice into a composite one that preserves the safety and liveness of its components [30] as depicted in Figure 3.

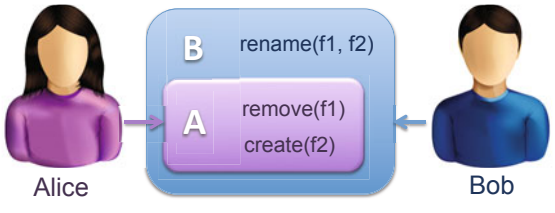


Fig. 3. Bob composes Alice's component operations remove and create into a new operation rename that preserves the safety and liveness of its components

By contrast, alternative synchronization techniques do not facilitate composition. For example, consider a simple directory abstraction mapping a name to a file. With transactions, one can compose the removal of a name and the creation of a new name into a rename action. If a user renames a file from one directory d_1 to another d_2 while another rename a file from d_2 to d_1, directories must be protected with care to avoid deadlocks. In the lock-based file system hierarchy of the Google File System [31], each directory at the same path depth has to be locked in a pre-determined ordering to prevent deadlock in such a scenario. In other words, Bob must first understand the locking strategy of Alice to ensure the liveness of his own operations. For the same reason, the header of the Linux kernel file `mm/filemap.c` comprises 50 lines of comments explaining the locking strategy.

Existing lock-free techniques are even more complex as they require a multi-word compare-and-swap to make the two renaming actions atomic while retaining concurrency [32].

By contrast, a transaction system detects a conflict between the two renaming transactions and let only one of the two commit, the other one is restarted or resumed later. Deciding upon the conflict resolution strategy is the task of a dedicated service, called a contention manager and various strategies have been proposed [33].

3 The Inherent Limitations of Transactions

A transaction delimits a region of accesses to shared locations and protects the set of locations that is accessed in this region. By contrast, a (fine-grained) lock generally protects a single location even though it is held during a series of accesses as depicted in Algorithm 3. This makes a crucial difference between transactions and locks in terms of expressiveness, concurrency and performance.

Algorithm 3. An implementation of a linked list operation with locks

```
1: lk-contains(val)_p:
2:   int result;
3:   node_lk *prev, *next;
4:   lock(&set → head → lock);
5:   curr = set → head;
6:   lock(&curr → next → lock);
7:   next = curr → next;
8:   while next → val < val do
9:     unlock(&curr → lock);
10:    curr = next;
11:    lock(&next → next → lock);
12:    next = curr → next;
13:  unlock(&curr → lock);
14:  unlock(&next → lock);
15:  result = (next → val == val);
16:  return result;
```

3.1 Lacking Expressiveness

To make our point that transactions are inherently limited in terms of expressiveness we define *atomicity* as a binary relation over shared memory accesses π and π' of a single transaction within an execution α: $atomicity(\pi, \pi')$ is true if π and π' appear in α as if they were both occurring at one common indivisible point of the execution. It is important to notice that this relation is not transitive, i.e., $atomicity(\pi_1, \pi_2) \wedge atomicity(\pi_2, \pi_3) \not\Rightarrow atomicity(\pi_1, \pi_3)$. In fact, as π_2 may appear to have executed at several consecutive points of the execution,

the points at which π_1 and π_2 appear to have occurred may be disjoint from the points at which π_2 and π_3 appear to have occurred.

A process locking x during the point interval $(p_1; p_2)$ of α, in which it accesses x, is guaranteed that any of its other accesses during this interval will appear atomic with its access to x. For example, the process guarantees $atomicity(r(x), r(y))$ and $atomicity(r(y), r(z))$ but not $atomicity(r(x), r(z))$ in the following lock-based program:

$$P_\ell = \mathsf{lock}(x)\ r(x)\ \mathsf{lock}(y)\ r(y)\ \mathsf{unlock}(x)\ \mathsf{lock}(z)\ r(z)\ \mathsf{unlock}(y)\ \mathsf{unlock}(z).$$

Conversely, a process executing the following transaction block ensures $atomicity(r(x), r(y))$, $atomicity(r(y), r(z))$ but also $atomicity(r(x), r(z))$, which is the transitive closure of the atomicity relations guaranteed by P_ℓ. Note that there is no way to ensure the two former atomicity relations with classic transactions without also ensuring the latter.

$$P_t = \mathsf{transaction}\{r(x)\ r(y)\ r(z)\}.$$

This lack of expressiveness when using transactions is directly implied by their syntax, which consists of an open/close block delimiting a compound statement [34]. In this sense, this expressiveness limitation is not related to the way transactions are used but to the transaction abstraction itself. This open/close block does neither accept a memory location nor a semantic hint as a parameter. Hence, it blindly guarantees that all its accesses appear as if there was an indivisible point in the execution where they *all* take effect.

Fig. 4. Among the correct linked list schedules, 20% of them are precluded when using transactions

3.2 Impact on Concurrency

The level of expressiveness is crucial especially when it restricts the set of acceptable schedules, and hence achievable concurrency, in a real workload. The low expressiveness of transactions translates actually into a concurrency loss on very common workloads. For example, consider the transactional linked list program

depicted in Algorithm 1. Clearly, the value of the $head \rightarrow next$ pointer observed by the transaction (Line 6) is no longer important when the transaction is checking whether the value val corresponds to a value of a node further in the list (Line 7), yet a concurrent modification of $head \rightarrow next$ can invalidate the transaction when reading $next \rightarrow val$; this is a false-conflict leading to unnecessary aborts. Such unnecessary aborts limit concurrency because they preclude schedules that would be correct, be all the transactions committed [35]. Conversely, the hand-over-hand locking program of Algorithm 3 allows such concurrent update (Line 7) when checking the value (Line 8), starting from the second iteration of the while-loop. Lock-free linked list algorithms [36,28] would not suffer such false-conflict either.

To quantify the impact of the limited expressiveness of transactions on the number of accepted schedules, consider that program P_t above executes concurrently with program $P_1 = \mathsf{transaction}\{w(x)\}$ and $P_2 = \mathsf{transaction}\{w(z)\}$. As there are four ways of placing the single access of one of these two programs between accesses of P_t and five ways of placing the remaining one in the resulting schedule, there are twenty possible schedules. Note that all are allowed in a linked list implementation; however, transactions that ensure opacity [3] (as it is the case for most classic ones) preclude four of these schedules: those in which P_t accesses x before P_1 ($P_t \prec P_1$), P_1 terminates before P_2 starts ($P_1 \prec P_2$) and in which P_2 accesses z before P_t ($P_2 \prec P_t$). The proportion of schedules precluded by transactions among all possible ones is depicted in Figure 4.

3.3 Impact on Performance

The metadata management overhead of transactions when starting, accessing shared memory and committing, is expected to be compensated by exploiting concurrency [22]. In scenarios like the previous linked list program where transactions fail to fully exploit all available concurrency, their performance cannot compete with lock-based or lock-free algorithms. Recall that this is due to the expressiveness limitation inherent to transactions—it is thus not tied to the way transactions are used but to the abstraction itself. The conjunction of overhead and limited concurrency of transactions prevents them from outperforming well-engineered lock-based and lock-free alternatives.

To illustrate the impact on performance, we compared the existing Java concurrency package to a classic transaction library written in Java, TL2 [16], on a 64-way Niagara 2 machine. We present the results obtained on a simple Collection benchmark of 2^{12} elements providing contains, add, remove and size operations with an update and a size ratios of 10% each. As the existing lock-free data structures do not support atomic size we had to use the copyOnWriteArraySet workaround of this package as recommended for circumventing this limitation [37]. We compared it against the linked list implementation building upon TL2. The throughput speedups over sequential of classic transaction and the existing collection are depicted in Figure 5. The existing collection performs 2.2× faster than classic transactions on 64 threads.

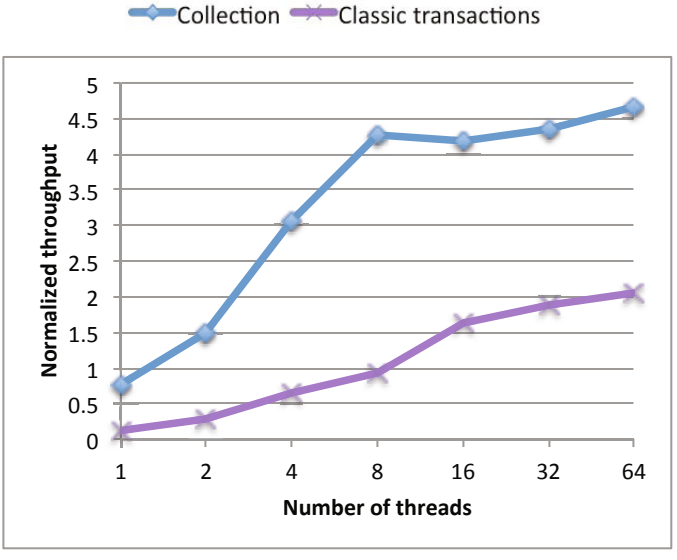


Fig. 5. Throughput (normalized over the sequential one) of classic transactions and the existing concurrent collection

4 Democratizing Transactions: The Challenge

Classic transactions share a single semantics for all types of applications. This simplifies the development of a transaction system by requiring the same guarantee for all its transactions, independently from their role in the concurrent applications. In some scenarios this semantics is, however, overly conservative and limits concurrency and performance (cf. Section 3). Without additional control, skilled programmers are frustrated by not being able to obtain highly efficient concurrent programs as depicted in Figure 6. In order to rather exploit adequately the concurrency allowed by the semantics of an application it is necessary to trade part of the simplicity of transactional memory for additional control.

We argue that for the transactional abstraction to really become a widely used programming paradigm it should be *democratized*. Not only is it important for transactions to be an off-the-shelf solution for novices, but also to give additional control to experts in concurrent programming.

Therefore, we believe that various transaction semantics should be able to run concurrently: a default semantics capturing the classic single-global-lock atomicity (i.e., opacity [3]), and more complex semantics capturing more subtle behaviors (e.g., elastic-opacity [27]). The challenge is twofold. First, the transaction abstraction should allow the expert programmers to easily express hints about the targeted application semantics without modifying the sequential code

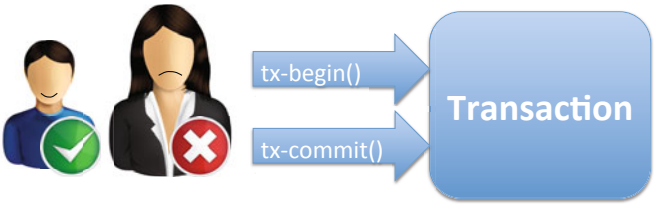


Fig. 6. The novice programmer benefits from the simplicity of transactions whereas the expert programmer is frustrated by its lack of flexibility

but simply delimiting its regions like for classic transactions. Second, the semantics of each transaction must be preserved even though multiple transactions of different semantics can access common data concurrently.

This second property is crucial and makes the development of a transactional system even more complex.

4.1 Expressiveness and Simplicity

Several relaxed transaction models have been proposed as an alternative to the classic transaction model. Such relaxed models can generally achieve a greater level of flexibility than the classic model by avoiding unnecessary aborts thus tolerating additional schedules.

An explicit early `release` can be used to ask statically the transaction to ignore false conflicts [15] and hence avoid unnecessary aborts. For example, to achieve the same expressiveness as the lock-based linked list of Algorithm 3 one could use early release to force the transaction to unprotect some of its read locations while executing. More precisely, a release call of location x could indicate from which point of the transaction all conflicts involving its read of x can start being ignored. Despite increasing expressiveness, the use of early release may hamper transaction composition. Alice may implement an atomic linked list $\mathsf{add}(x)$ using early release, yet Bob cannot reuse Alice's code to develop an atomic $\mathsf{addIfAbsent}(x, y)$ that inserts x only if y is absent. Typically, the resulting operation would not be atomic: two instances $\mathsf{addIfAbsent}(x, y)$ and $\mathsf{addIfAbsent}(y, x)$ may insert concurrently x and y, leading to an inconsistency.

A first relaxing methodology consists in open nesting [24] that is considered effective to increase concurrency. The key underlying idea is that nested transactions typically commit before the outer transaction ends but pass a high level abstraction of their changes to the outer transaction. As a result, the changes committed by a nested transaction become immediately visible from concurrent transactions and abstract locks indicate which pairs of nested operations conflict. Open nesting may lead to deadlocks if the accesses to shared locations are not ordered with care [38]. Specifically, this problem is similar to the one raised with explicit locks as open nesting let the programmers acquire abstract locks even upon abort.

A second relaxing methodology is transactional boosting [39]. It benefits from commutativity by considering transactional operations at a high level of abstraction. If two high level operations commute, they can be executed in any order despite the conflicts between their low-level operations. To this end, high level operations are considered as a whole and the programmer must identify operations that commute and define inverse operations. Considering higher level operations diminishes the amount of information that needs to be logged and possibly rolled back. Each operation acquires an abstract lock similar to open nesting so that two operations conflict if and only if they do not commute. Upon abort, a transaction rolls back its changes by executing the appropriate inverse operations that compensate its logged operations. Typically such models require the programmer to identify commutative operations and to write an appropriate compensating block of action for each non-commutative operation, such a compensate block is typically as long as the corresponding transaction block itself.

Inherently more complex to use than the classic transaction model, these initial relaxed transaction models lost the appealing aspects of transactions: either by requiring significant code refactoring or by breaking composition. Therefore, it is crucial to guarantee sequential code preservation and transaction composition when deriving new relaxed models targeting high expressiveness.

4.2 Sequentiality and Composition

A relaxed transaction model preserving sequential code and guaranteeing composition was proposed as the elastic transaction model [27]. This model provides a semantics of transactions that enables to efficiently implement search structures. Just like for a classic transaction, the programmer must simply delimit the blocks of code that represent elastic transactions, thus preserving sequential code as depicted in Algorithm 4. Elastic transactions are fully compatible with classic transactions thus inheriting the ability to compose of the classic model. Bob directly encapsulates Alice's elastic transactions, into another transaction, choosing between labeling it as elastic or classic, hence guaranteeing atomicity and deadlock-freedom of its own operation. Typically, Bob can easily compose Alice's elastic $\mathsf{add}(x)$ into a classic $\mathsf{addIfAbsent}(x, y)$.

In contrast with classic transactions, during its execution an elastic transaction can be cut (by the elastic transactional system) into multiple classic transactions, depending on the conflicts it detects. For example, consider the following history of shared accesses in which transaction j adds 1 while transaction i is parsing the data structure to add 3 at its end.

$$\mathcal{H} = r(h)^i, r(n)^i, r(h)^j, r(n)^j, w(h)^j, r(t)^i, w(n)^i.$$

This history is clearly neither serializable [2] nor opaque [3] since there is no history in which transactions i and j execute sequentially and where $r(h)^i$ occurs before $w(h)^j$ and $r(n)^j$ occurs before $w(n)^i$ (yet the high level insert operations of this history are atomic). A traditional transactional scheme would detect two

Algorithm 4. Java pseudocode of the add() operation with elastic transactions

```
1: public boolean add(E e):
2:   transaction(elastic) {
3:     Node⟨E⟩ prev = null
4:     Node⟨E⟩ next = head
5:     E v
6:
7:     if next == null then // empty
8:       head = newNode⟨E⟩(e, next)
9:       return false
10:    while (v = next.getValue()).compareTo(e) < 0 do // non-empty
11:      prev = next
12:      next = next.getNext()
13:      if next == null then break
14:    if v.compareTo(e) == 0 then
15:      return false
16:    if prev == null then
17:      Node⟨E⟩ n = new Node⟨E⟩(e, next)
18:      head = n
19:    else prev.setNext(new Node⟨E⟩(e, next))
20:    return true
21:  }
```

contradicting conflicts between transactions i and j, and the transactions could not both commit. Nonetheless, history $\mathcal{H}$ does not violate the correctness of the integer set: 1 appears to be added before 3 in the linked list and both are present at the end of the execution.

The programmer has simply to label transaction i as being elastic to solve this issue. Then, history $\mathcal{H}$ can be viewed as the combination of several pieces:

$$f(\mathcal{H}) = \boxed{r(h)^i, r(n)^i}^{s_1}, r(h)^j, r(n)^j, w(h)^j, \boxed{r(t)^i, w(n)^i}^{s_2}.$$

In $f(\mathcal{H})$, elastic transaction i has been cut into two transactions s_1 and s_2. Crucial to the correctness of this cut no two modifications on n and t have occurred between $r(n)^{s_1}$ and $r(t)^{s_2}$. Otherwise the transaction would have to abort.

These cuts enable more concurrency than what the expert programmer could do with classic transactions. First, a cut can split dynamically an elastic transaction depending on the interleaving of its accesses with other transaction accesses, yet it would be incorrect to replace statically the elastic transaction by multiple classic transactions, as the interleaving is not predictable. Second, identifying commutativity of accesses cannot enable the concurrency of elastic transactions because, depending on the current interleaving of accesses, two accesses that are (statically) non-commutative can be considered *dynamically-commuting* in

elastic transactions. For example, elastic transactions enable additional concurrency between two linked list adds by allowing the history involving transactions t_1 and t_2: $r(h)^{t_1}, r(n)^{t_2}, w(h)^{t_2}, w(n)^{t_1}$ in which neither $r(n)^{t_2}$ and $w(n)^{t_1}$ nor $r(h)^{t_1}$ and $w(h)^{t_2}$ commute.

4.3 Impact on Performance

To illustrate the benefit of combining relaxed and classic transactions, the collection benchmark was run in the exact same settings as the one used to obtain Figure 5. Each of the three parse operations contains, add and remove is implemented with an elastic transaction and the size operation, which returns an atomic snapshot of the number of elements, is implemented with a classic transaction to ensure atomicity of all four operations. As an example, the Java pseudocode of the add operation based on an elastic transaction is depicted in Algorithm 4.

The performance of combining elastic transactions with classic transactions, is compared in Figure 7 against the performance obtained with the existing concurrent collection package and with the classic transactions alone. The best performance we obtained by combining elastic and classic transactions is higher than classic transactions alone by 3.5× and than the existing collection package by 1.6×. Unfortunately, the performance does not scale up to the maximum number of threads 64. We conjecture that the slow-down between 32 and 64

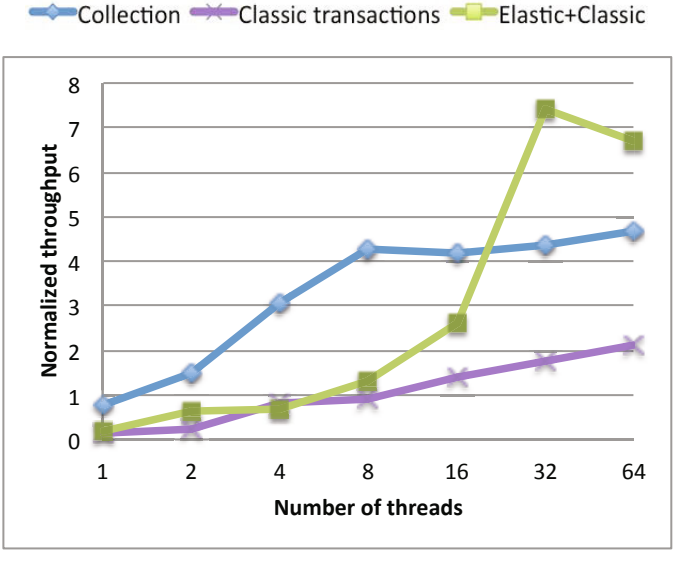


Fig. 7. Throughput (normalized over the sequential one) of elastic and classic transactions, the classic transactions alone and the existing concurrent collection

threads by repeatedly aborting the size operations, in the same vein as balance operations of the bank benchmark [40] or toxic transactions [41]. More precisely, the size executes within a classic transaction that has limited concurrency and which may thus produce an abort each time a concurrent update (add or remove) is modifying concurrently any location of the data structure.

5 Mixing Several Semantics

Mixing semantics means providing multiple transactions of different semantics to let the programmer choose the right semantics for each delimited region of the program. As these transactions can potentially access concurrently the same locations, it is crucial that one transaction does not alter the semantics of the others.

More precisely, we consider the semantic of classic transactions, opacity, to be the strongest one. Hence the novice can use exclusively the default semantics for all transactions, making sure that the resulting program is correct. Nevertheless, the expert can use a relaxed semantics that preserves sequential code (like elastic one) for some transactions and the classic one for others, to obtain higher expressiveness and better performance. The challenge is to preserve the semantics of all individual transactions. In the case of mixing elastic with classic transactions the resulting correct histories should thus be equivalent to a sequential legal history of elastic sub-transactions and classic transactions, as long as elastic sub-transactions result from consistent cuts (as required by the elastic transaction semantics). Consequently, mixing additional semantics may become rapidly challenging.

The key idea of mixing several semantics relies on providing various kinds of transactions among which the programmer can choose the adequate one that better matches its needs. More specifically, the programmer can start a transaction that executes the default intuitive semantics unless the tx-begin call is given some parameter, that serves as a hint to indicate the transaction semantics. With mixed semantics, not only does the transaction remain an off-the-shelf paradigm for novices, but it also gives control to the experts to boost the performance of some transactions (Figure 8).

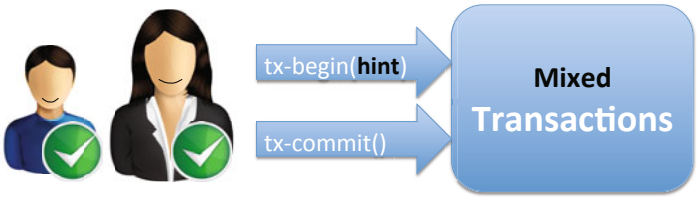


Fig. 8. Novice and expert programmers should both benefit from the simplicity and flexibility of a mixed transactional model

5.1 Combining Classic, Snapshot and Elastic Transactions

To go a step further in exploiting mixed transactions, here is an example of an additional transaction semantics, called *snapshot*, in addition to the two pre-existing ones, elastic and classic. This snapshot transaction semantics provides a way for the programmer to implement an atomic snapshot operation that can run concurrently with updating transactions (elastic or classic) modifying the data structure in a complex way (even at distinct locations). This is typically an appealing semantics to design an operation whose result depends on multiple elements of the data structure, like a Java Iterator. As an example, a size operation preserving sequential code and that is depicted in Algorithm 5 uses a snapshot transaction.

Algorithm 5. Java pseudocode of the size() operation with snapshot transactions

```
1:  public int size():
2:    transaction(snapshot) {
3:      int n = 0
4:      Node⟨E⟩ curr = head
5:
6:      while curr ≠ null do
7:        curr = curr.getNext()
8:        n++
9:      return n
10:   }
```

The key idea is for the snapshot to detect the locations that have been concurrently modified and to exploit multiversion concurrency control to bypass these conflicts. Using a global counter and version numbers associated with location values, the snapshot can detect at read time whether a location has been concurrently updated by comparing its current version to the value of the global counter at the time the snapshot started. If such a concurrent modification is detected, the snapshot has to select an old value (with a lower version number) of the overwritten location that is consistent with the start time of the snapshot transaction (i.e., higher than the value of the global counter at the time it started).

More precisely, multiple versions must be maintained at each location by every update transaction, be they elastic or classic—in our case two versions were maintained, this was actually sufficient to speed up the performance significantly. All update transactions create a backup value-version pair before overwriting them. The snapshot transaction has simply to detect whether the location it aims at accessing has a higher version than its upper bound *ub* to try getting an older version that could let it commit. Naturally, the snapshot transaction may have to abort if the older version is still too recent as no transactions keep track of more than two versions here.

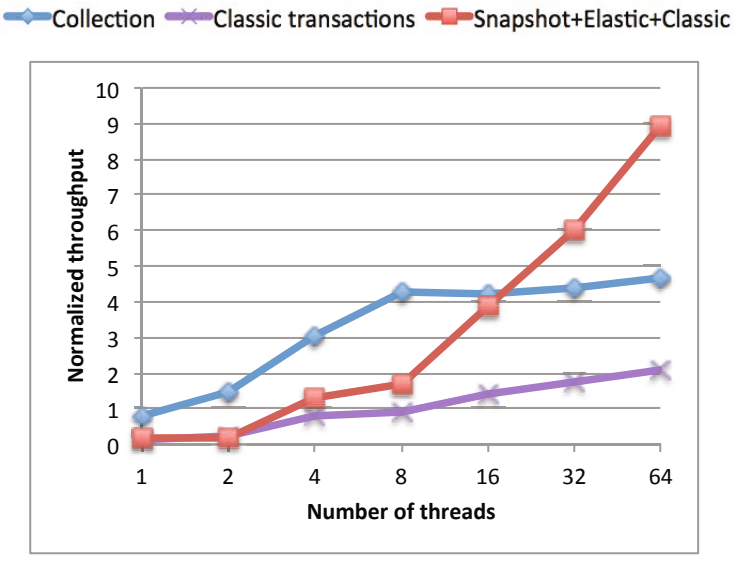


Fig. 9. Throughput (normalized over the sequential one) of the mixed transactions, the classic transaction and the collection package

5.2 Impact on Performance

The performance obtained when combining snapshot semantics in addition to the elastic one on the previous collection benchmark is depicted in Figure 9. The mixed transaction model performs 4.3× faster than the classic transaction model, TL2, and improves the concurrent collection package by 1.9× on 64 threads. Thanks to the snapshot semantics that remedy the scalability issue of the classic transactions, size operation in snapshot transactions commit more frequently than in default transactions. The reason is that a snapshot size returns potentially stale values that have been concurrently overwritten, while classic size would abort. Even though the overhead of the polymorphic transactions makes it slower than the concurrent collection package at low levels of parallelism, the performance scales well with the level of concurrency up to the maximum number of hardware threads we had at our disposal, and compensates the overhead effect at high level of parallelism.

6 Concluding Remarks

The transaction abstraction is in essence a middleware paradigm that allows multiple processes running on one or more processors (machines) to interact. The transaction abstraction was proposed long ago and has constituted an active area of research over the years.

Yet, transactions have not been widely adopted in practical concurrent and distributed programming and this is due, we believe, to their inherent cost and limited concurrency. In short, expert programmers need an alternative to bypass the simplicity of the concept and express their skills, potentially to obtain better performance.

We argue for democratizing the concept by enabling the co-existence of different semantics of it in the same application. Although a novice programmer will still be able to exploit the simplicity of the transaction abstraction in its default semantics, expert programmers would exploit, when possible, more expressive semantics of relaxed transaction models to gain in concurrency. This raises new challenges to guarantee that various semantics can cohabit smoothly in the same system but promises to further leverage the transaction abstraction.

References

1. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Commun. ACM 19, 624–633 (1976)
2. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26, 631–653 (1979)
3. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory. Morgan&Claypool (2010)
4. Liskov, B., Scheifler, R.: Guardians and actions: linguistic support for robust, distributed programs. In: POPL, pp. 7–19 (1982)
5. Liskov, B.: The Argus Language and System. In: Alford, M.W., Hommel, G., Schneider, F.B., Ansart, J.P., Lamport, L., Mullery, G.P., Zhou, T.H. (eds.) Distributed Systems. LNCS, vol. 190, pp. 343–430. Springer, Heidelberg (1985)
6. Almes, G.T., Black, A.P., Lazowska, E.D., Noe, J.D.: The eden system: A technical review. IEEE Trans. on Software Engineering SE-11(1), 43–59 (1985)
7. Guerraoui, R., Capobianchi, R., Lanusse, A., Roux, P.: Nesting Actions through Asynchronous Message Passing: the ACS Protocol. In: Madsen, O.L. (ed.) ECOOP 1992. LNCS, vol. 615, pp. 170–184. Springer, Heidelberg (1992)
8. Knight, T.: An architecture for mostly functional languages. In: LFP, pp. 105–112 (1986)
9. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21, 289–300 (1993)
10. Dice, D., Lev, Y., Moir, M., Nussbaum, D.: Early experience with a commercial hardware transactional memory implementation. SIGPLAN Not. 44, 157–168 (2009)
11. Diestelhorst, S., Hohmuth, M., Pohlack, M.: Sane semantics of best-effort hardware transactional memory. In: WTTM (2010)
12. Dalessandro, L., Carouge, F., White, S., Lev, Y., Moir, M., Scott, M.L., Spear, M.F.: Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In: ASPLOS, pp. 39–52 (2011)
13. Felber, P., Riviere, E., Moreira, W., Harmanci, D., Marlier, P., Diestelhorst, S., Hohmuth, M., Pohlack, M., Cristal, A., Hur, I., Unsal, O., Stenstrom, P., Dragojevic, A., Guerraoui, R., Kapalka, M., Gramoli, V., Drepper, U., Tomic, S., Afek, Y., Korland, G., Shavit, N., Fetzer, C., Nowack, M., Riegel, T.: The velox transactional memory stack. IEEE Micro 30(5), 76–87 (2010)

14. Shavit, N., Touitou, D.: Software transactional memory. In: PODC, pp. 204–213 (1995)
15. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC, pp. 92–101 (2003)
16. Dice, D., Shalev, O., Shavit, N.N.: Transactional Locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
17. Riegel, T., Felber, P., Fetzer, C.: A Lazy Snapshot Algorithm with Eager Validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 284–298. Springer, Heidelberg (2006)
18. Dragojevic, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. In: PLDI, pp. 155–165 (2011)
19. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: a benchmark for software transactional memory. In: EuroSys, pp. 315–324 (2007)
20. Harmanci, D., Gramoli, V., Felber, P.: Atomic Boxes: Coordinated Exception Handling with Transactional Memory. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 634–657. Springer, Heidelberg (2011)
21. Cascaval, C., Blundell, C., Michael, M., Cain, H.W., Wu, P., Chiras, S., Chatterjee, S.: Software transactional memory: Why is it only a research toy? Queue 6, 46–58 (2008)
22. Dragojevic, A., Felber, P., Gramoli, V., Guerraoui, R.: Why STM can be more than a research toy. Commun. ACM 54(4), 70–77 (2011)
23. Lynch, N.A.: Multilevel atomicity a new correctness criterion for database concurrency control. ACM Trans. Database Syst. 8 (1983)
24. Moss, J.E.B.: Open nested transactions: Semantics and support. In: WMPI (2006)
25. Reuter, A.: Concurrency on high-traffic data elements. In: PODS, pp. 83–92 (1982)
26. O'Neil, P.E.: The escrow transactional method. ACM Trans. Database Syst. 11, 405–430 (1986)
27. Felber, P., Gramoli, V., Guerraoui, R.: Elastic Transactions. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 93–107. Springer, Heidelberg (2009)
28. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA, pp. 73–82 (2002)
29. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N., Shavit, N.: A Lazy Concurrent List-Based Set Algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 3–16. Springer, Heidelberg (2006)
30. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP, pp. 48–60 (2005)
31. Ghemawat, S., Gobioff, H., Leung, S.-T.: The google file system. In: SOSP (2003)
32. Greenwald, M.: Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In: PODC, pp. 260–269 (2002)
33. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC, pp. 240–248 (2005)
34. Transactional Memory Specification Drafting Group: Draft specification of transactional language constructs for C++ (2009), http://software.intel.com/file/21569
35. Gramoli, V., Harmanci, D., Felber, P.: On the input acceptance of transactional memory. Parallel Processing Letters 20(1), 31–50 (2010)
36. Harris, T.: A Pragmatic Implementation of Non-Blocking Linked-Lists. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001)

37. Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., Holmes, D.: Java Concurrency in Practice. Addison-Wesley (2005)
38. Ni, Y., Menon, V., Abd-Tabatabai, A.-R., Hosking, A.L., Hudson, R.L., Moss, J.E.B., Saha, B., Shpeisman, T.: Open nesting in software transactional memory. In: PPoPP (2007)
39. Herlihy, M., Koskinen, E.: Transactional boosting: A methodology for highly-concurrent transactional objects. In: PPoPP (2008)
40. Harmanci, D., Gramoli, V., Felber, P., Fetzer, C.: Extensible transactional memory testbed. J. Parallel and Distrib. Comp. 70(10), 1053–1067 (2010)
41. Liu, Y., Spear, M.: Toxic transactions. In: Transact (2011)

Scaling Microblogging Services with Divergent Traffic Demands

Tianyin Xu[1], Yang Chen[1], Lei Jiao[1], Ben Y. Zhao[2], Pan Hui[3], and Xiaoming Fu[1]

[1] University of Goettingen
[2] U.C. Santa Barbara
[3] Deutsche Telekom Laboratories

Abstract. Today's microblogging services such as Twitter have long outgrown their initial designs as SMS-based social networks. Instead, a massive and steadily-growing user population of more than 100 million is using Twitter for everything from capturing the mood of the country to detecting earthquakes and Internet service failures. It is unsurprising that the traditional centralized client-server architecture has not scaled with user demands, leading to server overload and significant impairment of availability. In this paper, we argue that the divergence in usage models of microblogging services can be best addressed using complementary mechanisms, one that provides reliable messages between friends, and another that delivers events from popular celebrities and media outlets to their thousands or even millions of followers. We present *Cuckoo*, a new microblogging system that offloads processing and bandwidth costs away from a small centralized server base while ensuring reliable message delivery. We use a 20-day Twitter availability measurement to guide our design, and trace-driven emulation of 30,000 Twitter users to evaluate our Cuckoo prototype. Compared to the centralized approach, Cuckoo achieves 30-50% server bandwidth savings and 50-60% CPU load reduction, while guaranteeing reliable message delivery.

1 Introduction

In recent years, microblogging services such as Twitter have reached phenomenal levels of success and become a significant new form of Internet communication utility. Twitter, the most successful service, has more than 100 million users and generates more than 65 million "tweets" per day [23, 29]. In addition, Twitter usage peaks during prominent events. For example, a record was set during the FIFA World Cup 2010 when fans wrote 2,940 tweets per second in a 30-second period after a goal [21].

While originally designed as an online social network (OSN) for users with quick updates, the usage of Twitter has evolved to encompass a wide variety of applications. Twitter usage is so wide spread and pervasive that its traffic is often used as a way to capture the sentiment of the country. Studies have used Twitter traffic to accurately predict gross revenue for movie openings [9], even

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 20–40, 2011.

producing effective predictions for election results [12]. Still other projects have demonstrated that Twitter traffic can be mined as a sensor for Internet service failures [20] and even a real-time warning system for earthquakes [1].

In addition to these applications, Twitter usage by its users has also evolved significantly over time. Recent studies have shown that while many users still use it as a social communication tool, much of Twitter traffic today is communication from celebrities and personalities to their fans and followers [4, 17]. These asymmetric communication channels more closely resemble news media outlets than social communication channels. Because of the popularity of celebrities on Twitter (e.g., Lady Gaga, Britney Spears, and Justin Bieber account for over 30 million followers), these accounts are generating a large amount of traffic and placing tremendous load on Twitter's servers.

These major sources of traffic have a very tangible impact on the performance and availability of Twitter as a service. Despite the efforts to scale the system, Twitter has suffered significant loss in availability from malicious attacks and hardware failures [5, 8], and more frequently from traffic overload and flash crowds [25, 30, 31]. As short-term solutions, Twitter has employed per-user request and connection limits [11], as well as network usage monitoring and doubling the capacity of internal networks [31], all with limited success. Given the rapid and continuing growth of traffic demands, it is clearly challenging and likely costly to scale up with the demands using the current centralized architecture.

In this paper, we explore an alternative architecture for popular microblogging services such as Twitter. In our system, *Cuckoo*[1], our goal is to explore designs that leverage bandwidth and processing resources at client machines without sacrificing service availability or reliable delivery of contents. One of our insights is to recognize the two different roles these services play, those of an online social network and a news delivery medium. We use complementary mechanisms to address the dual roles while minimizing resource consumption. In the social network component, users are connected via mostly symmetric social links, and have a limited number of connections. Here, we allow a "publisher" or creator of a tweet to directly push the content to his (or her) friends via unicast. In the news delivery component, content producers are typically celebrities or media outlets, each connected via asymmetric links to a large number of followers. Given the large number of users with shared interests, we use gossip to provide highly reliable and load-balanced content delivery. Moreover, Cuckoo's delivery mechanisms support heterogeneous client access (e.g., mobile phone access) which is becoming increasingly common in microblogging services [19].

To ensure consistency and high data availability, Cuckoo uses a set of centralized servers to augment client peers in a peer-assisted architecture. This combination greatly simplifies data management challenges while reducing the server load. From an economic perspective, a Cuckoo service provider is still viable, because he (or she) will keep the master copies of all user contents, and can still generate revenue, e.g., by using content-based ads.

[1] We first outlined our idea in an earlier workshop paper [35].

We have implemented a Cuckoo prototype and made its source code and datasets publicly available[2]. We evaluated a small-scale deployment for 50 Twitter users running on 5 laptops as a demonstration [33]. In addition, we have conducted laboratory experiments using a detailed Twitter trace containing 30,000 users. We show that Cuckoo incurs 30-50% server bandwidth savings, 50-60% server CPU reduction compared with its centralized ilk, as well as reliable message delivery and efficient micronews dissemination between Cuckoo peers.

In summary, this paper makes three key contributions:

1. A novel system architecture for microblogging services to address the scalability issues, which relieves main server burden and achieves scalable content delivery by decoupling microblogging's dual functionality components.
2. A detailed availability measurement of Twitter during a flash crowd event.
3. A prototype implementation and trace-driven emulation of 30,000 Twitter users yielding notable bandwidth savings, CPU and memory reduction, as well as reliable message delivery and efficient micronews dissemination.

2 Background and Related Work

With immense and steadily-growing popularity over recent years, microblogging services have attracted considerable interests in the research community. We provide some background and summarize the state of the art.

Microblogging Model. The common model of microblogging services is the simplified publish-subscribe (Pub-Sub) model (c.f., [34]) based on the "follow" operation. The microblogging model is deceptively simple: The user can publish tweets within a length limit of viewable text (e.g., up to 140 characters in Twitter). The other users who have explicitly followed that user will receive all his (or her) tweets, i.e., being a *follower* means that the user will receive all the news from the *followees*. Currently, the microblogging model is implemented by using naïve polling for detecting updates in the centralized architecture.

There are several prior works on Pub-Sub systems that abandon the use of naïve polling, thus achieving high scalability and performance [22, 26]. Their key idea is cooperative polling between dedicated middleware mediators, named *brokers*. Microblogging differentiates from the traditional Pub-Sub systems by the system architecture. In microblogging, there is no always-on broker that collects events from publishers and sends notifications to subscribers. The key problem of microblogging is how to directly deliver publishers' tweets to their followers with divergent traffic demands. Cuckoo shares the insight with the prior works that the blind polling is the prime culprit of poor performance and limited scalability. Instead, Cuckoo enables user clients to share tweets in the peer-assisted fashion. On the other hand, Cuckoo interoperates with the current polling-based web architecture, requiring no change to legacy web servers.

Microblogging Measurement and Analysis. Microblogging services are widely recognized as online social network services for the explicit and implicit

[2] Cuckoo source code and selected datasets can be found at `http://mycuckoo.org/`.

social relations [11, 14, 16, 20]. For example, users exhibiting reciprocity (i.e., following each other) should be acquaintances, typical in OSNs. According to the "follow" relations, Krishnamurthy et al. identify distinct groups of users, e.g., broadcasters and evangelists [16]. Different social groups have different social behavior. Ghosh et al. study the relations and restrictions on the number of social links in microblogging, based on which a network growth model is proposed [11]. Java et al. report early observations of Twitter and analyze social communities formed by users with similar interests [14]. On the other hand, some researchers recently argue that microblogging, as exemplified by Twitter, serves more as news media outlets than OSN services [4, 17, 27]. Due to the one-sided nature of the "follow" relation, there are a small number of highly-subscribed users (e.g., celebrities and mass media) who have large numbers of followers and post far more tweets than the other users. These users generate the greatest per-capita proportion of network traffic and trend the trends.

One of Cuckoo's design rationales is to separate microblogging's dual components, i.e., social network and news media. Cuckoo employs different mechanisms towards scalable message delivery, gearing to the different dissemination models of the two components. Moreover, Cuckoo takes advantage of the inherent social relations to optimize system performance and information sharing.

Decentralized Microblogging and OSN Systems. There are several decentralized OSN systems proposed for different research concerns. FETHR [27] is a recently proposed microblogging system that envisions fully decentralized microblogging services. Its main idea is to let users directly contact each other via HTTP and employ gossip for popular content propagation. However, as a truly P2P system, FETHR cannot guarantee reliable data delivery since it does not consider the asynchronism of user access. As a result, some tweets will not get to users. Moreover, FETHR does not elaborate the gossip component nor implement it in its prototype. Other practical issues such as client heterogeneity support are also missing in FETHR. PeerSoN [2] is a prototype of P2P OSNs that uses encryption to protect user privacy against OSN providers and third-party applications. It uses dedicated DHT for data lookup, based on which direct user information exchanging can be achieved. Vis-à-Vis [28] is based on the concept of VIS, a kind of paid cloud-computing utility such as Amazon EC2 used for managing and storing user data. VISs self-organize into multi-tier DHTs representing OSN groups, with one DHT for each group. Safebook [6] is a decentralized OSN that aims at protecting users' security and privacy based on trust transitivity.

Cuckoo proposes a new system architecture tailored for microblogging services. It consists of two overlay networks with different content delivery mechanisms. The delivery mechanisms support heterogeneous client access by differentiating client types. On the other hand, since fully distributed P2P systems have hardly achieved success in terms of availability and reliability, Cuckoo employs a set of servers as a backup database that ensures high data availability and effectively eliminates the inconsistency due to the asynchronism of user access.

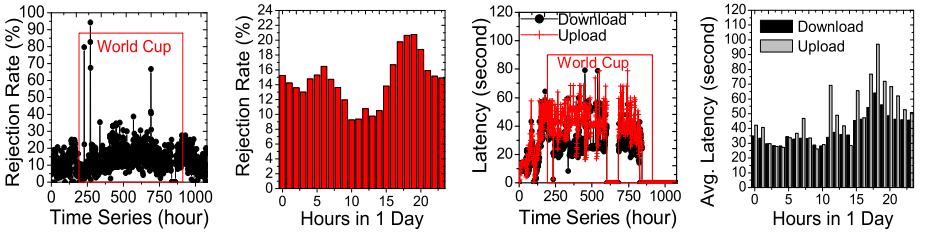


Fig. 1. Twitter measurement: Service rejection (a) in time series (Jun. 4–18, 2010), (b) in 24 hours; Response latency (c) in time series (Jun. 4–18, 2010), (d) in 24 hours

3 Measuring Availability at High Load

To provide concrete motivation for our work beyond the prior efforts, we conducted measurement studies on current microblogging systems. Our study includes a 20-day Twitter availability and performability [34] measurement and a user behavior analysis for over 300,000 Twitter users. In addition, we measure the system scalability of the generic centralized microblogging architecture.

Availability and Performability of Twitter. We first conducted a measurement study on the availability and performability of Twitter in terms of service rejection rate and response latency. The study was set in NET lab in Göttingen, Germany from 00:00, Jun. 4 to 23:00, Jul. 18, 2010, Berlin time (CEST), including the period of World Cup 2010 in the same time zone, which is regarded as Twitter's worst month since October 2009 from a site stability and service outage perspective [30]. We used JTwitter as the Twitter API to do the measurement.

For service rejection rate, we randomly selected a Twitter user and sent the request for his (or her) recent 200 tweets to the Twitter site every 5 seconds. If the Twitter site returns a 50X error (e.g., 502 error), it indicates that something went wrong (over-capacity in most cases) at Twitter's end and we count for one service rejection event. Fig. 1(a) shows the average service rejection rate per hour during our measurement period. We see that Twitter's availability was poor – the rejection rate was already about 10% in normal time. Moreover, the flash crowd caused by FIFA World Cup made an obvious impact on service rejection rate which increased from 10% to 20%. Since the flash crowd generated a significant surge over Twitter servers' capacity, the performance of the offered service degraded tremendously. Fig. 1(b) reports the average rejection rate for each hour in one day. We find that there existed some peak hours (e.g., 18:00 – 19:00) that had the worst performance in terms of service rejection.

For response latency, we measured both upload latency and download latency. Upload latency refers to the interval between sending a tweet to the Twitter site and receiving the ACK, while download latency is the interval between sending the request and receiving the required contents. For one measurement round, we first generated an artificial tweet by combining random characters in an predefined alphabet, posted it on Twitter and recorded the upload latency. Then, we requested the posted tweet from Twitter and recorded the download

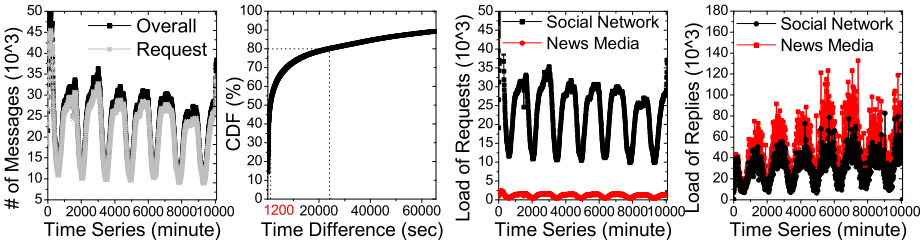


Fig. 2. User access patterns: (a) # of request messages to the servers; (b) Time differences of two adjacent tweets; (c) Incoming traffic load; (d) Outgoing traffic load

latency. Such round was repeated every 5 seconds. Similar as Fig. 1(a) and 1(b), Fig. 1(c) and 1(d) shows the measured response latency of the Twitter service[3]. No surprisingly, Twitter's performability, in terms of response latency, was unsatisfactory especially during World Cup with the download latency about 200 seconds and upload latency about 400 seconds. Twitter engineers also noticed this outage and poor site performance [30, 31], their solutions include doubling the capacity, monitoring, and rebalancing the traffic on their internal networks, which do not scale well with the unprecedented growth.

User Access Pattern Analysis. To further study the server load according to user access patterns of Twitter services, we analyze large-scale Twitter user traces. We collected 3,117,750 users' profile, social relations, and all the tweets maintained on the Twitter site. Using 4 machines with whitelisted IPs, we used snowball crawling that began with the most popular 20 users reported in [17] using Twitter API. The crawling period was from Mar. 6 to Apr. 2, 2010. In this paper, we focus on the user access patterns in the 1-week period from Feb. 1 to Feb. 7, 2010. To simplify the problem and yet accurately represent the traffic patterns of Twitter services, we consider two built-in Twitter's interaction models: *post* and *request*. The polling period is set as one minute according to the setting options of common Twitter clients (e.g., Ambientweet, Gwibber, Osfoora) [34]. For session durations, we use the duration dataset provided in [13]. The details of the above datasets and data processing are described in Section 5.1.

Fig. 2(a) shows the server load in terms of the number of received messages on the server side. We can see that over 90% are request messages which make up the dominating traffic proportion. Specially, at leisure time when users post fewer tweets, the request messages almost occupy the whole traffic. One objective of Cuckoo is thus to eliminate the unnecessary traffic caused by these polling requests. Fig. 2(b) is the cumulative distribution function (CDF) of the time differences between two adjacent tweets of each user. Although the burstyness of human behavior leads to tweets with small time intervals, there are still 50% of time differences larger than 1200 second and 20% larger than 24,000 second. In the worst case that the polling requests are fully scheduled in these intervals, the resource waste due to unnecessary traffic is tremendous.

[3] The gaps in Fig. 1(c) is due to server cutoffs during the measurement period.

We further analyze the traffic load by separating it into social network usage and news media usage. The separation is based on the observations of previous studies [4, 17, 27] which report that there are two kinds of users in microblogging: social network users and news media outlets. We regard users having more than 1000 followers as *media users* and the others as *social users*. The threshold 1000 is chosen according to the homophily analysis in [17] which reports that in Twitter only users with followers 1000 or less show assortativity, one of the characteristic features of human social networks. There are 3,087,849 (99.04%) social users and 29,901 (0.96%) media users among all users. Tweets posted by media users are identified as news media usage while social users' tweets are regarded as social network usage. Fig. 2(c) shows the incoming traffic load in terms of received messages. For a request message, we calculate the percentage of media users among the requester's followees as news media usage and the rest percentage as social network usage. From Fig. 2(c), we find that the social network usage occupies the dominant proportion of incoming traffic load – about 95% of incoming load is for social network usage while less than 5% is for news media. Fig. 2(d) reports the outgoing traffic load in terms of replied tweets. For each tweet within a reply message (reply to a request), we identify it into social network or news media according to whether its publisher is a media user or a social user. We can see from Fig. 2(d) that although news media usage holds small proportion of server requests (Fig. 2(c)), it occupies a great proportion of outgoing traffic load, with 1.66 times on average more than the proportion of social network usage. Thus, the dual functionality components of microblogging have divergent traffic patterns, and the mix of them at the same time makes the system using a single dissemination mechanism hard to scale.

Scalability of the Generic Centralized Microblogging System. To study the scalability of the generic centralized microblogging system, we treat Twitter as a black box and reverse engineer its operations based on Twitter traces because the details of Twitter's implementation remain proprietary. Still, we consider *post* and *request* as the main interaction models. Each user interaction is implemented through one or more connections with centralized servers. For example, to post a new tweet, a user opens a TCP connection with one server, sends the tweet message, and then receives ACK to display. On the other hand, users detect updates by periodically polling through established connections.

We use the Twitter trace described in the previous section to evaluate the scalability of the centralized microblogging architecture. We employ Breadth First Search (BFS) as the graph search algorithm with the start user Ustream who has over 1,500,000 followers. We prepare 4 datasets for 10,000, 30,000, 50,000, and 100,000 users respectively and prune the social links outside the datasets. We set the polling period to one minute. We run 4 server programs on a Dell PowerEdge T300, with four 2.83 Ghz quad-core 64-bit Intel Xeon CPU and 4GB of RAM. To measure CPU and memory usage of the server machine, we use the statistics provided by `vmstat` utility. For traffic usage, we use `bwm` utility to record incoming and outgoing bandwidth in every 5 seconds.

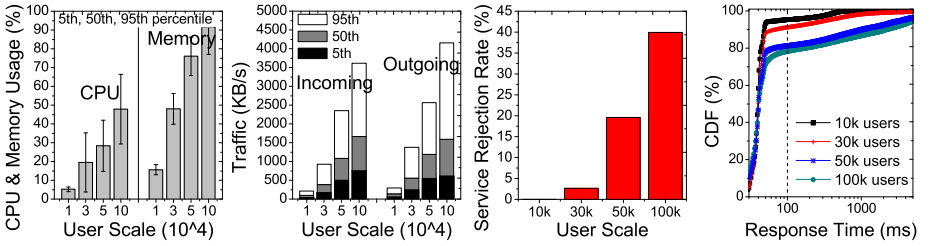


Fig. 3. Scalability of the centralized microblogging architecture: (a) CPU and memory usage; (b) Traffic usage; (c) Service rejection rate; (d) Response latency

Fig. 3 demonstrates the limited system scalability of the centralized architecture in terms of CPU and memory usage, traffic usage, service rejection rate, and response time. For CPU usage, memory usage, and traffic usage (Fig. 3(a) and Fig. 3(b)), we can find the linear growth of these metrics with the increasing number of users. For example, the 50th percentile of CPU usage is 5.2%, 19.5%, 28.4%, and 47.9% for user scale 10,000, 30,000, 50,000, and 100,000 respectively. Fig. 3(c) shows service rejection rate with different user scales. When user scale is 10,000, the servers can handle all the requests even at peak time. Thus, the server rejection rate is almost 0 for 10,000 users. For 30,000 users, it is difficult for the servers to satisfy all the requests at peak time and the service rejection rate is lower than 5%. But for 100,000 users, even in regular time, the servers are overloaded so that the rejection rate is extremely high. Fig. 3(d) shows the CDF of the round-trip response time of the service with different user scales. We can see that the response latency is greatly impacted by the scale of users. When the user scale is 10,000, over 90% of requests are satisfied within 100 ms. While the user scale increases to 30,000, only 80% of requests have the response time less than 100 ms. For 50,000 user scale, the servers have to reject most user requests to avoid getting exhausted and keep response to limited number of requests.

In summary, we make three key observations from our measurement studies. First, the centralized architecture has limited scalability with the increasing number of users. Second, the main server load and traffic waste are caused by the polling requests. Third, the social network and news media components of microblogging have divergent traffic patterns, which makes the system using a single dissemination mechanism hard to scale. Thus, there is a significant opportunity to eliminate the server burden and traffic waste towards high scalability by abandoning polling and decoupling the dual functionality components.

4 Design

In this section, we first present Cuckoo's system architecture and explain how Cuckoo is geared to the characteristics of microblogging services. Next, we describe the building blocks of Cuckoo, which put together the whole system.

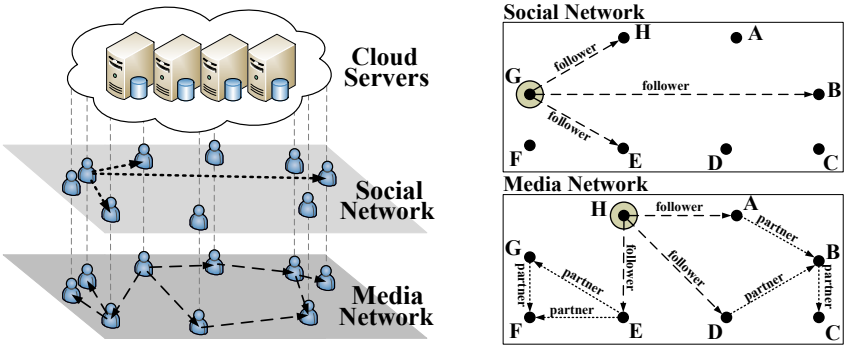


Fig. 4. Cuckoo: (a) System architecture; (b) Complementary content delivery

4.1 System Architecture

Decoupling the Two Components. The biggest reason that microblogging systems like Twitter do not scale is because they are being used as both social networks and news media infrastructures at the same time. The two components of microblogging have divergent traffic and workload patterns due to their different dissemination models. As discussed in Section 3, although the social network component occupies more than 95% request load, the news media component holds greater proportion of dissemination load, 1.66 times more than that of the social network. On one hand, the social network component is made up of most of users with limited numbers of followers. It is reported in [27] that half of Twitter users have 10 or fewer followers, 90% have less than 100 followers, and 95% have less than 186 followers. Moreover, social users do not generate much per-capita traffic. The three-week Twitter trace in [27] shows that most users sent about 100 messages during that period. On the other hand, the news media component is initiated by a small number of highly-subscribed users and then broadcasted to large numbers of other users. The study on entire Twittersphere [17] shows that there are about 0.1% users with over 10,000 followers. There are only 40 users with more than a million followers and all of them are either celebrities or mass media. Besides, media users post tweets (named *micronews*) much more frequently than social users. The correlation analysis in [17] shows that the number of tweets grows by an order of magnitude for the users with number of followers greater than 5000. Due to the sharp gap between the dissemination models of microblogging's two components, there is no single dissemination mechanism can really address these two at the same time.

Cuckoo effectively addresses both dissemination models by decoupling the two functionality components and using complementary mechanisms. Fig. 4(a) shows the high-level architecture of Cuckoo which includes two kinds of logical overlay networks formed by Cuckoo peers at the network edge. For the social network component, a *social network* is formed where each publisher peer sends new tweets directly to all its follower peers in the unicast fashion. For the news media component, Cuckoo peers with the same interest form a *media network* and

use gossip to disseminate micronews, i.e., enabling followers to share micronews with each other. The two overlay networks are geared to the two dissemination models of microblogging's dual components. For social users with limited numbers of followers, the one-to-one unicast delivery is simple and reliable. While for news media, no single peer can afford delivering micronews to large numbers of news subscribers. For example, in Twitter, Lady Gaga has more than 10.4 million followers. If using unicast-like delivery, it will take at least several hours to disseminate only one tweet, not to mention the overload of the sender. Thus, it is necessary to let interested peers be involved in the micronews dissemination. Fig. 4(b) demonstrates Cuckoo's complementary content delivery mechanisms corresponding to the two overlay networks. Besides, Cuckoo employs a set of stable peers to form a DHT (Distributed Hash Table), e.g., [24, 36] that maintains all the users' connection information (e.g., IP address, port) named *node handlers* (NHs). Thus, distributed user lookup is realized: Firstly, a peer can find any other user's connection information in less than $O(log(N))$ hops on average in an N-node DHT. Secondly, we use DHT-based random walks to provide efficient partner information collection for gossip dissemination.

Combination of Server Cloud and Client Peers. As shown in Section 3, centralized microblogging systems such as Twitter impose high server load and traffic waste, which makes centralized servers to be the performance bottleneck and central point of failure. Thus, the traditional centralized client-server architecture is hard to scale. Meanwhile, truly decentralized P2P systems have earned notoriety for the difficulties coping with availability and consistency, and thus achieved limited success in the past. For example, FETHR [27] provides no guarantee on data delivery. The FETHR peer can receive tweets posted during its online duration while missing most tweets posted at its offline time. The follow operation will also be crippled if the potential followee is not online.

Cuckoo incorporates the advantage of both centralized and decentralized architectures by the combination of a small server base (named *server cloud*) and client peers (i.e., *Cuckoo peers*). In Cuckoo, the server cloud plays important roles including ensuring high data availability and maintaining asynchronous consistency for peers. Besides the content delivery on the two overlay networks, the Cuckoo peers also upload their new tweets and social links to the server cloud, based on which each peer performs consistency checking at bootstrapping to detect missing events during its offline period. By abandoning naïve polling and offloading the dissemination operation cost, the server cloud gets rid of the main server load towards high scalability. On the other hand, the servers still keep their original functions to support other operations which do not lead to performance bottleneck such as tweet searching. On the rare occasion when the server cloud is unavailable (e.g., outage [25], under attack [5]), Cuckoo peers can still find and communicate with each other. Moreover, information loss in a single location [8] can be easily recovered, since in Cuckoo both service providers and users possess the data ownership. From the service providers' perspective,

Cuckoo lets them keep the same resources as in centralized systems, which is the basis of their business. In addition, Cuckoo is backward-compatible with the polling-based web architecture, requiring no special feature on the server side.

4.2 Social Relations

We describe how Cuckoo peers maintain the social relations to form the two overlay networks, and how "follow" is operated to build these relations.

Social Relation Maintenance. In typical microblogging services, a user has the following social relations: *followee*, *follower*, and *partner*. To form the social network, each peer maintains the followee and follower information in its local database. The follower information is maintained according to whether the peer is a social user or a media user. The social user who has only a few followers maintains all the followers' information, while the media user with large numbers of followers maintains only a logarithmic subset. Thus, the number of entries e_i in user i's follower list can be presented as: $e_i = max(min(F_i, H), log(F_i))$, where F_i denotes the number of followers of user i, and H is the threshold to separate social users and media users. To form the media network, the Cuckoo peer maintains sets of partners corresponding to the media users it follows (partners are only needed for media followees). Each Cuckoo peer collects and updates partner information using the DHT-based random walk mechanism. Note that the more people a user follows, the more information the user has to maintain so as to join multiple dissemination groups, which to some extent suppresses the behavior of evangelists (e.g., spammers or stalkers) [16].

Follow. The "follow" operations explicitly build the followee-follower relations between user pairs, which forms the basis of the social network. To follow a specific user, the Cuckoo peer first lookups the followee's NH according to his (or her) `userId`[4] via the DHT. The DHT maintains all users' NHs in the key-value format (`userId` as key and NH as value). Then, the peer sends a follow request that attaches its profile to the followee peer using the NH. There are 2 cases according to whether the followee peer is online or not: (1) If the followee peer is online, it receives the request and sends back a reply directly to the requester. After receiving the reply, the follower sends a notification to the server cloud to inform the built relation; (2) If the followee peer is offline, the requester submits its willing to the cloud. The cloud checks the validity and replies the results. Each Cuckoo peer checks the inconsistency between the follower list maintained locally and the one maintained by the server cloud at online bootstrapping. If there exist some new followers during its offline period, the peer sends replies as compensation. The consistency checking does not require complete comparison of the two follower lists. As long as the server cloud maintains users' follower list in reverse chronological timeline like Twitter, the Cuckoo peer is able to send the cloud its last recorded follower's `userId` and get back the new guys.

[4] In Cuckoo, each user is assigned a unique `userId` by the server cloud at registration, which simplifies the authentication and Id assignment.

4.3 Unicast Delivery for the Social Network

When a user posts a new tweet, the microblogging service should guarantee that all the users' followers could receive that tweet. For the social network where users' social links are limit in size (e.g., a few hundred followers), serial unicast-like content delivery is simple and reliable. The publisher peer tries to push the newly posted tweet via direct unicast socket to each follower. This is achieved by locally caching each follower's latest node handler (NH). To ensure that followee peers always keep the up-to-date NHs, a user informs all his (or her) followees when changing the NH, e.g., in the case that the user accesses the service using different devices in different places. Moreover, the user is required to update the NH replicas in the DHT so that any other user can search up-to-date NHs.

The unicast-like delivery for the social network can ensure all the online followers to receive their followees' new updates in time. However, for offline followers being absent from the delivery process, they should regain the missing tweets when re-entering the system. Cuckoo achieves this also by consistency checking, i.e., each peer fetches the bunch of tweets posted at its offline period from the server cloud at bootstrapping. Since tweets are maintained in the reverse chronological timeline, a new coming user's missing parts can be efficiently detected by giving his (or her) last departure time or the `statusId`[5] of his (or her) last received tweet. This checking process is also applicable for media users in the media network. Note that the consistency checking is only used to detect missing tweets and social links, not to check the NHs maintained in the DHT.

4.4 Gossip Dissemination for the Media Network

In the media network, media users cannot afford sending updates to all their followers. In this case, Cuckoo uses gossip-based dissemination, i.e., enable interested users to be involved in the micronews dissemination process. Gossip information dissemination has been proved to be scalable and resilient to network dynamics. The theoretical support provided in [15] proves if there are n nodes and each node gossips to $log(n) + c$ other nodes on average, the probability that everyone gets the message converges to $e^{-e^{-c}}$, very close to 1.0 without considering the bandwidth constraint, latency, failure, etc. This result provides a guideline for Cuckoo's partner management, i.e., maintain the number of partners (called *fanout*) to be logarithmic of the number of followers.

To discovery online partners in case of churn, we design a DHT-based partner collection mechanism which is elaborated in [34]. We sketch our basic idea as follows. The joining peer picks a random `nodeId` X and asks a bootstrap node to route a special *hello message* on the DHT using X as the destination key. The hello message announces the new peer's presence as well as its interests on media users (i.e., its media followees' `userIds`). This hello message is routed to the DHT node with `nodeId` numerically closest to X. Meanwhile, the nodes along the DHT route overhear the message and check the new node's interests. In this way, the stable nodes in the DHT construct probabilistic follower indices

[5] In Cuckoo, each tweet is bounded to a unique `statusId`.

of media users. To look for new online partners of a media user, a Cuckoo peer uses DHT-based random walks [3] to collect partners over the DHT topology by checking these indices. The DHT-based random walk ensures that nodes in the DHT are visited only once during a collection process. Since the media users have high popularity, the random walk-based collection is efficient [34].

Cuckoo adopts the simple "infect and die" model [7] for micronews dissemination. Peers, once infected, remain infectious for one round precisely, before dying, i.e., the peer followed a media user gossips each of the followee's tweet (i.e., micronews) exactly once, namely after receiving that tweet for the first time, but will not further gossip even when receiving subsequent copies of the same tweet. Initially, the media user sends a gossip message containing the micronews to a subset of its online followers. Upon receiving the gossiped message, the Cuckoo peer determines whether it has received this micronews or not by checking the `statusId` of the tweet. For a new tweet, the peer saves it locally and continues gossiping to the $log(n) + c$ partners, where n is the number of all the online followers of the media user. Otherwise, the peer discards the message and takes no action. In this case, the micronews is disseminated within the circle of interested peers, i.e., the followers. The number of gossip rounds R, i.e., the network hops necessary to spread a micronews to all the online followers respects the equation [7]: $R = \frac{log(n)}{log(log(n))} + O(1)$, which shows that it takes at most a logarithmic number of steps for a micronews to reach every online follower.

4.5 Support for Client Heterogeneity

User clients in deployed microblogging systems are heterogenous with different bandwidth, energy, storage, processing capacity, etc. For example, the CEO of Twitter recently stated that over 40% of all tweets were from mobile devices, up from only 25% a year ago [19]. Thus, it is important to support client heterogeneity in Cuckoo, considering the economic burden of increased load on mobile peers such as higher cost for network usage (due to expensive or limited mobile data plans), and higher energy consumption resulting in reduced battery life.

Cuckoo differentiates user clients into three categories named *Cuckoo-Comp*, *Cuckoo-Lite*, and *Cuckoo-Mobile*. Cuckoo-Comp is designed for *stable nodes* which reside in the system for a relatively long time (more than 6 hours per day in our experiments). These stable Cuckoo-Comp peers construct the DHT to support user lookup and partner collection. The stable nodes are only a small subset of all the nodes (about 15% in our dataset), but their relatively long life spans allow them to keep the DHT stable with low churn. Several mechanisms can be integrated to identify stable nodes in overlay networks, e.g., the nodes already with higher ages tend to stay longer [32]. Cuckoo-Lite is designed for lightweight clients (e.g., laptops with wireless access) while Cuckoo-Mobile is for mobile devices (e.g., smart phones). Neither of them joins the DHT and the main difference between them is that Cuckoo-Mobile peers do not participate in the gossip dissemination process in the media network while the Cuckoo-Lite do (as the Cuckoo-Comp). Since mobile devices have energy and bandwidth constraints, they have no incentive to further forward the received messages. Thus,

we regard the Cuckoo-Mobile peers as *leaf nodes*. We call both Cuckoo-Comp and Cuckoo-Lite peers as *gossip nodes* that can use heterogeneity-aware gossip (e.g.,[10]) to tune the fanout. The dissemination is initiated from the publisher, gossiped through the Cuckoo-Comp and Cuckoo-Lite peers, and simultaneously spread to Cuckoo-Mobile peers. The details of dissemination with heterogenous clients can be found in [34] including the solutions to support text message based phone clients and clients behind NATs (Network Address Translations).

4.6 Message Loss Detection and Security Issues

Detecting Lost Tweets. While Cuckoo's gossip-based probabilistic dissemination achieves high resilience to node failures as well as high coverage rate, it provides no guarantee that each tweet is reached to all the followers due to the intrinsic uncertainty brought by the randomness. Thus, we need a mechanism to detect which tweet fails to arrive. Cuckoo exploits the `statusId` to solve this problem. In Cuckoo, each tweet is bounded to a unique `statusId`. The `statusId` is made up of two parts. The prefix is the `userId` of its publisher and the postfix is a `long` sequence number maintained by the publisher's counter. By checking `statusIds` of received tweets, the Cuckoo peer can easily identify gaps between the sequence numbers of `statusIds`. Then, the Cuckoo peer could fetch the lost tweets from either other peers by content query [34] or the server cloud.

Security. The presence of malicious users (e.g., attackers, spammers) requires additional components to safeguard against various kinds of security threats. Although the main concern of this paper is not on the security aspect, we introduce the basic ideas of Cuckoo's security components in this section and refer the details to [34]. To defend against spam distributed by the malware that impersonate normal users, the digital signature based on asymmetric key cryptography is attached within each message for authentication. Moreover, these digital signatures are capable of defending against violating message integrity by altering the contents of tweets. For DoS attacks and content censorships that target on crippling the server cloud [5], Cuckoo exploits its distributed nature: peers can still deliver/disseminate messages via the social/media network, and mark the blocked tweets for further uploading. For the brute-force attacks where malicious nodes generate unwanted traffic to harass normal peers' operations, trust and reputation mechanisms can be employed. The social relations maintained on each peer provide nature trust relationships to build the reputation model, based on which unwanted communications can be thwarted [18].

5 Experimental Evaluation

To evaluate Cuckoo's performance, we run Cuckoo prototype using trace-driven emulation of the Twitter trace containing 30,000 users which reconstructs the part of Twitter's traffic patterns from Feb. 1 to Feb. 7, 2010. We evaluate Cuckoo in two aspects: the performance gain of the server cloud, as well as the performance of message sharing and micronews dissemination between Cuckoo peers.

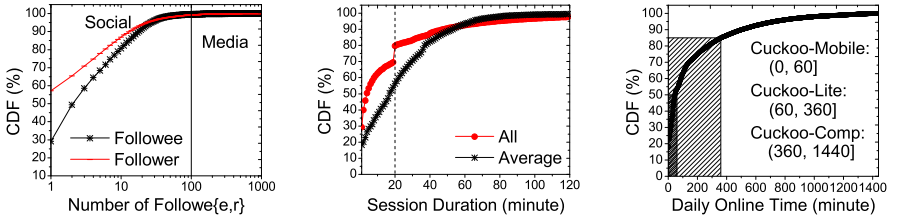


Fig. 5. Dataset: (a) Number of followe{e, r}s; (b) Session duration; (c) Online time

5.1 Experiment Settings

Dataset. We use the raw dataset described in Section 3 to evaluate Cuckoo's performance. The raw dataset contains 3,117,750 users' profiles, social links, and all the tweets maintained on the Twitter site[6] from Mar. 6 to Apr. 2, 2010. Still, we focus on user access patterns during the 1-week period from Feb. 1 to Feb. 7, 2010. We use BFS with the start user Ustream to create the experiment dataset containing 30,000 users' information to match the capacity of the emulation testbed. For social links, we prune the irrelevant links outside the dataset. Fig. 5(a) shows the CDF of users' followee/follower number. We separate the social users and media users according to each user's follower number with the threshold H as 100 instead of 1000 in Section 3 due to the social link pruning, and get 29,874 (99.58%) social users and 126 (0.42%) media users. We use the OSN session dataset provided by [13] to conservatively emulate Cuckoo users' session durations because so far no microblogging service provides user session information or user online/offline status information. Fig. 5(b) plots the CDF of the average session duration of each user and the CDF of all the durations. We classify the three types of Cuckoo users according to their daily online time, i.e., Cuckoo-Comp users are those whose daily online time exceeds 360 minutes, and Cuckoo-Mobile users spend less than 60 minutes online per day, and the others are Cuckoo-Lite users. Fig. 5(c) shows the CDF of the daily online time and the classification. We can see that about 50% Cuckoo peers are Cuckoo-Mobile clients. The details of data processing and dataset analysis can be found in [34].

Implementation. We have built a Cuckoo prototype using Java. Our implementation comprises both the Cuckoo peer and the server cloud. The prototype of Cuckoo peer adds up to 5000 lines of Java code including the three types of clients with different software components. We use Java socket to implement end-to-end message delivery, and define different types of application-layer messages. For local data management, we use XML to store node states and social links including followee/follower profiles, followee tweets, and self-posted tweets. We choose Pastry [24] and its implementation FreePastry as our overlay infrastructure for Pastry's good properties (e.g., locality awareness) as well as FreePastry's platform independence (Java source code). Note that Cuckoo does not rely on any Pastry's special feature (e.g., leaf set), so it is applicable for any structured

[6] Twitter only reserves about 3000 tweets per user and discards the previous tweets.

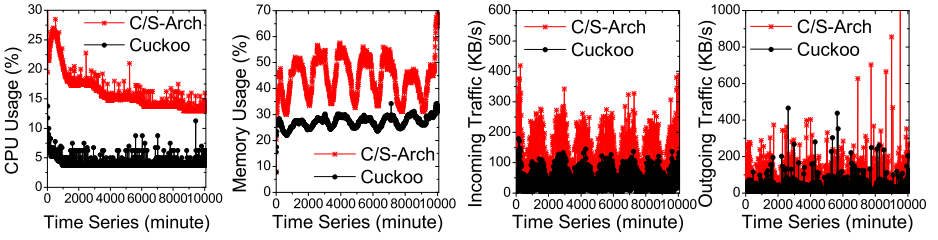


Fig. 6. Resource usage of the server cloud in time series (from Feb. 1 to Feb. 7, 2010): (a) CPU; (b) Memory; (c) Incoming traffic; (d) Outgoing traffic

overlay that supports the key-based routing. The server cloud prototype adds up to 1500 lines of Java code. It uses plain text files to store all users' information.

Deployment. We deploy 30,000 Cuckoo peers without GUI on 12 machines including 3 Dell PowerEdge T300, each with four 2.83 Ghz quad-core 64-bit Intel Xeon CPU and 4 GB RAM, and 9 Dell Optiplex 755, each with two 3.00 Ghz Intel Core 2 Duo CPU and 3.6 GB RAM. We deploy four servers to build the server cloud on another Dell PowerEdge T300 machine and let them share storage, so that these servers have no inconsistency problem. We locate these machines into two LANs connected by a 10 Gb/s Ethernet cable. We run the Cuckoo peers based on the 1-week Twitter trace described in the previous section. Still, we use `vmstat` utility to measure CPU usage and memory usage for the cloud machine, and `bwm` utility to record server bandwidth in every 5 seconds.

5.2 Server Cloud Performance

To characterize the performance gain of Cuckoo, we compare the performance of the server cloud under the Cuckoo architecture with that under the traditional client-server architecture (c.f., Section 3), denoted as "C/S-Arch". Remember that Cuckoo is fully compatible with the current polling-based web architecture, i.e., Cuckoo does not require any extra functionality on the server side. Thus, the server cloud implementations for Cuckoo and C/S-Arch are exactly the same. For fair comparison, both of them use the same machine and system configuration.

Resource Usage. We characterize the resource usage of the server cloud in Cuckoo compared with that in C/S-Arch in terms of CPU usage, memory usage, as well as incoming and outgoing bandwidth usage. Fig. 6(a) shows the server cloud's CPU usage of Cuckoo and C/S-Arch in time series. We can see that Cuckoo achieves notable reduction of CPU usage compared with C/S-Arch – the server cloud in Cuckoo consumes 60% less CPU than C/S-Arch, with the average value being 5%. The main usage of CPU is for the database lookup and I/O scheduling. Since the server cloud in Cuckoo receives far less requests than that in C/S-Arch, the CPU usage reduction is not surprising. Fig. 6(b) shows the server cloud's memory usage of Cuckoo and C/S-Arch in time series. Compared with CPU, memory usage is more sensitive to the number of requests. As a result of message overhead savings, Cuckoo effectively reduces the memory

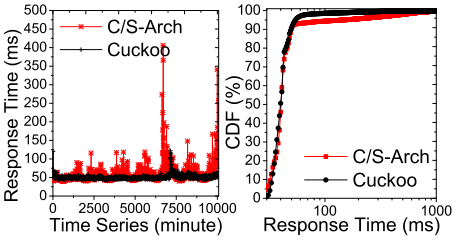


Fig. 7. Response latency (a) in time series, (b) in CDF

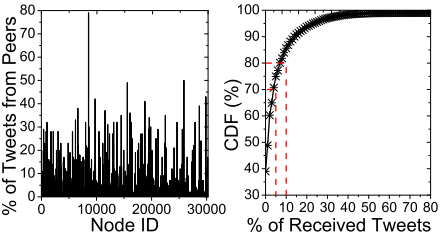


Fig. 8. Percentage of received tweets (a) in time series, (b) in CDF

usage compared with C/S-Arch. The server cloud of C/S-Arch consumes 50% of memory at peak time and 30% at leisure time, while the Cuckoo cloud's memory usage is around 25%. In summary, Cuckoo achieves about 50%/16% of memory usage reduction for the server cloud at peak/leisure time. Fig. 6(c) and Fig. 6(d) demonstrate the server cloud's incoming and outgoing bandwidth usage of Cuckoo and C/S-Arch in time series. The bandwidth usage is directly decided by the message overhead. The larger volume of messages the server cloud receives/sends, the more bandwidth is consumed. Cuckoo effectively reduces the incoming and outgoing bandwidth consumed, with about 120 KB/s, 70 KB/s at peak, leisure time for incoming bandwidth, about 200 KB/s, 90 KB/s at peak, leisure time for outgoing bandwidth. The incoming, outgoing bandwidth for C/S-Arch is 300 KB/s, 400 KB/s at peak time and 400 KB/s, 100 KB/s at leisure time. Thus, the server cloud of Cuckoo saves about 50% of bandwidth consumed for both incoming and outgoing traffic compared with that of C/S-Arch.

Response Latency. We examine the response latency provided by the server cloud of Cuckoo and C/S-Arch. Fig. 7(a) shows the response latency in time series and Fig. 7(b) shows the CDF of all the recorded response latency. We can see that at leisure time, the response latency of Cuckoo and C/S-Arch is similar (about 50 ms). However, at peak time, the response latency of Cuckoo is far less than that of C/S-Arch. The response latency of Cuckoo is relatively smooth, being around 50 ms in most time, while at peak time the response latency of C/S-Arch is more fluctuant and higher that can reach 100 ms or more. Since in Cuckoo the peak-valley difference of message overhead is smaller than that in C/S-Arch in terms of CPU, memory usage as well as bandwidth consumed, even at peak time the server cloud has enough resources to satisfy all the requests and posts. In contrast, at peak time, the server cloud of C/S-Arch has too much burden so that it can hardly satisfy all the concurrent requests at the same time.

5.3 Cuckoo Peer Performance

In this section, we characterize the performance of Cuckoo peers. Each peer maintains a message log that records all the received, sent, and forwarded messages. By collecting these logs from Cuckoo peers, we analyze the performance of message sharing. Moreover, the message logs provide the detailed information of

disseminated messages including `statusIds` of tweets, network hops, and redundancy. Based on these, we analyze the performance of micronews dissemination.

Message Sharing. Fig. 8 shows the performance of message sharing between Cuckoo peers. Fig. 8(a) shows for each peer, the percentage of tweets received from other Cuckoo peers other than from the server cloud, while Fig. 8(b) is the CDF of the percentages. According to Fig. 8(a) and 8(b), around 30% users get more than 5% of their overall subscribed tweets from other peers, and around 20% get more than 10%. The performance of message sharing is mainly decided by two aspects: users' access behavior and users' online durations. Generally speaking, the more overlapped behavior the followee-follower pairs have, the higher probability that follower peers could receive tweets from followees. For online durations, the longer the user stay online, the higher probability he (or she) can receive tweets from other peers. In the extreme case that a user keeps online all the time, he (or she) cannot miss any subscribed tweet without fetching from the server cloud. We should note that the duration dataset used in our experiments leads to a pessimistic deviation of the message sharing performance. Due to the centralized architecture of existing OSN services, the OSN operators employ *session timeout* to reduce users' polling so as to mitigate server load. In our duration dataset, the timeout is set as 20 minutes (see the vertical jump in Fig. 5(b)) [13], i.e., a user session is supposed to be disconnected as long as the user has no interaction (e.g., post tweets, follow friends) with the server in 20 minutes. However, the situation is completely opposite in Cuckoo where users are highly encouraged to stay online as long as they can. The long online durations of users can significantly improve the performance of message sharing without performance degradation of the server cloud (no polling any more). Thus, we can expect the better performance of message sharing in Cuckoo.

Micronews Dissemination. We evaluate the performance of micronews dissemination in the media network. Media users who have more than 100 followers use gossip to disseminate tweets, i.e., *micronews* (see Fig. 5(a)). In our experiment, each Cuckoo-Comp or Cuckoo-Lite peer (i.e., gossip node) maintains the fanout $f = T$ for one media followee, where $T = log(n) + 2$. In addition, due to the mobile spreading mechanism [34] that delivers tweets to leaf nodes, a gossip peer is likely to maintain some Cuckoo-Mobile partners. Since leaf nodes occupy 50% of all the nodes in our dataset (Fig. 5(c)), a gossip peer maintains at most $2T$ partners for gossip and mobile spreading. For instance, the media user Ustream with 29,927 followers sends no more than 24 messages for each dissemination process. Fig. 9 demonstrates the performance and overhead of Cuckoo's micronews dissemination in terms of coverage, network hops, and redundancy. The coverage and redundancy are conflict with each other impacted by the fanout. Larger fanout leads to higher coverage while imposing higher redundancy. Thus, the fanout should be chosen carefully to tradeoff the two metrics.

For the ease of presentation, we select three typical media users to illustrate the dissemination performance in terms of coverage rate. Fig. 9(a) shows the coverage rate of each dissemination process in time series. In this figure, the media user Ustream, Etsy, jalenrose with 29,927, 6,914, 696 followers publishes

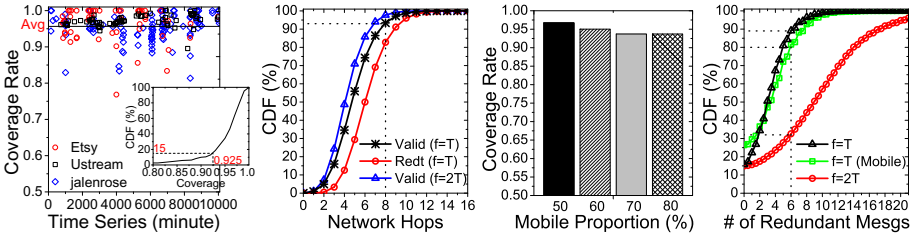


Fig. 9. Micronews dissemination: (a) Coverage; (b) Average network hops; (c) Coverage rate with different proportion of mobile nodes; (d) Average redundant messages

61, 59, 117 tweets respectively in the one-week period. We can see from the CDF that around 85% dissemination processes cover more than 92.5% of all the online followers of the media users, with the average coverage rate equal to 96.7%. All the dissemination processes of the three media users have coverage rate higher than 70%, and there is only few process with coverage rate lower than 80% due to the uncertainty of gossip and user asynchronism. Fig. 9(b) shows the CDF of the network hops of tweets received by all the users. We compare the network hops of *valid* tweets (i.e., tweets received for first time) with those of *redundant* (marked as "Redt" in Fig. 9(b)) tweets, as well as the hops of valid tweets with doubled fanout, i.e., $f = 2T$. We can see that 90% of valid micronews received are within 8 network hops, while redundant tweets use more hops to reach the followers. On the other hand, increasing fanout reduces limited network hops of dissemination. With $f = 2T$, each user only reduces less than one hop on average to receive micronews, while the partner maintenance overhead is doubled. We further study the performance of Cuckoo's client heterogeneity support by tuning the proportion of Cuckoo-Mobile peers among all the nodes. Fig. 9(c) shows the average coverage rate of micronews dissemination with different mobile proportion. According to the figure, Cuckoo achieves stable dissemination performance in terms of coverage rate under client heterogeneity. Even when the mobile proportion reaches 80%, the dissemination can still achieve over 90% coverage. Nevertheless, when the mobile proportion is high, the high dissemination coverage is based on the high overhead of Cuckoo's gossip nodes: each gossip node is likely to maintain and spread micronews to extra $(\frac{\rho}{1-\rho}) \times T$ leaf nodes, where ρ is the proportion of Cuckoo-Mobile peers [34].

Client Overhead. In Cuckoo, the client overhead for message delivery is twofold: outgoing traffic overhead and incoming traffic overhead. The outgoing traffic overhead is bounded according to the delivery/dissemination mechanisms. The overhead of unicast delivery is n-tweet sending per process, where n is the number of online followers. For gossip dissemination, the overhead is f-tweet sending per process where f is the fanout. The incoming traffic overhead is mainly caused by receiving redundant messages. Fig. 9(d) shows the CDF of average redundant messages received by each peer for one dissemination process, compared with that of $f = 2T$. Moreover, we pick out the mobile users (i.e., Cuckoo-Mobile) and show their redundancy. According to the figure, around 89% of users receive

less than 6 redundant tweets for one dissemination process while 80% of mobile users among them receive less than 6, which is of acceptable overhead due to the small size of tweet messages. On the other hand, the increase of fanout causes larger redundancy. For $f = 2T$, more than 65% of users receive more than 6 redundant tweets, while the increase of coverage rate is trivial (less than 1%). Other client overhead includes the overhead for DHT maintenance (only for Cuckoo-Comp peers) and partner maintenance (e.g, announcement, discovery).

6 Conclusion

We have presented Cuckoo, a novel system architecture designed for scalable microblogging services. Based on the observation of divergent traffic demands, Cuckoo decouples the dual components of microblogging services. We use complementary mechanisms for reliable content delivery while offloading processing and bandwidth costs away from a small centralized server base. We have prototyped Cuckoo and evaluated our prototype using trace-driven emulation over 30,000 Twitter users. Compared with the centralized architecture, Cuckoo achieves notable server bandwidth savings, CPU and memory reduction, while guaranteeing reliable message delivery. In short, Cuckoo provides good performance for microblogging both as a social network and as a news media.

Acknowledgements. We thank our shepherd Ranveer Chandra and the anonymous reviewers for their valuable feedback. This work is supported in part by the National Science Foundation under grants IIS-0916307 and IIS-847925.

References

1. AFP.com. 'Twitters' Beat Media in Reporting China Earthquake (2008)
2. Buchegger, S., Schiöberg, D., Vu, L.-H., Datta, A.: PeerSoN: P2P Social Networking – Early Experiences and Insights. In: Proc. of SNS (2009)
3. Castro, M., Costa, M., Rowstron, A.: Debunking Some Myths about Structured and Unstructured Overlays. In: Proc. of NSDI (2005)
4. Cha, M., Haddadi, H., Benevenuto, F., Gummadi, K.P.: Measuring User Influence in Twitter: The Million Follower Fallacy. In: Proc. of ICWSM (2010)
5. CNET News. Twitter Crippled by Denial-of-Service Attack (2009)
6. Cutillo, L.A., Molva, R., Strufe, T.: Safebook: A Privacy Preserving Online Social Network Leveraging on Real-Life Trust. IEEE Communication Magazine 47(12), 94–101 (2009)
7. Eugster, P.T., Guerraoui, R., Kermarrec, A.M., Massoulié, L.: From Epidemics to Distributed Computing. IEEE Computer 37, 60–67 (2004)
8. Examiner.com. San Francisco Twitter Users Shocked to Lose All Their Followers (2010)
9. FastCompany.com. Twitter Predicts Box-Office Sales Better Than a Prediction Market (2010)
10. Frey, D., Guerraoui, R., Kermarrec, A.-M., Koldehofe, B., Mogensen, M., Monod, M., Quéma, V.: Heterogeneous Gossip. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 42–61. Springer, Heidelberg (2009)
11. Ghosh, S., Korlam, G., Ganguly, N.: The Effects of Restrictions on Number of Connections in OSNs: A Case-Study on Twitter. In: Proc. of WOSN (2010)

12. Guardian.com. Twitter Election Predictions Are More Accurate Than YouGov (2010)
13. Gyarmati, L., Trinh, T.A.: Measuring User Behavior in Online Social Networks. IEEE Network Magazine, Special Issue on Online Social Networks 24(5), 26–31 (2010)
14. Java, A., Song, X., Finin, T., Tseng, B.: Why We Twitter: Understanding Microblogging Usage and Communities. In: Proc. of WEBKDD/SNA-KDD (2007)
15. Kermarrec, A.-M., Massoulié, L., Ganesh, A.: Probabilistic Reliable Dissemination in Large-Scale Systems. IEEE Transactions on Parallel and Distributed Systems 14(3), 248–258 (2003)
16. Krishnamurthy, B., Gill, P., Arlitt, M.: A Few Chirps about Twitter. In: Proc. of WOSN (2008)
17. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a Social Network or a News Media? In: Proc. of WWW (2010)
18. Mislove, A., Post, A., Druschel, P., Gummadi, K.P.: Ostra: Leveraging Trust to Thwart Unwanted Communication. In: Proc. of NSDI (2008)
19. Mobilesyrup.com. Twitter CEO: 40 Percent of All Tweets Created on Mobile Devices (2011)
20. Motoyama, M., Meeder, B., Levchenko, K., Voelker, G.M., Savage, S.: Measuring Online Service Availability Using Twitter. In: Proc. of WOSN (2010)
21. New York Times. Sports Fans Break Records on Twitter (2010)
22. Ramasubramanian, V., Peterson, R., Sirer, E.G.: Corona: A High Performance Publish-Subscribe System for the World Wide Web. In: Proc. of NSDI (2006)
23. Reuters News. Twitter Snags over 100 Million Users, Eyes Money-Making (2010)
24. Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In: Liu, H. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–351. Springer, Heidelberg (2001)
25. Pingdom, R.: Twitter Growing Pains Cause Lots of Downtime in 2007 (2007)
26. Sandler, D., Mislove, A., Post, A., Druschel, P.: FeedTree: Sharing Web Micronews with Peer-to-Peer Event Notification. In: van Renesse, R. (ed.) IPTPS 2005. LNCS, vol. 3640, pp. 141–151. Springer, Heidelberg (2005)
27. Sandler, D.R., Wallach, D.S.: Birds of a FETHR: Open, Decentralized Micropublishing. In: Proc. of IPTPS (2009)
28. Shakimov, A., Varshavsky, A., Cox, L.P., Cáceres, R.: Privacy, Cost, and Availability Tradeoffs in Decentralized OSNs. In: Proc. of WOSN (2009)
29. Twitter Blog. Big Goals, Big Game, Big Records (2010)
30. Twitter Blog. What's Happening with Twitter? (2010)
31. Twitter Engineering Blog. A Perfect Storm.....of Whales (2010)
32. Wang, F., Xiong, Y., Liu, J.: mTreebone: A Collaborative Tree-Mesh Overlay Network for Multicast Video Streaming. IEEE Transactions on Parallel and Distributed Systems 21(3), 379–392 (2010)
33. Xu, T., Chen, Y., Fu, X., Hui, P.: Twittering by Cuckoo – Decentralized and Socio-Aware Online Microblogging Services. In: SIGCOMM Demo (2010)
34. Xu, T., Chen, Y., Jiao, L., Zhao, B.Y., Hui, P., Fu, X.: Cuckoo: Scaling Microblogging Services with Divergent Traffic Demands. Technical Report IFI-TB-2011-01, Univ. of Goettingen (2011)
35. Xu, T., Chen, Y., Zhao, J., Fu, X.: Cuckoo: Towards Decentralized, Socio-Aware Online Microblogging Services and Data Measurement. In: Proc. of HotPlanet (2010)
36. Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., Kubiatowicz, J.D.: Tapestry: A Resilient Global-Scale Overlay for Service Deployment. IEEE Journal on Selected Areas in Communications 22(1), 41–53 (2004)

Contrail: Enabling Decentralized Social Networks on Smartphones

Patrick Stuedi[1,⋆], Iqbal Mohomed[2,⋆], Mahesh Balakrishnan[3],
Z. Morley Mao[4], Venugopalan Ramasubramanian[3], Doug Terry[3], and Ted Wobber[3]

[1] IBM Research, Zurich, Switzerland
stu@zurich.ibm.com
[2] IBM Research, T.J. Watson, USA
iqbal@us.ibm.com
[3] Microsoft Research, Silicon Valley, USA
{maheshba,rama,terry,wobber}@microsoft.com
[4] University of Michigan, USA
zmao@umich.edu

Abstract. Mobile devices are increasingly used for social networking applications, where data is shared between devices belonging to different users. Today, such applications are implemented as centralized services, forcing users to trust corporations with their personal data. While decentralized designs for such applications can provide privacy, they are difficult to achieve on current devices due to constraints on connectivity, energy and bandwidth. Contrail is a communication platform that allows decentralized social networks to overcome these challenges. In Contrail, a user installs content filters on her friends' devices that express her interests; she subsequently receives new data generated by her friends that match the filters. Both data and filters are exchanged between devices via cloud-based relays in encrypted form, giving the cloud no visibility into either. In addition to providing privacy, Contrail enables applications that are very efficient in terms of energy and bandwidth.

1 Introduction

The emergence of powerful smartphones and ubiquitous 3G connectivity has led to a number of new mobile applications. Many of these applications are centered on social networking, where users on mobile devices want to selectively consume content generated by their friends' devices. For example, Alice wants to receive pictures taken by her friends in which she is tagged, view status updates by her friends mentioning the movie "The Social Network", and be notified of her child's location if he strays too far from home.

Today, such applications exist in the form of centralized services such as Facebook, FourSquare or Flickr; new content generated by a device is first uploaded to a central server, which then selectively redistributes it to other devices. A centralized version of the child-tracking application would have the child's phone periodically update a central server with his location; the server would then notify Alice if the location is outside bounds specified by her. Centralized solutions are simple and efficient, allowing

⋆ This work was performed while these authors were with Microsoft Research.

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 41–60, 2011.

a device to upload data just once to the cloud in order to share it with multiple recipients, without requiring any of them to be online at the same time.

However, centralized solutions come at the cost of user privacy. Individuals are forced to trust corporations to not misuse their data or sell it to third parties. They must also trust companies to guard their data against malicious hackers or repressive governments. These concerns are amplified by the very personal nature of data generated on mobile devices. In the example of Alice's location-tracking application, the central server knows both the current location of her child as well as the location of Alice's home. While privacy requirements are subjective and vary from person to person, today's technology offers a stark choice: give up privacy or stay offline.

In contrast, decentralized designs can offer better privacy to end-users. Since our focus is on privacy, we use the term 'decentralized' to refer to any system where a user's data can be viewed unencrypted only on trusted devices, and not at any intermediate point in the network. We expect such systems to execute application logic exclusively on edge devices, using encrypted channels between devices to coordinate across them. Decentralized designs for privacy-aware social networks have been explored in the context of wired end-hosts [1, 3].

Unfortunately, implementing decentralized applications on modern smartphones is challenging. At a basic level, getting messages from one device to another can be surprisingly difficult; smartphones and the wireless 3G/4G networks they run on are designed for simple client-server interactions, not inter-device communication. Assuming smartphones can somehow exchange messages, a more complex challenge for decentralized applications relates to minimizing communication, a crucial goal in the context of battery limitations and bandwidth caps.

In this paper, we present Contrail, a communication platform that enables efficient, decentralized social networks on smartphones. At the heart of Contrail is a simple cloud-based messaging layer that enables basic connectivity between smartphones, allowing them to efficiently and securely exchange encrypted data with other devices. Over this messaging layer, Contrail implements a novel form of publish/subscribe that uses *sender-side* content filters to minimize bandwidth and energy usage while preserving privacy. Additionally, Contrail provides mechanisms that are critical for reducing the energy and bandwidth footprint of applications, such as the ability to flag in-flight data as expired or obsolete.

Contrail's content filters allow devices to selectively receive subsets of data produced by other devices. When Alice wants some data from Bob – for example, all photos taken by Bob in Seattle – she attempts to install a content filter on his smartphone expressing her interest. If Bob agrees to install this filter on his device (he can choose to decline the request), all subsequent photos taken by him in Seattle are routed to Alice's phone. Similarly, Alice could install a filter on her child's phone expressing her interest in his location if he leaves a certain bounding area. Content filters support a wide range of social network applications, including location-based services, photo and video sharing, message walls and social games.

Contrail is implemented on the Windows Azure cloud platform and on Windows Mobile 6.5 devices. Our evaluation shows that this implementation offers latency and throughput between edge devices that is limited only by current 3G network speeds.

We have also implemented several social network applications on Contrail, including location-tracking and photo-sharing. This paper makes the following contributions:

- We describe the challenges faced in implementing a decentralized social network on smartphones, and translate these into a set of requirements for a communication platform.
- We describe the design of the Contrail system, which combines the novel idea of sender-side content filters with other techniques to enable efficient social networks on smartphones.
- We present an implementation of Contrail on Windows Azure and Windows Mobile 6.5, and evaluate its performance.

2 Problem Statement

Our primary goal is to enable decentralized social network applications on smartphones. As described, we expect such applications to obtain privacy by placing logic at edge devices and coordinating via encrypted channels. In this section, we elaborate on the challenges such applications face.

We use the child-tracking application as a running example. Consider a simple implementation of this application — once every five minutes, the child's (let's call him Junior) device generates a location update, encrypts it, and sends it to Alice's phone. On Alice's phone, the update is decrypted and then checked against predefined bounds (that correspond to Alice's home, for example). If Junior is out of bounds, an alarm is triggered on Alice's phone. This implementation is decentralized – no central server sees Junior's location or Alice's interests – and consequently offers privacy.

As we mentioned, the first challenge faced in building such an application is basic connectivity: Junior's phone can't easily send messages to Alice's phone. 3G/4G networks do not usually support incoming TCP connections. Even when they do, smartphones are disconnected more often than not; devices can be in low-signal areas, run out of battery, have power-aware radios that sleep intermittently, or simply be turned off. In fact, two devices that wish to communicate with each other may never be online simultaneously. As a result, conventional tunneling solutions used in wired networks do not translate well to this setting.

One option for connectivity is to use existing solutions meant for decoupled communication, such as SMS or e-mail. Junior's phone can send its current location to Alice's phone inside an e-mail. Since SMS and e-mail use centralized servers only as "dumb" message relays, their payloads can be encrypted, offering private communication channels between devices. However, these mechanisms are designed for human-readable content, and can be slow, bulky and inflexible when used as a general message transport.

More fundamentally, transports such as e-mail or SMS offer no support for building efficient social networks on smartphones. To understand this point, we outline a number of key dimensions of efficiency. We also illustrate how the location-tracking application (implemented over e-mail) fails to be efficient on each count.

Download Efficiency: A device should only download data it is interested in. *Alice's phone receives a constant barrage of updates from Junior's phone even when he's at home, draining her battery and using up bandwidth.*

Upload Efficiency: A device should only upload data if some other device is interested in it. *Junior's phone continuously uploads location updates even when he's at home, using up energy and bandwidth.*

Multicast Efficiency: A device should upload data only once for multiple recipients. *Bob wants to know where Junior's phone is, as well. If Junior's phone sends separate messages to Bob and Alice, it now drains even faster and uses up more bandwidth.*

Semantic Efficiency: A device should only download data that is not expired or obsolete. *When Alice turns on her phone after keeping it switched off for a meeting, she receives a flood of location updates from Junior's phone, even though she only cares about his last location.*

Some of these properties (such as upload and download efficiency) can be achieved via extra application logic, while others (such as multicast and semantic efficiency) require explicit hooks from the transport layer. Clearly, the simple decentralized implementation of the location-tracking application that uses e-mail as a transport fails to offer any of these efficiency properties (except multicast efficiency, since a single e-mail can be uploaded once for multiple recipients). In contrast, a purely centralized solution does not provide privacy, but does offer all the efficiency properties (except upload efficiency).

Required is a transport layer that makes it trivial for applications to achieve all four efficiency properties while also providing decoupled connectivity and privacy. In essence, these efficiency properties amount to ensuring that data is only uploaded and downloaded by devices when absolutely necessary. For a transport layer to assist applications in achieving this goal, it has to understand application-level requirements; in other words, the application has to specify to the transport layer which devices require what data.

Why not use existing Pub/Sub implementations? Publish/subscribe interfaces are a natural fit for this problem. In a pub/sub system, the application running on each node *subscribes* to specific data; for example, a server might wish to receive stock quotes of MSFT if it is above $25. Subsequently, data *published* by other nodes — such as updates to the MSFT stock price — is routed selectively to other nodes based on their subscriptions.

Unfortunately, existing pub/sub implementations do not provide the guarantees we need to build decentralized social networks. Pub/sub systems typically *filter* data — i.e., match data to subscriptions — at centralized servers, in which case they do not provide privacy. Alternatively, they filter data at the edge receivers, in which case they cannot provide the upload and download efficiency properties; data must be uploaded by the sender and downloaded by the receiver before it can be determined if the receiver really wants it.

More generally, an important goal of publish/subscribe systems is *anonymous* communication, where senders can transmit data to interested receivers without having to know and enumerate their identities. In contrast, we are interested in secure, private communication between trusted nodes. This leads us to make very different design choices from current pub/sub systems, as will become clear in the following sections.

3 Design of Contrail

Here, we provide a high-level description of Contrail's design. We describe the two main mechanisms in Contrail – sender-side filters and cloud relays – and explain how they provide the properties enumerated in the previous section.

3.1 Sender-Side Filters

The Contrail universe consists of users, the devices belonging to those users, and cloud-based relay servers. In a brand new instance of Contrail, no device sends or receives messages; from this starting point, we progressively describe how communication occurs. Two kinds of messages exist in Contrail — *filter installation requests* and *data messages*. First, we describe when and why these messages are sent between devices; later, we will describe how they are sent.

A Contrail filter is simply an application-defined function that accepts some unit of data as input and returns true or false. Filters are installed by one device (we call this the consumer device) on another device (the producer device). Once a filter is installed on the producer device, it is evaluated by that device on any new data; if it matches, that data is transmitted to the consumer device. Filters are application-defined; for example, they might check if GPS coordinates lie within some area, test photograph tags for equality with some string, or scan status updates for some keyword. For ease of exposition, we assume that there is only one application running on the devices; later, we will describe multiplexing mechanisms.

A device can attempt to install a Contrail filter on some other device by sending a filter installation request. The request only reaches the producer device if it includes the consumer device in a white-list. This is similar to users 'adding' each other on conventional social networks; for example, for Alice's phone to install a filter on Bob's phone, Bob would have to include Alice's phone (or, using a wildcard, any of her phones) on the white-list of his phone (or all of his phones). This allows Alice's device to request filter installations on his device.

The filter installation succeeds only if the producer device accepts the request. On the producer device, incoming filter installation requests are relayed to the application, which decides whether to accept them or not (possibly based on user input). Once a filter is installed, data matching it is allowed to travel back from the producer device to the consumer device.

Contrail's content filters give us privacy, since the filtering of data occurs on trusted edge devices, not central servers. They also give us upload and download efficiency; a device only uploads data matching a filter installed on it by another device. Conversely, it only downloads data matching a filter installed by it on another device.

3.2 Cloud Relays

Now we describe the mechanics of how messages (filter installation requests as well as data messages) travel from one device to another. Contrail consists of a client-side module that executes on each device, and a messaging layer that resides in the cloud. Each

client-side module periodically initiates a TCP connection to the cloud-based messaging layer via 3G (or a WiFi hotspot). In simple terms, a message sent by one device to another is first uploaded to the cloud via one device-to-cloud connection, and subsequently pulled by the recipient device via another such connection. These device-to-cloud interactions are the only network-level connections that occur in the system; for ease of exposition, we assume no out-of-band interactions between devices via channels such as Bluetooth.

Contrail's cloud layer consists of stateless application servers and a persistent storage tier. When devices connect to the cloud, they interact with one of these application servers; we call this the *proxy* for the device. If a device uploads a message meant for an offline recipient, its proxy stores the message in the storage tier. When the recipient device comes online, its proxy checks the storage tier for any messages meant for it and transfers them. On the other hand, if the recipient is online and connected to some other application server, the two proxies interact directly to transfer the message, without the storage tier in the critical path.

As described, the design of Contrail's cloud layer enables decoupled connectivity between devices. To provide multicast efficiency, the cloud layer allows senders to specify multiple recipients on a message. To provide semantic efficiency, it allows senders to set expiry times on messages, and to mark new messages as superseding older in-flight messages. When a message sent to an offline device expires before the device comes online, or is made obsolete by a new message, it is deleted from the cloud's storage tier.

Consequently, Contrail's combination of edge-based content filters and a cloud-based relaying layer allow it to offer all the properties of interest to us. Social network applications built using Contrail are privacy-aware, can work across devices decoupled in space and time, and are naturally efficient in terms of energy and bandwidth.

3.3 Reliability and Security in Contrail

To understand Contrail's reliability and security guarantees, we need to first state our assumptions about the cloud. Our reliability guarantee assumes the cloud does not lie about persistence; data stored in the cloud will not be lost. Our privacy guarantees do not make any assumptions about the cloud. In other words, a malicious cloud can interfere with Contrail's reliability and performance, but cannot view user data. Also, our design can be easily implemented on any existing cloud platform; consequently, if the cloud we use does not offer the desired reliability and performance, we can switch to one that does.

Contrail's cloud layer offers *reliable* communication — all messages are buffered on the sender device until its proxy acknowledges that it has stored the message persistently in the cloud's storage tier. This in-cloud copy of the message is deleted once the receiver device acknowledges receipt to its own proxy. This allows reliable communication between devices that are not simultaneously online. It is also an efficient reliability option when both devices are online, since it allows a fast sender to upload and disconnect once all messages have been persisted, without waiting for the receiver to finish downloading them.

Contrail's cloud layer also offers *secure* communication via a combination of well-known mechanisms. The flow of messages is tightly restricted by the white-lists

described previously; for social network applications, we expect these white-lists to correspond to friend lists, ensuring that messages only travel along the edges of the social graph. White-lists for users are stored in the cloud and proxies only relay filter installation requests between devices as permitted by these. Our assumption is that the cloud will honor these white-lists. As a result, devices cannot be spammed with filters by unknown rogue devices.

Privacy is ensured via device-to-device encryption: the cloud sees only encrypted payloads. Our strategy for encrypted communication is not novel; we use simple off-the-shelf techniques. We use public key encryption to exchange symmetric keys between devices, which are then used for encrypting all messages. For example, if Bob wants to send messages to Alice, he first sends her a message encrypted with her public key, so that only someone with her private key can decrypt it. That message contains a symmetric key which is used for all future messages (since symmetric encryption is faster and uses less energy on a smartphone than public key encryption).

For messages meant for multiple recipients, we encrypt the payload with a freshly generated symmetric key and then include this symmetric key as well in the message, encrypted separately with each recipient's public key. For example, if Alice is sending a photograph to Bob, Charlie and Donald, the outgoing message consists of the photograph encrypted with the new symmetric key, along with three versions of the symmetric key, encrypted with Bob's, Charlie's and Donald's public keys respectively. These per-message symmetric keys are cached and reused if many messages are sent to the same set of people.

In some applications, users may want to authenticate messages, ensuring that they did indeed originate from the apparent sender and were not tampered with. To handle this, Contrail computes a hash of the payload of each message and signs it with the sending user's private key.

Contrail does not provide privacy of inter-device relationships; through the white-lists, the cloud knows which devices (and which users) are talking to each other, even if it does not know what they are talking about. In the context of a social network, this amounts to the cloud knowing who your friends are. We think this is an acceptable trade-off: white-lists enable a spam-free system resistant to denial-of-service attacks (a critical property for resource-constrained devices), but require users to reveal their friend lists to the cloud.

4 The Contrail System

As described, Contrail consists of a client-side module that executes on each device and a messaging layer that runs in the cloud. In this section, we delve into the details of these two components.

4.1 Contrail on the Phone

Identifiers in Contrail: The basic unit of data in Contrail is an *item*. An item is defined as the combination of a payload and application-defined metadata. While metadata can be in any form, the default option in Contrail is to represent it as a hash-table of

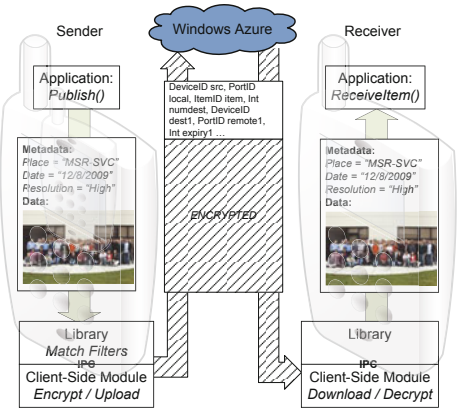


Fig. 1. The path taken by a data item through the Contrail stack

key-value pairs. For example, an item used by a photo-sharing application would store the actual photograph in the payload, and attach metadata pairs to it such as ("date", "9/19/2010") and ("location", "San Francisco, CA"). Each item has an application-specified *ItemID*. The ItemID does not have to be unique across items generated by different applications; applications can set the same ItemID for different items (such as different versions of a document) to indicate that the later one makes the other obsolete.

A Contrail *end-point* is a pair consisting of a *DeviceID* and a *PortID*. The DeviceID is a globally unique identifier similar to a DNS name that is assigned to each client-side module. The PortID is a locally unique identifier used to multiplex traffic across different applications on the same device.

Contrail API: Contrail provides a library for applications running on the mobile device. The library offers to following API:

```
OpenPort(PortID local, Callback cb)
Publish(PortID local, Item itm, ItemID iid)
InstallFilter(PortID local, Filter f, DeviceID dest, PortID remote)
ReceiveItem(PortID local)
```

To use Contrail, an application creates an end-point by calling the $OpenPort$ function, specifying a PortID and a filter installation callback function. Once the application opens a port, other end-points – i.e., other instances of the application on different devices with open ports – can try to install filters on it, in order to receive data from it. These filters are delivered to the application via the filter installation callback. When a filter is received by the application, the application can either accept or reject it, by returning $true$ or $false$ from the callback, respectively.

To actually send data to other end-points, the application calls the $Publish$ function with an item as a parameter; see Figure 1. This results in all the installed filters on that port being evaluated on the item. The evaluation of the filters is performed by the Contrail library, within the application's own process. If the item is matched by one or more filters, it is transferred by the library to the shared module via IPC, along with a

list of destinations corresponding to the end-points that installed the matching filters. The shared module in turn constructs a data message with the item as the payload and uploads it to the cloud.

The basic format of a data message is shown in Figure 1. The header of the data message includes the source end-point information, the ItemID of the encapsulated item, the number of destination end-points, and routing information for each destination. The routing information for each destination consists of the (DeviceID, ItemID) pair as well as the expiry time of the item for that destination. Expiry times are destination-specific since we believe their utility to be driven by receivers that don't wish to receive stale data.

To install filters on other end-points, the application uses the $InstallFilter$ function. Once it has installed filters, the application can receive messages by calling the $ReceiveMessage$ function, which blocks for incoming items. The Contrail library also supports asynchronous interfaces for receiving messages; we omit these for brevity.

Push vs Pull: In addition to these interfaces, Contrail allows applications to tune the behavior of the shared module. For many applications, the shared module can simply keep a connection constantly open to the cloud; this is how push notifications work for the iPhone e-mail client, for example. For others, keeping a connection open constantly can be wasteful. If the application receives data at fixed, long intervals (a message every hour, for instance), or does not care about minimizing end-to-end latency, it may prefer the shared module to connect and disconnect periodically.

To support such applications, Contrail exposes two parameters. The *polling-interval* parameter, expressed in milliseconds, allows the application to regulate the frequency with which the shared module polls the cloud for new messages. The *idle-timeout* parameter specifies how long a connection is allowed to remain idle before it is torn down. Creating connections more frequently and keeping them open longer results in lower latencies for message delivery at the cost of energy and bandwidth. Since the shared module is shared by multiple applications, it chooses the lowest *polling-interval* and longest *idle-timeout* requested across all applications.

4.2 Contrail in the Cloud

The Contrail messaging layer is designed to run on any generic cloud provider; this flexibility allows for applications to switch between cloud providers when faced with faults and security issues. The only assumption Contrail makes about the cloud infrastructure it runs in, is that it provides an object store accessible through a put/get interface. Today most cloud providers (e.g., Microsoft Azure, Google AppEngine, Amazon AWS) do provide such a service.

Connecting with the Cloud: When a Contrail device connects to the cloud, it is directed to a randomly chosen application server (in Azure, these are called *worker roles*). We call this application server the proxy for that device during that connection. If this is the first time that the device has connected to the cloud, the proxy creates a message queue for the device in the storage tier. The name of this queue is simply the DeviceID of the connecting device. The purpose of the queue is to hold incoming data items and filters sent to the device from other Contrail end-points.

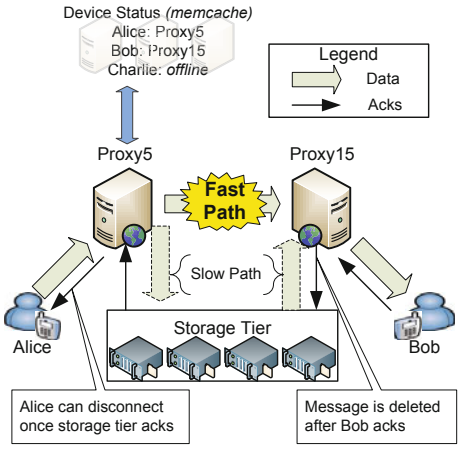


Fig. 2. Contrail implementation: data travels between proxies on a fast path for online devices and a slow path for reliability and offline devices

Upon accepting the connection from the device, the proxy updates a central map with the status of the device. This map has an entry for each device, including whether it's currently online or offline, along with its current proxy if it's online. The map is stored in an in-memory storage service such as memcached; since Azure does not currently have such a service, we implemented our own over standard worker roles.

Relaying messages: If the connecting device has a message to send to another device, the proxy first checks the device map. If the receiver device is online and connected to the cloud, the proxy of the sending device opens a connection to the proxy of the target device and transfers over the message (we call this the *fast path*). The destination proxy then relays the message to the target device.

In parallel, it also writes the message to the queue of the target device in the storage tier (the *slow path*). This happens whether the target device is offline or online. When the target device is offline, writing it persistently allows the device to retrieve it at a later time; when it is online, it ensures that the message will be reliably delivered without requiring the sending device to stay online. Once the message is persisted in the storage tier, the proxy sends back an acknowledgment to the sending device. This lets the sending device delete the message from its buffers and go offline if required, with the guarantee that the message will be eventually delivered to the recipient.

Delivering messages: To receive messages from other devices via the fast path, the proxy listens for connections from other proxies. When the device first connects, the proxy also checks for incoming messages in the storage tier sent via the slow path while the device was offline. When a device successfully downloads a message, it sends back an acknowledgment to its proxy that triggers the deletion of the message from the storage tier. This ensures that messages are not stored forever in the storage tier.

Alice

```
PortID localP = OpenPort("any_port", null);
SetPollingInterval(localP, 30);
SetIdleTimeout(localP, 0);
/* App-defined function to a filter
   matching locations within Mountain View */
Filter filter = create_mtnview_filter();
/* Install filter on remote port with
   id equals "location_update" */
InstallFilter(localP,filter,
              remotedevice,"location_port");
/* Alice receives location updates from
child's phone if he leaves Mountain View */
Item msg = ReceiveItem(localP);
if(msg!=null)
/*child has left Mountain View!*/
    freak_out();
```

Alice's Child

```
PortID localP = OpenPort("location_port", null);
while(true)
{
/* Alice's phone determines her location using GPS */
Location current_location = get_current_location();
Item msg = new Item();
AddMetadataToItem(msg, "location",current_location);
/* Publishing with same ItemID "mycurlocation"
   every time makes previous location
   updates obsolete */
Publish(localP, msg, "mycurlocation");
sleep(1 minute);
}
```

Fig. 3. Code for child-tracking application using the Contrail API

Contrail ensures reliable delivery once the sender receives an acknowledgment, assuming that the cloud's storage tier does not suffer data loss and that the receiving device eventually connects to the cloud. The message is not removed from the sender's buffer until it is persisted on the cloud's storage tier, as indicated by the acknowledgment to the sender. It is not removed from the storage tier until it has been acknowledged by the receiver. Failures of the sender and receiver proxies or disconnections of the devices from the cloud can result in duplicate uploads and downloads of messages, but not loss.

5 Applications

Contrail makes it easy for developers to build social network applications that are decentralized yet efficient. We built several applications using Contrail, including location-tracking, photo-sharing, folder-sharing and chat. In this section, we first describe the design of the location-tracking application, and then elaborate on other possible applications.

5.1 The Location Notification Application

Here, we describe the details of the location notification application. The goal of this application is to notify users when the location of their friends satisfies some fixed condition; for example, as mentioned previously, a user Alice may want to know if her child is outside a threshold distance from his school, or if a friend she planned to meet at the mall has reached there. We will describe how Contrail allows such an application to be built in a manner that conserves bandwidth and power without sacrificing privacy, using filters as well as functionality such as item obsolescence and expiry times.

Figure 3 shows the pseudo-code for the location notification application. At a high level, this application uses filters in the following manner: Alice's device installs a filter on her child's device that includes the condition to be checked. The application running on her child's device periodically publishes his location as an item. Contrail on the child's device checks the installed filter on the location item, and pushes the item to the

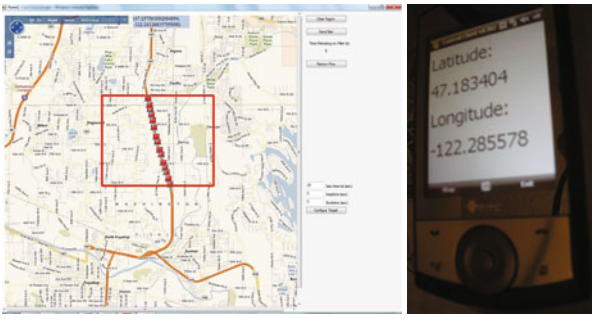


Fig. 4. Contrail application for selective location sharing

cloud if it matches. Importantly, each matching location update is published using the same ItemID ("mycurrentlocation" in the figure), making previous updates obsolete; as a result, if Alice's device connects to the cloud after a prolonged disconnection, she receives only the latest location update.

In the pseudo-code, we omit the details of the filter. In our example, the filter is a bounds check on the location item's latitude and longitude. We represent the Mountain View area as a box with four corners, each of which has a latitude and longitude. Our filter is a conjunction of comparisons between the current coordinates and the bounds of the box. While our current implementation is restricted to such filters, Contrail can easily support more complex queries; for example, we could compute the distance of the current coordinates from a fixed point and check it against a threshold.

This application can also be used to notify users of their friends' location within a specific area. For example, Alice may want to know Bob's location, but he may choose to reveal it to her only when he's within the Microsoft campus. Figure 6 shows our location-tracking application in such a scenario. Alice installs a filter on Bob's phone asking for his location within a specific part of Seattle, which he accepts. On the right is Bob's phone generating location updates, and on the left is a computer where Alice is tracking Bob's location. As can be seen, Alice views Bob's location only when he is within the bounds specified.

5.2 Potential Contrail Applications

Real-Time Interactive: Applications such as chat, collaborative document editing, audio/video-conferencing and real-time games can be built easily using Contrail. Currently, such applications use either centralized servers (e.g., Google Wave) or – as in the case of Skype – leverage application-specific peer-to-peer networks on the wired Internet to tunnel traffic from and to 3G devices. To set up a chat session involving two or more people, for example, the application would simply have each participating device install filters on the other devices.

In addition to the obvious benefit of privacy, real-time applications benefit from Contrail's upload and multicast efficiency — a web-cam could stop uploading if nobody is watching it, or upload a stream just once for multiple viewers. Contrail's semantic

efficiency properties are also useful to such applications; they can set expiry times on outgoing items, ensuring that receivers do not get stale video frames, for example. Similarly, they can set up obsolescence relationships, ensuring that the receiver only receives the latest video frame or the latest version of a document.

Content Sharing: Contrail is useful for sharing bulk data items such as photographs or videos. Simple sharing is trivial to implement in Contrail; users can accept filters from their friends to enable sharing and then tag new media with the appropriate metadata. An application that wants to let users search their social network for existing content – as opposed to continuously receive new content – would simply use temporary filters with very short lifetimes and re-publish existing content through these filters. Interestingly, each query can also be propagated along the social graph at the application-level if recipients of the filter install it on their own friends, thus implementing P2P search on the social graph. Contrail's main benefit for content sharing applications is privacy, since the content metadata is not exposed to third parties.

Sensor Aggregation: Mobile devices can be viewed as sensors from which data can be aggregated, processed and queried (for example, phones being used to track traffic). Contrail is a great fit for sensor aggregation applications, since filters can be used to construct arbitrary aggregation topologies that save bandwidth and enforce privacy. For example, all Microsoft employees at the Silicon Valley campus could transmit their GPS locations to a local Microsoft server they trust, which then knows their individual locations; in turn, this server could transmit anonymized or aggregated data to a public server. This example would require the local Microsoft server to install filters on employee devices, and the public server to install a filter on the Microsoft server. As such, this example shows that a Contrail instance can include trusted machines in addition to edge devices.

Can Facebook be built using Contrail?

An interesting question for Contrail is whether it can support the same kinds of applications currently found on centralized services such as Facebook. We believe that most of these applications are easy to build on Contrail. For instance, message walls are simple to implement — Alice can install a catch-all filter on Bob's device that is evaluated on all new status updates. Facebook-style commentary threads for individual status updates seem difficult to achieve at first glance, since users can view comments made by each other on a common friend's wall even if they aren't each other's friends; for example, if Alice comments on Bob's status update, all of Bob's friends can view her comment.

In Contrail, communication between non-friends can be achieved by having users republish information at the level of the application. For example, to allow all of Bob's friends to view Alice's comment on his status update, consider a scheme where each user installs two filters on their friends: one to get status updates, and another to get comments. Now, Alice gets Bob's status update (along with all his other friends) via the status update filter; she then publishes a comment that only Bob gets via the comments filter. Bob then publishes the comment as a status update to his wall so that everybody else gets it.

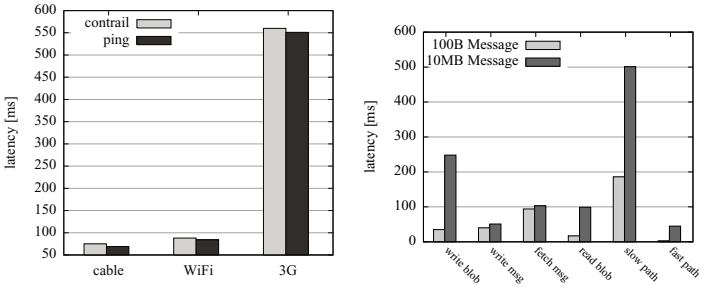


Fig. 5. a) Contrail's end-to-end latency between devices is close to network latency. b) Contrail's overhead in the cloud on the fast path (right-most bars) and the slow path (5 left-most bars).

6 Evaluation

We have evaluated Contrail using our prototype implementation. All our experiments are on a real implementation of Contrail running on Windows Azure. For clients, we use Windows Mobile phones connected to 3G networks, laptops tethered to these phones, and (for scaling experiments) instances in the Amazon EC2 cloud.

The first part of our evaluation focuses on the Contrail cloud-based messaging layer. We show that it provides good performance in terms of end-to-end latency and throughput. We also show that it is highly scalable. The second part of our evaluation focuses on the edge device; we show that Contrail's sender-side filters do not have a high computational overhead. We also evaluate the impact on the edge device of Contrail's tunable parameters.

6.1 End-to-End Latency

Figure 5a shows the end-to-end latency for an item to travel from one laptop to another via Contrail over different networks: when directly attached to a home cable network, when accessing that cable network over WiFi, and when tethered to a 3G phone. Both laptops are in the same physical location and the size of the message is 400 bytes. To understand what fraction of the observed latency was Contrail overhead, we also measured network-level ping latency from one of the devices to a ping server located near the Azure data center hosting the Contrail instance. The resulting graph shows that Contrail's end-to-end latency is limited almost entirely by latency on the network. Contrail itself adds no more than 5 to 10 ms of latency overhead.

Where is this extra latency used up? To find out, we instrumented the path of a Contrail message through the cloud using the Azure Diagnostics tracing framework. In Figure 5b, we show the measurement results for two different message payload sizes, of 100B and 10MB respectively. All the numbers shown are averages taken from 10 samples; we found the differences between each sample to be very small.

To understand Figure 5b, recall that messages in the Contrail cloud follow two separate paths: a fast path via a direct TCP connection between proxies when the communicating devices are both online, and a slow path that involves persisting the message to

disk. The right-most bar in Figure 5b shows the latency on the fast path. This number is crucial; it determines Contrail's latency overhead between two online devices. As can be observed, the latency overhead of a message on the fast path lies slightly below 50ms for a 10MB packet, and is around 4ms for a 100B message; this corresponds to the overhead observed in the previous end-to-end latency graph (Figure 5a).

The four left-most bars in Figure 5a show latency on the slow path. The 'write blob' stage refers to the time it takes the sender proxy to persist a message to the cloud's storage tier (in this case, Azure Blob Storage). The 'message write' stage refers to the time taken to update the queue of the offline recipient with a pointer to the message in the blob store.

6.2 Contrail Scalability

Next, we show that Contrail can scale to large numbers of client devices simply by adding more application servers (or Azure worker role instances) in the cloud. An important value proposition for cloud computing is the notion of elasticity. As load increases, additional computing resources can be harnessed to prevent degradation in the user experience. In the case of Azure, the unit of scaling is an instance, which corresponds roughly to a single virtual machine. We conducted an experiment where we varied the number of clients that were simultaneously connected to the cloud. The experiment was performed under three conditions: where message traffic was being handled by 1, 2 and 10 Azure instances. In this experiment, the clients ran on Amazon EC2 machines (in their US-West Coast facility). We used 100 small EC2 instances and ran 10 clients per instance, after verifying that running 10 clients per machine would not saturate the resources of one instance. Each EC2 client sent a message via Contrail – running in the Azure cloud – to itself every second. Figure 6a shows the average end-to-end message latency across users. We see that while a single instance can easily handle up to 200 simultaneous clients (average round-trip message latency of under 80ms), supporting 300 clients at the same time results in degraded performance (an average message latency of over 200 seconds). However, with 2 Azure instances, we can support up to 400 simultaneous clients (77ms for 300 clients and 87ms for 400 clients). With 500 clients, we start to notice performance degradation (over 200ms), while 600 simultaneous clients result in very high message latency. Finally, we observed that with 10 Azure instances, we were able to support at least 1000 simultaneous clients (78ms). These results indicate that the elastic nature of the cloud provides a scalable routing fabric for Contrail applications. Contrail is a trivially partitionable cloud application: as additional clients use Contrail, performance can be maintained by increasing the number of cloud instances.

6.3 Contrail Throughput

Apart from end-to-end latency on small items, we are also interested in knowing the data rate at which two Contrail clients can communicate. In this experiment we measured throughput of two different scenarios. *Online throughput* is the data rate at which two devices can communicate if both devices are connected to the cloud simultaneously. *Offline throughput* is the data rate at which a device can receive data waiting for it in

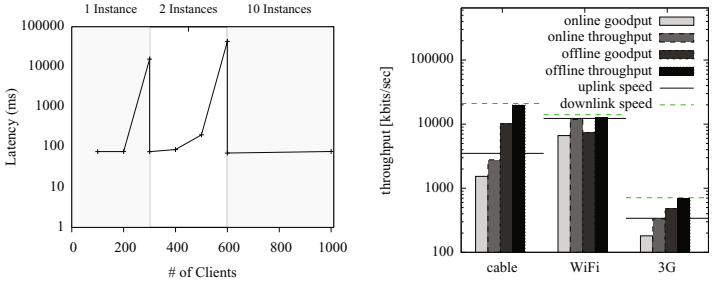


Fig. 6. a) Contrail can scale to thousands of clients simply by adding more server instances in the cloud. b) Throughput and Goodput between two Contrail devices.

the Contrail cloud's persistent storage; this is data sent to the cloud while the receiver device was offline.

Figure 6b shows both online and offline throughput for the case where two laptops are attached to a) a cable network, b) a WiFi network, and c) a 3G network. The meanings of throughput and goodput in the figure are standard: one measures the total bytes transferred per second and includes the overhead of Contrail's headers and serialization mechanisms, while the other measures only the payload bytes transferred per second.

We can see in Figure 6b that Contrail's raw throughput reaches the network limit for all three network types. For online throughput, we are limited by the sender's uplink bandwidth, since the sending device is actively transferring data even as the receiver consumes it. For offline throughput, we are limited by the receiver's downlink bandwidth, since the cloud is able to send data at a fast enough rate.

The figure also shows that Contrail's goodput is much lower than its throughput. This is a limitation of our current implementation, which uses XML serialization of data messages (mainly because it is the only serialization mode natively supported on the Windows Mobile SDK). In the future, we expect to implement custom binary serialization to reduce the gap between goodput and throughput.

In Figure 7a, we evaluate the performance impact of item granularity. The Contrail implementation does not fragment items across multiple messages; each item is sent in a single Contrail message. As a result, applications must decide at what granularity to use items; for example, an application sharing a collection of photos could bundle them all into a single item, or send each photo individually as a separate item.

Accordingly, Figure 7a shows the transfer time of a) a 10MB file when both Contraildevices are attached to a cable network, b) a 10MB file if both sender and receiver are connected to a WiFi network, and c) a 100KB file for the case where both devices are using a 3G network. For all three configurations, smaller items result in lower transfer times up to a point; this is because the messaging infrastructure of Contrail behaves like a store-and-forward network, reading a message to completion before forwarding it to the receiver device. Consequently, the smaller the items, the faster the receiver device starts downloading useful data. Beyond a point, however, smaller items give worse performance, since each message comes with its own headers.

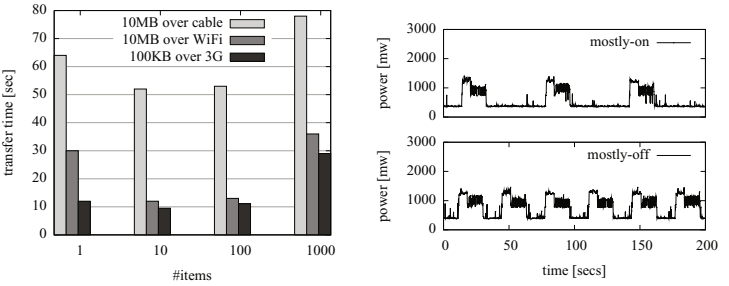


Fig. 7. a) Item granularity: smaller items result in better performance up to a point. b) Contrail uses less power when the connection is kept mostly on (top) as opposed to mostly off.

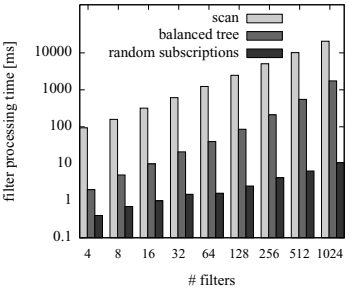
6.4 Energy Consumption and Filtering

In the next set of experiments we study the effects of different options for a Contrail client to communicate with the cloud. As explained in Section 4, the Contrail API lets the application choose proper values for *polling-interval* (pi) and *idle-timeout* (it). Together, these parameters control how frequently the device opens a connection to the cloud and how long it keeps this connection open. Our initial hypothesis was that a longer value of *idle-timeout* would result in higher battery usage but lower message latencies, since the device would stay connected to the cloud for longer periods of time. We tested this hypothesis using a mobile phone running Windows Mobile 6.1. We intercepted the main power cycle between the battery and the phone and measured the instant power consumption using a dedicated power monitor [2].

Figure 7b shows power consumption of two different configurations, one where the polling interval is zero but the idle-timeout is 60 seconds (corresponding to tearing down and re-opening a connection immediately, once a minute), and another one where the polling interval is 30 seconds and the idle-timeout is 0 (establishing a connection every half-minute and tearing it down immediately). Essentially, the first case corresponds to having the connection open almost constantly (mostly-on), while the second case corresponds to creating short-lived connections periodically (mostly-off). The y-axis of the figure corresponds to the instant power consumption and the x-axis refers to time the experiment is running. We are not sending or receiving any data in this experiment.

The figure shows that for both configurations the mobile phone manages to enter a low power state: in the mostly-on case, this state occurs while the connection is on, whereas in the mostly-off case it occurs when the connection is off. This indicates that keeping a connection open does not come with a significant energy penalty. Also, keeping the connection open allows the phone to receive Contrail messages immediately, as opposed to the mostly-off case where it has to wait for a connection to be opened. This result suggests that – at least on this particular hardware – keeping a connection open is always the better strategy.

Despite this result, Contrail still supports the option to configure *idle-timeout* and *polling interval*. Our rationale is that different mobile devices may show different characteristics when it comes to energy consumption. In addition, certain applications may expect messages only at fixed intervals – for example, if a user is receiving updates



Data Rate	Battery Lifetime
0 msgs/minute	6.49 hours
1 msg/minute	5.12 hours
60 msgs/minute	3.95 hours

Fig. 8. a) Filter execution time on a contrail mobile phone. b) Filtering data reduces messages and extends battery lifetime.

from a 3G-enabled temperature sensor – or may prefer to only download the latest version of some data instead of all intermediate versions.

Next, we evaluate the feasibility of Contrail's sender-side filters. Evaluating filters on edge devices may seem infeasible when we consider that it is not uncommon for users on a social network website to have hundreds of friends (which might translate to an equivalent number of installed filters for each application). In this experiment, we study how fast Contrail can match all these filters when a new data item is generated on the mobile phone. We use a specific type of filter in our experiments: conjunctions of equality checks.

The matching time depends heavily on the matching algorithm and the actual set of filters that need to be matched. We study three cases. In the first case, we keep the filters in a list and iterate through the list every time a new item is generated. As can be observed from Figure 8a (label 'scan'), this approach very quickly results in a matching time of several seconds if the number of filters is large. In a second case we implemented a well known matching algorithm that uses a tree data structure to store the filters [4]. We generated filters in the worst possible manner which would cause the algorithm to visit every node in the tree while matching a data item. From Figure 8a (label 'balanced tree') it can be seen that the tree-based matching algorithm reduces the average matching time to a value below one second for 512 filters. In a third case, we used the same matching algorithm, but this time with randomly generated filters. The matching time in this case is just a few milliseconds, even for 1000 filters. This is because the algorithm mostly only traverses one path from the root of the tree to a leaf, where a leaf stores all the filters matching a particular data item.

Lastly, Table 8b present some measurements to show the energy consumption on a Contrail device at different data rates. Clearly, reducing messages improves battery lifetime by a large amount. Thus, Contrail's filtering mechanisms can help applications minimize their battery consumption.

7 Related Work

Content-based Publish/Subscribe [8] is a well-known paradigm that uses content filters to route messages from publishers to subscribers. Contrail filters are similar to those

used by Pub/Sub systems and offer similar benefits, such as decoupled transmission and bandwidth efficiency. However, Contrail uses filters for one-to-one and one-to-many communication between trusted, known devices. In contrast, Pub/Sub is aimed at scaling communication between anonymous sets of publishers and subscribers who do not know each other directly. Many of the results from the Pub/Sub literature on efficient filter matching apply to Contrail as well. Content filters are also to be found in replication frameworks [13].

Prior work by Ford et al. [9] has investigated naming and interconnection schemes for personal mobile devices. Haggle [18] is a network architecture for mobile devices that includes addressing and routing. MobiClique [11] explores opportunistic communication between devices on a social graph. All these projects are focused on settings where devices do not necessarily have ubiquitous 3G connectivity; as a result, many of the design decisions involve cooperation between proximal devices.

Contrail is an example of an Off-By-Default [5,19] network architecture; devices have to install filters on each other to enable communication.

The design of the Contrail client-side module is related to work on efficient polling strategies for phones [10]. Contrail can also leverage hierarchical power management techniques [17,15]. In addition, Contrail can be easily enhanced to support upload and download priorities for data [12]; for example, if a user wants to prioritize her tweets over her video uploads.

Privacy-aware architectures for mobile devices typically rely on trusted delegate machines for computing [14,7]. Contrail is complementary to such techniques; it provides a networking layer that can be used to interconnect devices and delegates.

Privacy-preserving computing techniques already enable specific functionality such as keyword search [6,16]. Contrail is complementary to these solutions; it is possible that applications will push simple functionality into the cloud using privacy-preserving techniques while retaining more general functionality on edge devices in the form of Contrail.

8 Conclusion

Building decentralized, privacy-aware social networks on smartphones is a daunting task; devices are often disconnected and have tight budgets for energy and bandwidth. Contrail is a communication platform that makes it easy for developers to build decentralized social network applications. Contrail enables efficient, privacy-aware applications that trigger communication between devices only when strictly necessary. It achieves this via two mechanisms: sender-side filters that reside on edge devices and cloud-based relays that provide reliable, secure communication between devices.

References

1. Diaspora, http://www.joindiaspora.com
2. Monsoon power monitor, https://www.msoon.com/LabEquipment/PowerMonitor
3. Privacy-aware and highly-available osn profiles. In: 6th International Workshop on Collaborative Peer-to-Peer Systems (COPS 2010) (2010)

4. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., Chandra, T.D.: Matching events in a content-based subscription system. In: PODC 1999: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing. ACM, New York (1999)
5. Ballani, H., Chawathe, Y., Ratnasamy, S., Roscoe, T., Shenker, S.: Off by default. In: Proc. 4th ACM Workshop on Hot Topics in Networks (Hotnets-IV), Citeseer (2005)
6. Boneh, D., Di Crescenzo, G., Ostrovsky, R., Persiano, G.: Public Key Encryption with Keyword Search. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 506–522. Springer, Heidelberg (2004)
7. Cáceres, R., Cox, L., Lim, H., Shakimov, A., Varshavsky, A.: Virtual individual servers as privacy-preserving proxies for mobile devices. In: Proceedings of the 1st ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds, pp. 37–42. ACM (2009)
8. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
9. Ford, B., Strauss, J., Lesniewski, C., Rhea, S., Kaashoek, F., Morris, R.: Persistent Personal Names for Globally Connected Mobile Devices
10. Li, D., Anand, M.: Majab: improving resource management for web-based applications on mobile devices. In: MobiSys 2009: Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services, pp. 95–108. ACM, New York (2009)
11. Pietiläinen, A.-K., Oliver, E., LeBrun, J., Varghese, G., Diot, C.: Mobiclique: middleware for mobile social networking. In: WOSN 2009: Proceedings of the 2nd ACM Workshop on Online Social Networks, pp. 49–54. ACM, New York (2009)
12. Qureshi, A., Guttag, J.V.: Horde: separating network striping policy from mechanism. In: MobiSys, pp. 121–134 (2005)
13. Ramasubramanian, V., Rodeheffer, T.L., Terry, D.B., Walraed-Sullivan, M., Wobber, T., Marshall, C.C., Vahdat, A.: Cimbiosys: a platform for content-based partial replication. In: NSDI 2009: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, pp. 261–276. USENIX Association, Berkeley (2009)
14. Sadeh, N., Hong, J., Cranor, L., Fette, I., Kelley, P., Prabaker, M., Rao, J.: Understanding and capturing people privacy policies in a mobile social networking application. Personal and Ubiquitous Computing 13(6), 401–412 (2009)
15. Shih, E., Bahl, P., Sinclair, M.: Wake on wireless: An event driven energy saving strategy for battery operated devices. In: Proceedings of the 8th Annual International Conference on Mobile Computing and Networking, pp. 160–171. ACM, New York (2002)
16. Song, D., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of the 2000 IEEE Symposium on Security and Privacy, S&P 2000 (2000)
17. Sorber, J., Banerjee, N., Corner, M., Rollins, S.: Turducken: Hierarchical power management for mobile devices. In: Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services. ACM, New York (2005)
18. Su, J., Scott, J., Hui, P., Crowcroft, J., De Lara, E., Diot, C., Goel, A., Lim, M., Upton, E.: Haggle: Seamless Networking for Mobile Applications. In: Krumm, J., Abowd, G.D., Seneviratne, A., Strang, T. (eds.) UbiComp 2007. LNCS, vol. 4717, pp. 391–408. Springer, Heidelberg (2007)
19. Zhang, H., DeCleene, B., Kurose, J., Towsley, D.: Bootstrapping Deny-By-Default Access Control For Mobile Ad-Hoc Networks. In: IEEE Military Communications Conference (MILCOM) 2008, San Diego, November 17-19 (2008)

Confidant: Protecting OSN Data without Locking It Up

Dongtao Liu[1], Amre Shakimov[1], Ramón Cáceres[2],
Alexander Varshavsky[2], and Landon P. Cox[1]

[1] Duke University
[2] AT&T Labs

Abstract. Online social networks (OSNs) are immensely popular, but participants are increasingly uneasy with centralized services' handling of user data. Decentralized OSNs offer the potential to address user's anxiety while also enhancing the features and scalability offered by existing, centralized services. In this paper, we present Confidant, a decentralized OSN designed to support a scalable application framework for OSN data without compromising users' privacy. Confidant replicates a user's data on servers controlled by her friends. Because data is stored on trusted servers, Confidant allows application code to run directly on these storage servers. To manage access-control policies under weakly-consistent replication, Confidant eliminates write conflicts through a lightweight cloud-based state manager and through a simple mechanism for updating the bindings between access policies and replicated data.

Keywords: Decentralization, Onlie Social Networks, Peer-to-peer, Cloud.

1 Introduction

Online social networks (OSNs) such as Facebook, MySpace, and Twitter have enhanced the lives of millions of users worldwide. Facebook has surpassed 700 million active users per month and already attracts 32% of global Internet users every day, with the average user spending 32 minutes each day on the site [8, 9]. The aggregate volume of personal data shared through Facebook is staggering: across all users, Facebook receives nearly 30 billion new items each month. Such high levels of participation should not be surprising: OSNs are fun, useful, and free.

OSN users also trust providers to manage their data responsibly, mining it internally for targeted advertising, and otherwise enforcing user-specified access policies to protect their profiles, messages, and photos from unwanted viewing. Unfortunately, behavior by OSN providers has not met these expectations. In late 2009, Facebook unilaterally eliminated existing restrictions on users' friend lists and other information by making them world-readable. In addition, the site's privacy "transition tool" actively encouraged users to replace restrictions limiting access to "Networks and Friends" with the more permissive "Everyone" option. Similarly, Google revealed many users' most-emailed Gmail contacts by making their Buzz friend list world-readable. Both services eventually reinstated previous access policies after heavy criticism, but many users' sensitive information was exposed for days or weeks.

With OSNs now central to many people's lives, it is critical to address the rising tension between the value of participation and the uncertain privacy guarantees provided

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 61–80, 2011.

by existing services. Users want to continue enjoying OSNs, but they also want to retain control of their data and limit the trust they place in large service providers. *Decentralized OSNs* offer the hope that these goals can be met while potentially improving the functionality and scalability offered by today's successful centralized services.

Several decentralized OSN architectures have been proposed [1, 16, 22] that assume the servers where OSN data is stored are untrusted: data is encrypted before it is stored and plaintext is only viewable by client machines with the appropriate decryption keys. The appeal of this approach is that encrypted data can be stored anywhere, including peer-to-peer DHTs, cloud services such Amazon S3, or even existing OSNs such as Facebook.

However, OSNs have also evolved into large-scale platforms for third-party applications, and we observe that decentralized OSNs that rely on untrusted storage fundamentally limit the kinds of applications that can be built on top of an OSN: if storage servers cannot be trusted with plaintext data, application code that accesses OSN data can execute only on trusted clients. In the worst case, application code must download all relevant data to a client machine, and then decrypt and operate on the data locally. For mobile and desktop clients alike, this can lead to bandwidth, storage, and compute scalability problems.

A partial solution is for OSN designers to anticipate in advance how applications might want to use users' data and require clients to maintain an index of pre-defined features such as key words over the encrypted data [6, 7, 16, 20]. Unfortunately, these techniques limit applications to searching over those pre-defined features, and cannot scalably support richer interactions with the data such as trend spotting or image-based search.

The central question of this paper is: how can decentralized OSNs support a scalable, general-purpose application framework? To answer this question we present the design and implementation of *Confidant*. Confidant's approach to decentralized OSNs is to use information from the social graph to store users' data in plaintext on machines that they trust. The intuition behind our approach is that since a user's friends' machines already have read access to her OSN data, why not allow them to serve her data as well?

Confidant participants use commodity machines such as personal desktop PCs or enterprise workstations as storage servers, and create replicas on machines controlled by a small subset of their friends (e.g., ten friends). Trust relationships between users can be exposed to Confidant in many ways, such as by mining the social graph of existing OSNs (e.g., the level of interaction between users or the number of overlapping friends), or by asking users to manually identify friends to host their data. Once a user has selected her replicas, an application running on behalf of a Confidant user may be authorized to remotely execute sandboxed scripts on these trusted machines.

It is important to note that serving OSN data from personal machines does not introduce new risks to data confidentiality. Centralized and decentralized OSNs alike must store credentials on personal machines (e.g., web cookies, OAuth tokens, or cryptographic keys), and if an attacker compromises a personal machine it will have access to any of the data authorized by the credentials stored on that machine. At the same time, as with other decentralized OSNs, users can feel safe knowing that any data entrusted to Confidant will not be leaked by a centralized OSN behind their back.

The main challenge for Confidant is managing OSN data and access-control policies under weakly-consistent replication. As in prior work [17, 27], Confidant relies on eventual consistency among replicas and treats objects and access policies as immutable first-class data items. However, managing data shared within an OSN presents a different set of challenges than those addressed previously. First, OSN users commonly create new OSN data from multiple clients and should not have to deal with the inconvenience of manually resolving conflicts. Confidant eliminates conflicts without compromising data confidentiality or integrity by serializing a user's updates through a highly available and lightweight state manager hosted in the cloud. In addition, OSN users who inadvertently mis-share an item must be allowed to recover from their mistake without perturbing the policies protecting other data items. Confidant addresses this issue through flexible rebinding of access policies to data items.

To summarize, this paper makes the following contributions. Confidant's design represents the first decentralized OSN architecture to leverage trustworthy storage servers based on inter-personal relationships. This design choice allows Confidant to support a scalable, general-purpose application framework without relying on a centralized OSN to manage users' data. Confidant also provides an access-control scheme for weakly-consistent, replicated data that is tailored to the needs of OSN users. In particular, Confidant eliminates write conflicts and allows participants to recover from access-control mistakes by binding access policies to individual data items rather than to groups of items.

Finally, we have evaluated Confidant using trace-driven simulations and experiments with a prototype. Our simulation results show that typical OSN users should expect read and write success rates of between 99 and 100%. Experiments with our prototype show that applications such as remote keyword search, trending topics, and face detection will scale well; Confidant scripts for processing status updates and photos from 100 friends are between 3 and 30 times faster than an approach relying on untrusted remote storage.

The rest of this paper is organized as follows: Section 2 gives a high-level overview of the Confidant architecture, Section 3 describes Confidant's design, Section 4 describes our prototype implementation, Section 5 presents an evaluation of our design and prototype implementation, Section 6 describes related work, and Section 7 provides our conclusions.

2 Overview

This Section provides a high-level overview of the Confidant architecture and trust model.

2.1 Architecture

Figure 1 shows the Confidant architecture, including paths for posting new data (steps 1-2), retrieving new data (steps 3-5), and running Confidant applications (steps 6-8). Keeping costs low is a critical design goal for Confidant, since existing OSNs such as Facebook are popular in large part because they are free. As a result, Confidant relies

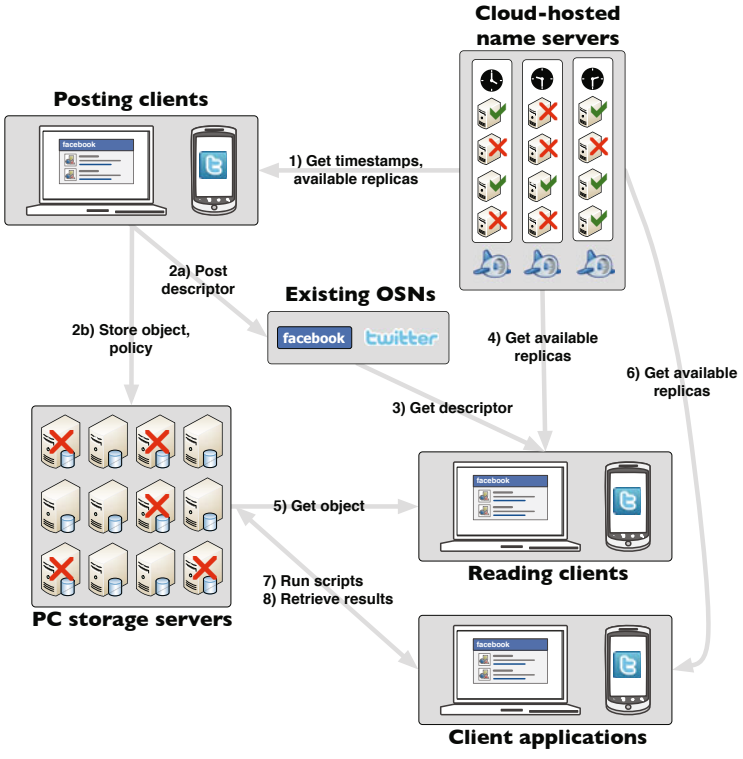


Fig. 1. Confidant architecture

on two low-cost forms of infrastructure: desktop and enterprise PC *storage servers*, and lightweight cloud-based *name servers*. PCs and workstations are a sunk cost for most users and free cloud services such as Google AppEngine and Heroku allow a user to execute a single-threaded server process while maintaining a small amount of persistent state. We assume that every Confidant user controls both a storage server and a name server.

A storage server hosts a user's OSN data and authorizes client read and write requests. Because storage servers can experience transient failures, a user may select a small number of her friends' servers to host *replicas* of her data. Each replica manages a copy of the user's data and participates in anti-entropy protocols to ensure eventual consistency. Storage servers also allow authorized users to run sandboxed application *scripts* directly on their hardware.

Name servers are assumed to be highly available, but due to resource and trust constraints have limited functionality. A name server is only responsible for maintaining two pieces of state: its owner's *logical clock*, and a list of available replicas. Maintaining this state in a highly-available, centralized location is appealing for two reasons. First, placing the list of available replicas in a stable, well-known location simplifies data retrieval. Second, maintaining logical clocks in centralized locations allows Confidant to serialize a user's updates and eliminate write conflicts.

Eliminating conflicts is important because users are likely to access Confidant data via multiple *clients*. A single physical machine such as a laptop can serve as both a storage server and a client, but we explicitly separate client and server roles due to the resource constraints of clients such as mobile phones. Client functionality is limited to uploading and retrieving content and remotely executing application scripts.

2.2 Trust and Threat Model

Trust in Confidant is based on physical control of hardware and inter-personal relationships.

Users trust their clients to read their data, create new data on their behalf, and update the access policies protecting their data. A user's clients are not allowed to create objects on behalf of other users or alter the access policies protecting other users' data.

We assume that a user's storage server and its replicas will participate in anti-entropy protocols, enforce access policies, and correctly run application scripts. Correct execution of scripts requires storage servers to access plaintext copies of data and to preserve the integrity of a script's code, runtime state, and input data. To ensure that replicas meet these trust requirements, Confidant users place their data only on servers controlled by trusted friends.

Serving a user's data from her friends' PCs rather than from third-party servers creates no additional threats to data confidentiality than existing centralized and decentralized OSNs. Users already share their OSN data with their friends and, as with all other OSNs, we assume that users do not collude or share data with unauthorized entities. Software misconfiguration and malware are serious problems for user-managed machines, but these vulnerabilities are not unique to Confidant. If an attacker compromises a Facebook user's PC or mobile device, the attacker can use the Facebook credentials stored on the machine to access any data the owner is authorized to view. Decentralized OSNs such as Persona are also vulnerable to compromised personal machines.

However, even if a user is trusted to preserve the confidentiality of their friend's data, their storage server might not be trusted to preserve the integrity of application scripts. A compromised storage server can corrupt script results by modifying a script's execution, injecting false data, or removing legitimate data. To reduce the likelihood of corrupted script results, we assume that users can identify a small number of friends whose storage servers will behave as expected. Based on several proposals to gauge the strength of social ties using the level of interaction between OSN users [3, 10, 25], it is reasonable to assume that users will find enough storage servers to act as replicas. For example, Facebook has reported that male users regularly interact with an average of seven friends, while women regularly interact with an average of ten friends [21].

Limiting the trust users must place in cloud-based services is an important design goal for Confidant. Thus, our cloud-based name servers are trusted to correctly maintain information about storage servers' availability and a user's logical clock, but are not trusted to access plaintext data. Confidant is agnostic to the mechanism by which users become aware of new data, though for convenience and incremental deployability our prototype implementation uses Facebook. Services like Twitter or open, decentralized alternatives could also be plugged in. Regardless of what notification service is used, Confidant only trusts machines controlled by friends to access plaintext data.

3 Design

In this section, we describe the Confidant design.

3.1 Cryptographic State

Confidant encodes component roles and trust relationships using techniques described by work on Attribute-Based Access Control (ABAC) [26]. We do not support the full power of an ABAC system, and have adopted a subset of techniques (e.g., independently rooted certificate chains and signed attribute-assignments) that are appropriate to our decentralized OSN.

Principals in Confidant are defined by a public-key pair, and *users* are defined by a public-key pair called a *root key pair*. Users generate their own root keys, and distribute their root public key out of band (e.g., through their Facebook profile or via email). A user's root public key is distributed as a self-signed certificate called a *root certificate*; the root private key is kept in a secure, offline location. Through her root key-pair, a user also issues certificates for her storage server (*storage certificate*), name server (*name certificate*), and any clients under her control (*client certificate*): these certificates describe the principal's role (i.e., storage server, name server, or client) and its public key. For each certificate signed by a user's root key pair, the matching private key is only stored on the component, and expiration dates are set to an appropriate period of time. Users also generate *replica certificates* with their root key pair for any storage servers controlled by others who are authorized to serve their objects. All certificates are distributed out of band through a service such as Facebook or via email.

Users encode their inter-personal relationships through *groups*. A group is defined by four pieces of state: 1) a unique user who owns the group, 2) a list of users making up the group's membership, 3) an attribute string, and 4) a secret key. Group owners are the only users who can update a group's state. Group memberships grow monotonically; "removing" a member requires an owner to create a new group with the previous membership minus the evicted member.

A group's string provides a convenient mechanism for assigning attributes to sets of users, which can in turn be used to express access-control policies over sensitive data. For example, user Alice may wish to define groups with attributes such as "New York friends," "college friends," and "family." Group membership does not need to be symmetric. Bob may be included in Alice's group "New York friends," but Bob is not obligated to include Alice in any of the groups he owns.

Group keys are generated on the group owner's storage server and distributed as social attestations [22]; attestations are signed using the key pair of the owner's storage server. Social attestations in Confidant are nearly identical to those in Lockr, except that Confidant attestations also enumerate their group's membership. Within a social attestation, group members are represented by the string description and public key found in their root certificate. If new members are added to the group, the group owner distributes a new social attestation to members reflecting the larger group size. New and updated attestations are distributed epidemically among storage servers; clients periodically synchronize their set of attestations with their owner's storage server. Although there is no bound on the time for a client to receive an attestation, the key embedded

in an existing attestation can remain valid even if the membership enumerated in the attestation becomes stale.

Access policies specify the groups that are allowed to view an object, and are represented by logical expressions in disjunctive normal form (e.g., $(g_0 \wedge g_1) \vee (g_2 \wedge g_3)$), where each literal describes a group, and each conjunction indicates a set of group keys that could be used for authorization.

Finally, because social attestations contain a secret key, they must be handled carefully. Name servers cannot access social attestations since the machines on which name servers execute are physically controlled by a cloud provider rather than a user. Storage servers store copies of any attestations needed to authenticate access requests. Clients store any attestations required to upload new objects or access friends' objects.

Key revocation is known to be a difficult problem, and we use a set of well known techniques to address it. First, certificates and social attestations include an expiration date. If a private or secret key leaks, the certificate expiration date provides an upper bound on the key's usefulness. For group keys, users can generate new keys and attestations to protect any new objects they create. Storage servers can also be asked to ignore group keys that become compromised. This approach should scale well since the number of storage servers hosting a user's data is expected to be on the order of ten machines. We discuss how new policies can be assigned to old objects in the next section.

3.2 Objects and Access Policies

Data in Confidant is managed as *items*. Confidant supports two kinds of items: objects and access policies.

The unit of sharing in Confidant is an *object*. Like Facebook wall posts, comments, and photos, or Twitter tweets, objects are immutable. Every object has a unique *descriptor* with the following format:

$$\{owner, seq, acl\}$$

Descriptors function strictly as names (i.e., not capabilities) and can be embedded in feeds from untrusted services such as Facebook and Twitter without compromising data confidentiality.

The $owner$ and seq fields uniquely identify the object. The $owner$ field of a descriptor is set to the root public key of the user whose client created the object. The seq field is the object's sequence number. Each number is generated by a user's name server when an item is created and is unique for all items created by a user. The acl field is an expression in Confidant's access-policy language.

Objects consist of a meta-data header, followed by the object's content:

$$\{owner, seq, typ, t, len, \langle data \rangle\}$$

The $owner$ and seq fields are identical to those present in the object's descriptor. The typ field indicates the object's format (e.g., text or JPEG image), t field is a wall-clock timestamp. The end of the object is opaque data of length len.

The unit of protection in Confidant is an *access policy*. Access polices are treated as distinct data items with the following representation: $\{owner, seq_{ap}, acl, seq_{obj}\}$. As before, the acl field is an expression in Confidant's access-policy language. seq_{ap} is the sequence number associated with the access policy; this number is unique across all items (objects and policies) created by a user. The $owner$ and seq_{obj} fields of an access policy refer to the object to which the expression applies. To ensure that clients do not attempt to assign access policies to objects they do not own, storage servers must check that a new policy's $owner$ field matches the identity of the issuing client's certificate signer.

Like objects, access policies are immutable, although the binding between objects and polices can change according to the following rule: *an object is protected by the policy with the greatest sequence number that refers to the object*. Application of this rule allows clients to add and remove permissions using a single, simple mechanism. If a client wants to bind a new access policy to an old object, it increments its user's logical clock and creates a new policy using the new sequence number. To invalidate an object, a client can issue a new access policy with a null expression. If two objects are protected by the same logical expression, they will require separate access policies. Note that because the binding between objects and policies can change the acl included in an object descriptor is meant only as a hint [12], and may not reflect the current protection scheme for the object.

There are two potential drawbacks of not aggregating policies across objects: the overhead of storing and transferring extra policy items, and the added complexity of bulk policy changes. However, both drawbacks are minor concerns, given the relatively small number of items that individual users are likely to generate. According to Facebook, the average user creates only 70 data items every month [9]. Even for bulk policy rebindings covering years' worth of user data, iterating through all of a user's items several times should be reasonably fast. As a result, the flexibility to bind arbitrary policy expressions to items at any point in the item's lifetime outweigh the drawbacks.

Confidant's approach to object protection is similar to recent work on managing access policies in Cimbiosys [17, 27]. Both systems manage data as a weakly-consistent replicated state store, and both treat objects and policies as first-class data items. Despite the similarities, there are several important differences between Confidant and Cimbiosys.

First, Confidant serializes all of a user's updates by assigning new items a unique sequence number from the user's name server. This eliminates the complexity and inconvenience of automatically or manually resolving concurrent updates. Avoiding the pain of handling conflicts is an important consideration for OSNs. Experience with the Coda file system found that most write conflicts were caused by users updating their data from multiple clients [14], which mirrors the common behavior of OSN users accessing services from both a PC and mobile device. Confidant's name servers create a single point of failure, but we believe that this is an appropriate tradeoff given inconvenience of handling conflicts and the high availability of cloud services such as Google AppEngine.

Cimbiosys also applies access policies at a coarser granularity than Confidant. Cimbiosys access policies (called *claims*) are bound to *labels* rather than objects. Claims

allow principals to read or write sets of objects that bear the same label (e.g., "photos" or "contacts"). However, because the labels assigned to Cimbiosys items are permanent and claims are expressed in terms of labels, it is impossible for users to change the permissions of a single item within a labeled set; permissions can only be granted or revoked at the granularity of complete sets of items with a particular label. While this is a reasonable design choice for the home-networking setting for which Cimbiosys was designed, it is inappropriate for OSNs.

As the Cimbiosys authors point out, it is important for Cimbiosys users to label items correctly when they are created. This is too great a burden for OSN users, for whom fine-grained control is useful in many situations. For example, consider a user who initially labels an item "mobile photo" (perhaps accidentally) and shares it with her family and friends. Under Cimbiosys, if she later decided that it was a mistake to give her family access to the image, she would have to revoke her family's access to all items labeled "mobile photo," including any other images she might want to continue sharing with them. In Confidant, the user could simply create a new policy with a pointer to the image she would like to hide and a policy expression including only friends.

It should be noted that Cimbiosys could provide the same flexibility as Confidant by assigning each object a unique label, but the designers did not pursue this approach due to its perceived lack of efficiency. This design decision appears to be related to Cimbiosys's focus on defining policy claims in terms of principals rather than groups of principals, and an implicit assumption about the number of objects a user owns. SecPAL (Cimbiosys's policy logic) supports groups of principals, but without groups, creating a separate claim for each principal authorized to access each item in a massive data set might introduce scalability problems. Confidant can avoid these issues because individual user's OSN data sets are relatively small, allowing us to define policies in terms of groups of principals.

3.3 Name Servers

As described in Section 2.1, each user runs a name server within a low-cost cloud service such as Google AppEngine. Name servers manage two pieces of state: a list of IP addresses corresponding to online replicas and a logical clock. Entries in the list of IP addresses also include an expiration time, and become invalid if not updated in time. Name servers also maintain a list of storage servers authorized to act as replicas.

Retrieving a list of replicas is similar to a DNS lookup. Requests are unauthenticated, require no arguments from the caller, have no side-effects, and return the name server's list of $\langle$ IP addresses, public-key $\rangle$ pairs for each valid storage-server as well as the current value of the user's logical clock. Name servers use SSL to preserve the integrity of queries.

Storage servers set the IP address where they can be reached by periodically contacting the name servers associated with the replicas they host. These calls refresh the expiration time of the server's entry and allow the server's IP address to be returned as part of a lookup. Expiration times are intended to be on the order of tens of minutes. Because only authorized storage servers should be allowed to serve as replicas for a user's data, refreshing a server entry must be authenticated. Initial lease renewals require the name server and storage server to mutually authenticate and establish a session key

Table 1. Storage-server messages

Message	Format
Store request	$\{\{g_0, g_1, \ldots g_n\}, cert_C, \{replicas, obj, ap, rand, \{hash(obj, ap)rand\}_{K_C^-}\}_{gk_0, gk_1, \ldots gk_n}\}$
Policy update	$\{cert_C, replicas, ap, rand, \{hash(ap), rand\}_{K_C^-}\}$
Retrieve request	$\{owner, seq, \{g_0, g_1, \ldots g_n\}\}$
Retrieve response 1	$\{\{g_0, g_1, \ldots g_n\}, cert_R, \{obj, ap, rand, \{hash(obj, ap), rand\}_{K_R^-}\}_{gk_0, gk_1, \ldots gk_n}\}$
Retrieve response 2	$\{cert_R, ap, rand, \{hash(ap), rand\}_{K_R^-}\}$

using their signed certificates, but once the session key has been established it is used to authenticate future requests.

A name server's logical clock is used to assign sequence numbers to items and to help replicas synchronize when they come back online. The value of the logical clock increases monotonically. When a client wants to assign a sequence number to a new item, it requests an increment; the name server responds by adding one to the existing value and returning the new value of the clock. Since only clients under the user's control should be allowed to advance the clock, increment requests are authenticated. As with entry-refresh requests, clients and name servers initially establish a session key with their signed certificates, and use the session keys to authenticate future requests.

3.4 Storage Servers

Each Confidant user runs a storage server that contains plaintext copies of all of her objects and access policies. A storage server may also act as a *replica* for another user if the other user trusts the server's owner to 1) read all of her objects, 2) enforce access policies, and 3) preserve the integrity of any application scripts run on the server. As explained in Section 2.2, it is reasonable to assume that users can identify on the order of ten trustworthy replicas.

Once replicas have been selected, the user in control of each replica-set member must install its storage server's public key at the data owner's name server. Also, since replicas must authorize requests on behalf of the data owner, each member of a replica set must have access to all of the social attestations generated by the data owner. Sharing attestations with replicas does not affect confidentiality since, by definition, replicas already have full read access to the data owner's objects. Storage servers store objects and their associated access policies in a relational database and local processes access Confidant data through SQL queries.

For the rest of this paper, we assume that the entirety of a user's data is managed by a replica set, but Confidant is general enough to accommodate multiple data partitions. For example, users wishing to separate their data into work data and personal data can do so by creating separate sequence numbers and replica sets for each partition. The number of distinct data sets that a user wants to maintain with Confidant is limited by the amount of state she can afford to host in the cloud and the number of storage servers she trusts to host each partition.

Consistency. We apply an eventual consistency model to data stored within a replica set and rely on epidemic propagation to synchronize storage servers [5, 11]. Because objects and access policies are immutable, ensuring a consistent ordering of updates across replicas is not material. We are only concerned with whether the set of data items stored on behalf of a data owner is consistent with the set of all items the data owner has created. Servers achieve eventual consistency by applying standard anti-entropy techniques, which are described in greater detail in the Confidant Technical Report [13].

Updating and retrieving items. The messages used to update and retrieve items are listed in Table 1.

To add a new object, a client first retrieves a list of online replicas and two new sequence numbers (one for the object and one for the object's access policy). To store the new items, a client connects to the first server in the list returned by the name server and submits the store-request message described in Table 1. The header corresponds to a conjunction, $\{g_0, g_1, \ldots g_n\}$, from the the access policy ap; this indicates which group keys, $gk_0, gk_1, \ldots gk_n$, are used to protect the message in transit. The client also sends the replica its client certificate, $cert_C$.

The message payload consists of an object obj, an access policy ap, a random nonce $rand$, and a signed hash of the object and access policy. The client certificate and signed hash prove to the storage server that the update was generated by a trusted client. If store requests were only protected with group keys, then anyone with access to the proper social attestations could create items on the user's behalf; social attestations are meant to confer only read access, not write access. Note that the store-request message is vulnerable to a harmless man-in-the-middle attack in which another client swaps in its own certificate and re-encrypts the payload with its own private key.

Once a server has unpacked and verified a store request, it commits the new items to its local database and returns control to the client. The storage server is now responsible for propagating the update to the other replicas listed in the $replicas$ field of the message payload. We rely on anti-entropy to spread new items to the rest of the replica set in the face of network partitions and server failures. Storage servers' local database and the protocol for authorizing retrieve requests are described in more detail in the Confidant Technical Report [13].

Application framework. Our primary motivation for leveraging social relationships to select replicas is to enable scalable, general-purpose applications without sacrificing data confidentiality. Prior decentralized OSNs assumed that storage servers were not trusted to view plaintext data, which limited the class of operations that storage servers could perform on OSN data to feature-based searches (e.g., key-word [6, 7, 20] or location-based [16] search). Unfortunately, many popular and emerging OSN applications such as Twitter's trending topics and face.com's face-recognition service require iterating over a large corpus of plaintext data. Services such as these require a general-purpose distributed programming framework like MapReduce [4]. However, unless storage servers are trusted to view plaintext data, such applications can only be implemented by downloading an entire encrypted corpus to a client, where it must be decrypted and analyzed. This approach will not scale for clients executing on resource-limited desktop PCs and mobile devices.

Since a Confidant user's replicas are trusted, the biggest challenge in designing a scalable application framework is balancing the need to safely sandbox code executed on storage servers and and provide a rich programming API. We do not claim that our solution is perfect, only that it represents a reasonable point in the design space.

The unit of execution for Confidant's application framework is a *script*. Scripts must be constrained so that they cannot harm the storage servers on which they execute or access any unauthorized data. To protect the host storage server, scripts are written in Python and execute in a sandboxed Python environment, with pre-built libraries and modules; scripts run under a unique, temporary uid with limited privileges. The *chroot* utility allows storage servers to start scripts in a temporary "jail" directory such that they will not be able to access any other part of the file system.

Storage servers impose CPU, core size and execution time limits on scripts, and run a per-script reference monitor that mediates scripts' access to the object database. The DBus message system is used as an interprocess communication channel between a script and its reference monitor. Similar to Java RMI, the script obtains a proxy object of the reference monitor from the DBus registry service and uses its interface to query the object database. Scripts submit SQL queries to the reference monitor, which examines and rewrites their queries by adding predicates so that the query only returns data that is authorized by the group credentials submitted with the script. This creates a clean, flexible, and familiar programming environment for developers and relies on existing database mechanisms to enforce confidentiality. After the script completes its work it creates a file in its temporary directory which is returned to the requesting client as a response. If a script exceeds its resource limits the reference monitor terminates it and the storage server sends back an error message.

4 Implementation

We have implemented a Confidant prototype based on the design described in Section 3. This section describes our client, name server, and storage server implementations. We also describe three applications that we have implemented on top of Confidant.

4.1 Client

Our client is implemented as a Firefox web-browser extension that rewrites Facebook web pages and communicates with Confidant name and storage servers. Our Firefox extension transparently integrates Confidant data with a user's Facebook page.

We derive several benefits from interoperating with Facebook. One, users continue using their existing Facebook accounts, thus leveraging their considerable investment in creating social connections there and learning how to use the many popular features. Two, we take advantage of Facebook as a reliable medium for storing object descriptors and distributing them throughout the social graph. For more details on our client implementation, please see the Confidant Technical Report [13].

Facebook remains an untrusted service that should not have access to users' secret keys or sensitive data. Our browser extension modifies Facebook's default behavior by listening for browser events such as document loads, form submits, and button clicks.

When these events happen, our handling functions are triggered to run prior to Facebook's original functionality. For example, when a user wants to share a status update or post a wall message, the browser extension intercepts the control flow. Once in control, it contacts the user's Confidant name server to retrieve sequence numbers for the new object and policy as well as the IP address of the available replicas for storing these items. It then creates an object and policy with the appropriate descriptor and sends the items to a replica. Finally, the extension substitutes the original content from the Facebook page with the object descriptor to be sent to Facebook.

To retrieve a status update the browser extension scans the loaded page for the object descriptors, parses them, obtains a correct replica IP address from the name server, and downloads the object. Then the extension performs integrity checks, decrypts the data, and replaces the descriptor with the actual content.

For uploading pictures we modified Facebook's "Simple Uploader" that accepts individual pictures. The uploader proceeds in two steps. First, instead of uploading the actual picture to Facebook our extension sends a dummy image to be stored on Facebook. Next, when a user has to add a description of the picture, the extension creates a new object and descriptor. The object consists of the original picture and the picture's description. Using the IP address of a replica from the name server, the extension sends the object to Confidant and substitutes the actual description with the object's descriptor to be sent to Facebook.

Retrieving a picture in Confidant works similarly to a status update retrieval with the exception that the actual image is downloaded locally and linked to the web-page directly from the filesystem.

4.2 Name Server

As described in Section 3.3, each user runs a lightweight name server that maintains a list of available replicas. We host this server in the Google AppEngine. AppEngine is a highly available and low-cost cloud-computing infrastructure where users can run applications written in several languages; we used Python. Google does not charge for running applications until they exceed a free-resource quota. We engineered our name server so that its resource utilization should remain comfortably within AppEngine's free quota.

4.3 Storage Server

As described in Section 3.4, each user also runs a storage server with multiple roles: it maintains the primary copy of the user's sensitive data; it maintains a replica of the data belonging to a subset of the user's friends; it arbitrates and serves requests for this data; and it runs application scripts submitted for execution by the user's friends.

We implemented a storage server prototype in Python using the Django web framework. We also use MySQL to store data, Apache2 for serving requests, and JSON as a lightweight data-interchange protocol.

4.4 Applications

As described in Section 3.4, Confidant allows application scripts to execute on the distributed collection of storage servers. We have implemented the three representative applications described below:

Keyword-search Script: The *keyword-search script* is a simple script that searches through a user's friends' accessible status updates and returns those that satisfy criteria such as falling within a date range or having specific keywords. This is the simplest script we implemented. It creates a SQL statement with the required conditions, receives the result from the reference monitor, encrypts the result, and sends back the encrypted data to the client.

Trending-Topics Script: The *trending-topics script* calculates the most frequently used words within a user's friends' status updates. This script prepares a dictionary object $[\text{"}word1\text{"} : \text{"}count\text{"}, \text{"}word2\text{"} : \text{"}count\text{"}, ...]$ using the available status updates and wall messages on each storage server. We eliminate common words that should not be interpreted as trends such as *"I"*, *"me"*, *"am"*, and *"you"*. A client downloads these pre-processed objects from their friends' replicas and merges them into a single list. The client then sorts the list by value to produce the most trending keywords.

Face-Detection Script: We implemented a *face-detection script* that returns friends' pictures with a predefined number of faces in each of the returned pictures. For example, with $N = 1$ the script will return only portrait pictures, however with $N \geq 5$ it will return group photos such as pictures from a party or a conference. We implemented the face detection algorithm using the Python wrapper for OpenCV and we assume that the sandboxed environment has the wrapper along with the OpenCV library itself.

4.5 Untrusted Storage Server

In order to compare Confidant with the alternative approach to decentralized OSNs where storage servers are considered untrusted, we implemented a simple server that stores encrypted data and returns it to the requester with minimal security checks. We use this server to help evaluate the performance of Confidant in application tasks such as finding trending topics and face detection.

5 Evaluation

In evaluating Confidant, we sought answers to the following questions:

- How does the performance of our application framework compare to approaches that rely on untrusted storage servers?
- How many trusted friends will Confidant users need to ensure that at least one replica is always available?

Note that the Confidant Technical Report also describes the results from experiments measuring the latency of creating and retrieving photo and text objects [13]. We have omitted these results to focus on scalability and availability.

5.1 Application Performance

In this section, we compare the performance of Confidant's application framework to the performance of approaches that rely on untrusted storage servers. We measured the end-to-end latencies of the face-detection, keyword-search, and trending-topics scripts described in Section 4.4 and compared them to the latencies of fetching data encrypted under AES from an untrusted server and processing it locally. We refer to the latter approach as *Encrypted*.

All experiments used EC2 virtual machines as trusted storage servers. Using Amazon's EC2 allowed us to test the scalability of Confidant. We used a single untrusted server running on EC2 for all Encrypted experiments. Varying the number of untrusted servers had almost no effect on end-to-end latency of our Encrypted experiments because the decryption and processing time spent at the client dominated our measurements.

For a client, we used a consumer laptop (Intel Core 2 Duo processor, 1.86GHz, 6 MB L2, and 2 GB of memory) on Duke University's network. The client ran a multi-threaded python program that queried storage servers and processed replies as required. A separate thread was spawned for each request.

To populate the remote servers for our trending-topics and keyword-search experiments, we used status updates collected from Facebook by researchers at UCSB [18]. We used images from publicly available datasets from computer vision research groups at CMU, MIT and Cal Tech for evaluating the face-detection script's performance. Our image corpus contained 70 images altogether. We tuned the parameters of the face-detection algorithm so that it was reasonably accurate and required about 500 ms to process a 30Kb picture. It should be noted that we did not evaluate the accuracy of our face detection algorithm and, thus, we do not report how many of the returned pictures were false-positives. The average size of a picture used for the evaluation is 30 KB, with sizes ranging from 8Kb to 200Kb.

Confidant achieves compute scalability by off-loading computational tasks to trusted remote servers, while the Encrypted approach must first download data to a client and decrypt it before processing it locally. To isolate the effect of computational scalability on end-to-end latency, during our experiments the Confidant clients and Encrypted clients only differed in how much processing they performed rather than the volume of data they received. For the keyword-search experiments, we partitioned 1,000 status updates across the remote storage servers. For the trending-topics experiments, remote Confidant scripts collectively returned dictionaries based on 1,000 status updates to the client. For the face-detection experiments, we assigned each remote server 10 photos. Finally, for our experiments, we conservatively assumed that each of the client's friends had only one replica available.

Figure 2 shows the average over 20 trials of each script's latency under Confidant divided by the latency to perform the same task under the Encrypted scheme. Please note the logarithmic scale of the y-axis. Standard deviations for all averages were less than 2%. For 100 friends, Confidant was between 3 and 30 times faster (trending topics and face detection, respectively). While all three scripts performed better under Confidant, the face-detection script benefited the most from parallel remote execution due to the computational complexity of detecting faces in images. Trending topics

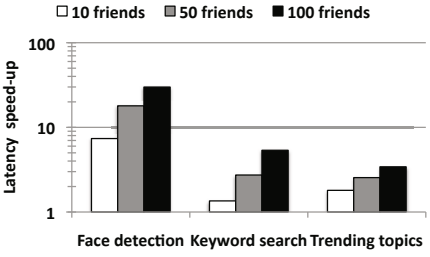


Fig. 2. Script performance

benefits the least due to the computational complexity of merging dictionaries from many sources that must be performed at the client. Nonetheless, these results demonstrate the performance and scalability benefits of Confidant's application framework.

5.2 Availability

Confidant relies on replicated storage servers to ensure that clients can successfully submit and retrieve items. The availability of a user's replica set is a function of two properties: the number of trusted friends a user has (i.e., the size of the replica set), and the availability of individual replica-set members.To explore the impact of these properties on the rates at which clients can successfully read and write, we simulated Confidant using two types of traces. To characterize how many friends users have and how they interact with their friends,we used a trace of the Facebook graph and wall messages collected at MPI [24]. This trace captures links and interactions between users and consists of approximately 60,000 users, connected by over 0.8 million links, with an average node degree of 25.6.

To characterize storage-server availability, we used the well-known Gnutella trace [19] and Microsoft-workstation trace [2]. The Microsoft-workstation trace is much more forgiving and machines have a much higher average online time than hosts in the Gnutella trace. It should be noted that there were many IP addresses listed in the Gnutella trace that were never online, and we removed all such nodes. This left us with approximately 15,000 hosts over 60 hours. From the Microsoft trace, we used roughly 52,000 hosts' records over 35 days.

Since the MPI trace contains more users than hosts in the Gnutella or Microsoft-workstation traces, we pruned the MPI trace to fit each availability trace. First, we sorted users in the MPI trace by level of interactivity (i.e., the number of wall posts and comments written and received), and assigned the users who were the most active a host from the availability trace. Connections to users who were not among the most active were cut. The resulting subgraph was denser than the original, exhibiting an average connection degree of 44.8 and 30.2, for the top 15,000 (Gnutella) and top 52,000 users (Microsoft), respectively. 86% of the top 15,000 users have 10 friends or more, while 61% of the top 52,000 users have 10 friends or more.

We included the most-active Facebook users from the MPI traces because they generated the most write events for our simulation. Of course, there is a correlation between activity and connectedness, as our increased average node degree demonstrates. This correlation biases in our favor since the resulting graph includes a larger fraction of highly-connected users. However, even with the higher average node degree of 44.8, this is still substantially lower than the average of 130 reported by Facebook [9].

The relationship between a user's activity on Facebook and the availability of her storage server is unknown. As a result, we explored three possible ways of assigning users in the MPI trace to hosts in our availability traces: randomly, by connection rank (a more connected user was assigned higher availability), and by interaction rank (a more active user was assigned higher availability). Under each of these mappings, nodes had a maximum replication factor of 10. If a user had 10 or more friends, its replicas were chosen to be the 10 users it interacted with the most. Nodes with fewer than 10 connections used all of their connections as replicas.

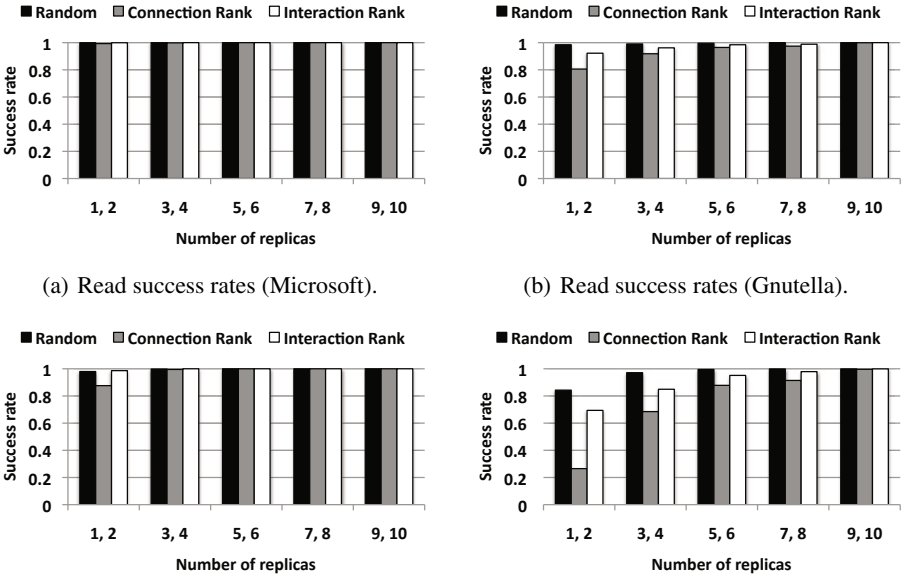
We assumed that reads and writes were generated by an always-available client such as a mobile device. We simulated writes using interactions from the MPI trace. For a write event, if at least one member of a client's replica set was available, the write was considered a success and was added to a read queue for each of its friends. Writes generated when no replica-set members were available counted as a failure and were not added to friends' read queues.

Reads were not captured in the MPI trace so we generated them synthetically. Clients chose a random time between 30 minutes and 60 minutes, and after that time passed attempted to retrieve the objects in their read queue. This read rate is consistent with the reported behavior of Facebook users [9]. Read failures occurred if a client tried to read an object while no member of the replica set was available, or none of the online replica servers had the object at that time. Otherwise, the read was considered a success. After clearing its read queue, clients chose another random period before they read again.

Figure 3(a) and Figure 3(c) show the read and write success rates for simulations using the Microsoft availability trace, while Figure 3(b) and Figure 3(d) show the read and write success rates using the Gnutella trace. As expected, nodes with more replicas had higher read and write success rates, regardless of how Facebook users were assigned host availability records. The overall higher read success rates across experiments are partially an artifact of failed write attempts being suppressed in the read results; if a node failed to perform a write, no other nodes attempted to read it.

One of the most striking features of our results is how much worse nodes with fewer replicas fared when low connectivity was correlated with low availability. This is because nodes with low connectivity were penalized twice. Not only did these users have access to fewer replicas, but their own server was assigned low availability as well. This combination of factors led to significantly lower success rates than for nodes with low connectivity under the other schemes or nodes with more replicas.

Unsurprisingly, read and write success rates under the Gnutella trace are much worse than under the Microsoft-workstation trace. The Gnutella trace likely provides a close to worst-case scenario for host availability since it captures client-process uptime rather than machine uptime. On the other hand, results from the Microsoft-workstation trace are likely close to a best-case scenario since the trace captures machine uptime in a



(a) Read success rates (Microsoft).

(b) Read success rates (Gnutella).

(c) Write success rates (Microsoft).

(d) Write success rates (Gnutella).

Fig. 3. Simulation results

corporate environment. Reality likely lies somewhere in between. It is important to note that even under the less-forgiving availability of the Gnutella trace, users with ten replicas fared well. The worst case for ten replicas was under a connection-based mapping, which generated a write-success rate of 99.6%. Under the Microsoft-workstation trace, three or four replicas provided a write-success rate of 99.5% using the connection-based mapping. Users with more than four replicas under the Microsoft trace had perfect read and write rates for all mappings. All of these results bode well for Confidant. Facebook users have an average of 130 friends, and we suspect that it would be straightforward for most users to identify 5-10 trusted friends who are willing to host their data.

6 Related Work

FriendStore [23] is a backup system that also identifies trustworthy storage sites through inter-personal relationships. FriendStore leverages trustworthy remote storage servers to mitigate long-standing fairness problems in peer-to-peer systems. A primary difference between Confidant and FriendStore is the nature of the data they must manage. Backup data is not meant to be shared and thus FriendStore does not require the same level of complexity to manage access-control policies that Confidant does.

While Confidant leverages knowledge of the social graph to provide data privacy without compromising data processing in a decentralized OSN, SPAR [15] uses social

information to improve the scalability of centralized OSNs such as Facebook or Twitter. By co-locating the data of socially proximate users on the same physical machine, SPAR can reduce the time to compose a user's update feed and eliminate network traffic. SPAR may also be useful for decentralized systems such as Confidant for reducing the number of storage servers clients have to communicate with.

As discussed Section 3.2, recent work on Cimbiosys [17, 27] comes closest to Confidant's scheme for managing replicated access policies. The two systems have a great deal in common, but their are two main differences. First, Confidant eliminates write conflicts by serializing a user's updates through her name server. Second, in Cimbiosys, policies are bound to immutable object attributes (labels), while in Confidant, policies are bound to individual data items. By binding policies at a finer granularity is more appropriate for OSNs, where users often want to change the permissions of a single item. However, Confidant can also enable efficient bulk policy changes due to the limited scale of user's personal OSN data sets.

Lockr [22] is an identity-management tool for OSNs that allows users to codify their relationships through social attestations. Confidant borrows this idea from Lockr and uses it to manage groups of users.

Persona [1] is one of many systems [6, 7, 16, 20] that assumes that remote storage servers are untrusted. We have articulated the advantages of locating trusted storage servers throughout this paper. However, it is worth noting that Confidant's collusion protections are weaker than those provided by Persona. Persona uses an attribute-based encryption scheme to defend against attacks in which colluding users combine keys to falsely claim membership in an intersection of groups. In Confidant we trade robustness to collusion attacks for a faster, simpler protection scheme based on common symmetric-key algorithms.

7 Conclusion

We have presented Confidant, a decentralized OSN designed to support a scalable application framework. The key insight behind Confidant is that friends who already have access to a user's data may be trusted to serve it as well. Trace-based simulations and experiments with a Confidant prototype demonstrate the feasibility of our approach.

Acknowledgements. The work by the co-authors from Duke University was supported by the National Science Foundation under award CNS-0916649, as well as by AT&T Labs and Amazon.

References

1. Baden, R., et al.: Persona: an online social network with user-defined privacy. In: SIGCOMM 2009 (2009)
2. Bolosky, W., et al.: Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. SIGMETRICS (2000)
3. Chun, H., et al.: Comparison of online social relations in volume vs interaction: a case study of cyworld. In: IMC 2008 (2008)

4. Dean, J., et al.: Mapreduce: simplified data processing on large clusters. Commun. ACM (2008)
5. Douglas, T., et al.: Managing update conflicts in bayou, a weakly connected replicated storage system. In: SOSP (1995)
6. Shi, E., et al.: Multi-dimensional range query over encrypted data. In: IEEE Symposium on Security and Privacy (2007)
7. Fabbri, D., et al.: Privatepond: Outsourced management of web corpuses. In: WebDB (2009)
8. Facebook site info from alexa.com, http://www.alexa.com/
9. Facebook statistics, http://www.facebook.com/press/
10. Gilbert, E., et al.: Predicting tie strength with social media. In: CHI 2009 (2009)
11. Golding, R.A.: A weak-consistency architecture for distributed information services. Computing Systems (1992)
12. Lampson, B.W.: Hints for computer system design. IEEE Software (1983)
13. Liu, D., Shakimov, A., Cáceres, R., Varshavsky, A., Cox, L.P.: Confidant: Protecting OSN Data without Locking it Up. Technical Report TR-2010-04, Duke University, Department of Computer Science, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA (August 2010)
14. Noble, B., Satyanarayanan, M.: An empirical study of a highly available file system. In: SIGMETRICS (1994)
15. Puhol, J., et al.: The little engine(s) that could: Scaling online social networks. In: SIGCOMM 2010 (2010)
16. Puttaswamy, K., Zhao, B.: Preserving privacy in location-based mobile social applications. In: HotMobile (2010)
17. Ramasubramanian, V., et al.: Cimbiosys: a platform for content-based partial replication. In: NSDI 2009 (2009)
18. Sala, A., et al.: Measurement-calibrated graph models for social network experiments. In: WWW 2010 (2010)
19. Saroiu, S., et al.: Measuring and analyzing the characteristics of napster and gnutella hosts. Multimedia Syst. (2003)
20. Song, D., et al.: Practical techniques for searches on encrypted data. In: IEEE Symposium on Security and Privacy (2000)
21. The Economist. Primates on facebook (February 2009)
22. Tootoonchian, A., et al.: Lockr: better privacy for social networks. In: CoNEXT 2009 (2009)
23. Tran, D., et al.: Friendstore: cooperative online backup using trusted nodes. In: SocialNets 2008 (2008)
24. Viswanath, B., et al.: On the evolution of user interaction in facebook. In: WOSN (2009)
25. Wilson, C., et al.: User interactions in social networks and their implications. In: EuroSys 2009 (2009)
26. Winsborough, W., et al.: Towards practical automated trust negotiation. In: Policy 2002 (2002)
27. Wobber, T., et al.: Policy-based access control for weakly consistent replication. In: EuroSys 2010 (2010)

Live Deduplication Storage of Virtual Machine Images in an Open-Source Cloud

Chun-Ho Ng, Mingcao Ma, Tsz-Yeung Wong, Patrick P.C. Lee, and John C.S. Lui

Dept. of Computer Science and Engineering,
The Chinese University of Hong Kong, Hong Kong
{chng,mcma,tywong,pclee,cslui}@cse.cuhk.edu.hk

Abstract. Deduplication is an approach of avoiding storing data blocks with identical content, and has been shown to effectively reduce the disk space for storing multi-gigabyte virtual machine (VM) images. However, it remains challenging to deploy deduplication in a real system, such as a cloud platform, where VM images are regularly inserted and retrieved. We propose LiveDFS, a live deduplication file system that enables deduplication storage of VM images in an open-source cloud that is deployed under low-cost commodity hardware settings with limited memory footprints. LiveDFS has several distinct features, including spatial locality, prefetching of metadata, and journaling. LiveDFS is POSIX-compliant and is implemented as a Linux kernel-space file system. We deploy our LiveDFS prototype as a storage layer in a cloud platform based on OpenStack, and conduct extensive experiments. Compared to an ordinary file system without deduplication, we show that LiveDFS can save at least 40% of space for storing VM images, while achieving reasonable performance in importing and retrieving VM images. Our work justifies the feasibility of deploying LiveDFS in an open-source cloud.

Keywords: Deduplication, virtual machine image storage, open-source cloud, file system, implementation, experimentation.

1 Introduction

Cloud computing makes computing and storage resources available to users on demand. Users can purchase resources from commercial cloud providers (e.g., Amazon EC2 [1]) in a pay-as-you-go manner [2]. On the other hand, commercial clouds may not be suitable for some users, for example, due to security concerns [25]. In particular, from developers' perspectives, commercial clouds are externally owned and it is difficult to validate new methodologies for a cloud without re-engineering the cloud infrastructures. An alternative is to deploy a self-manageable cloud using *open-source* cloud platforms, such as Eucalyptus [18] and OpenStack [22]. Such open-source cloud platforms can be deployed within in-house data centers as private clouds, while providing functionalities similar to commercial clouds. For example, Eucalyptus resembles the functionalities of Amazon EC2. Note that an open-source cloud can be deployed using *low-cost commodity hardware and operating systems* that are commonly available to general users [18].

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 81–100, 2011.

Similar to commercial clouds, an open-source cloud should provide users with *virtual machines (VMs)*, on which standard applications such as web servers and file storage can be hosted. To make deployment flexible for different needs of users and applications, it is desirable for the open-source cloud to support a variety of versions of VMs for different types of configurations (e.g., 32-bit/64-bit hardware, file systems, operating systems, etc). A major challenge is to scale up the storage of a large number of VM images, each of which is a file that could be of gigabytes. Certainly, increasing the storage capacity for hosting VM images is one option. However, this also implies higher operating costs for deploying an open-source cloud under commodity settings.

One promising technology for improving storage efficiency of VM images is *deduplication*, which eliminates redundant data blocks by creating smaller-size pointers to reference an already stored data block that has identical content. One major application of deduplication is the data backup in Content Addressable Storage (CAS) systems [23], in which each data block is identified by its *fingerprint* computed from a collision-resistant hash of the *content* of the data block. If two data blocks have the same fingerprint, then they are treated as having the same content. Recent studies [13,11] show that the VM images of different versions of the same Linux distribution generally have a high proportion of identical data blocks (e.g., about 30% of overlap in adjacent Fedora distributions [13]). Hence, deduplication can actually enhance the storage utilization of VM images. On the other hand, to enable deduplication for VM image storage in a cloud, we need to address several deployment issues:

- **Performance of VM operations.** Existing studies mainly focus on the effectiveness of using deduplication to save space for storing VM images, but there remain open issues regarding the deployment of deduplication for VM image storage. In particular, it remains uncertain if deduplication degrades the performance of existing VM operations, such as VM startup.
- **Support of general file system operations.** To make the management of VM images more effective, a deduplication solution should allow general file system operations such as data modification and deletion. For example, if an old VM version is no longer used, then we may want to purge the VM image file from the cloud for better maintenance. However, current deduplication techniques are mainly designed for backup systems, which require data be immutable and impose a write-once policy [23] to prevent data from being modified or deleted.
- **Compatibility with low-cost commodity settings.** Although there have been commercial file systems (e.g., SDFS [20] and ZFS [21]) that support efficient I/O operations while allowing deduplication, they are mainly designed for enterprise servers with a large capacity of main memory. Some deduplication systems [8,15] use flash memory to relieve the main memory requirement of deduplication. To make deduplication compatible with commodity settings, a deduplication solution should preserve the I/O performance with reasonable memory footprints and standard commodity hardware configurations.

In this paper, we present a live deduplication file system called *LiveDFS*, which enables deduplication storage of VM image files in an open-source cloud. In particular, we target the open-source cloud platforms that are deployed in low-cost commodity hardware and operating systems. LiveDFS supports general file system operations, such as

read, write, delete, while allowing *inline deduplication* (i.e., on-the-fly deduplication is applied to data that is to be written to the disk). LiveDFS consists of several design features that make deduplication efficient and practical.

- **Spatial locality.** LiveDFS stores only partial deduplication metadata (e.g., fingerprints) in memory for indexing, but puts the full metadata on disk. To mitigate the overhead of accessing the metadata on disk, LiveDFS exploits spatial locality by carefully placing the metadata next to their corresponding data blocks with respect to the underlying disk layout.
- **Prefetching of metadata.** LiveDFS prefetches deduplication metadata of the data blocks in the same block group into the page cache (i.e., the disk cache of Linux). This further reduces the seek time of updating both metadata and data blocks on the disk.
- **Journaling.** LiveDFS supports journaling, which keeps track of file system transactions and enables crash recovery of both data blocks and fingerprints. In addition, LiveDFS exploits the underlying journaling design to combine block writes in batch and reduce disk seeks, thereby improving the write performance.

LiveDFS is *POSIX-compliant*, so its above design features are implemented in such a way that is compliant with the Linux file system layout. It is implemented as a Linux kernel-space driver module, which can be loaded to the Linux kernel without the need of modifying and re-compiling the kernel source code. To justify the practicality of LiveDFS, we integrate it into an open-source cloud platform based on OpenStack [22]. Thus, LiveDFS serves as a storage layer between cloud computing nodes and the VM storage backend. We conduct extensive experiments and compare LiveDFS and the Linux Ext3 file system (Ext3FS). We show that LiveDFS saves at least 40% of storage space for VM images compared to Ext3FS. Given that deduplication introduces fragmentation [24], we also evaluate the performance overhead of LiveDFS in inserting and retrieving VM images in a cloud setting. To our knowledge, this is the first work that addresses the practical deployment of live deduplication for VM image storage in an open-source cloud.

The remainder of the paper proceeds as follows. In Section 2, we present the design of LiveDFS as a deduplication-enabled file system. In Section 3, we explain how LiveDFS is implemented and can be deployed in an open-source cloud based on OpenStack. In Section 4, we present the empirical experimental results. In Section 5, we review the related work in applying deduplication in storage. Finally, in Section 6, we conclude this paper and present future work.

2 LiveDFS Design

LiveDFS is a file system that implements inline deduplication for VM storage and is deployed as a storage layer for an open-source cloud. It is designed for *commodity hardware and operating systems*. For commodity hardware, we consider the native 32-bit/64-bit hardware systems with a few gigabytes of memory. Specifically, we seek to reduce the required memory capacity to reduce the hardware cost. For commodity operating systems, we consider Linux, on which LiveDFS is developed.

We make the following assumptions for LiveDFS design. LiveDFS is deployed in a single storage partition. It only applies deduplication to the stored data within the same partition, but not for the same data stored in different partitions. Nevertheless, it is feasible for a partition to have multiple storage devices, such that deduplication is applied in the file-system level, while data striping is applied in the storage devices and is transparent to the file system. In addition, LiveDFS mainly targets for VM image storage. We do not consider applying deduplication for other types of data objects, which may not have any content similarities for deduplication (e.g., encrypted files).

2.1 Primitives

We design LiveDFS as a branch of the Linux Ext3 file system (Ext3FS) [6]. LiveDFS supports general file system I/O operations such as read, write, and delete. It also supports other standard file system operations for files, directories, and metadata (e.g., changing directories, renaming a file, setting file attributes). Figure 1 depicts the file system layout of LiveDFS, which follows the layout of Ext3FS except that LiveDFS allows block sharing. In the following, we explain the major primitives of LiveDFS as a deduplication-enabled file system.

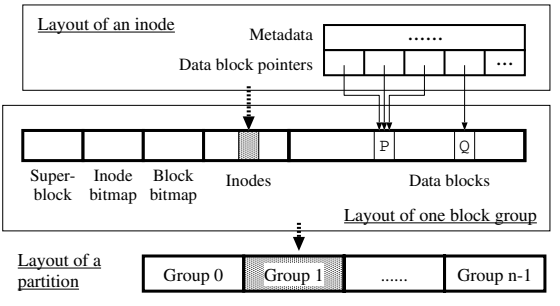


Fig. 1. File system layout of LiveDFS, which is similar to the Ext3FS but supports block sharing

Typical storage systems organize data into *blocks*, which could be of fixed size or variable size. Most CAS backup systems divide data into variable-size blocks, so as to exploit different granularities of duplicated contents and achieve a higher deduplication rate. The merits of using variable-size blocks in backup systems are studied in [28]. Nevertheless, we choose the fixed-size block implementation in LiveDFS with two reasons. First, most commodity CPUs (e.g., Intel x86) support fixed-size memory pages only. Thus, most mainstream file systems adopt the fixed-size block design to optimize the use of memory pages. Second, [11] shows that for VM image storage, the deduplication efficiencies of using fixed-size and variable-size blocks are similar. Hence, we define a *block* as a fixed-size data unit thereafter.

Similar to Ext3FS, LiveDFS arranges blocks into *block groups* (see Figure 1), each storing the metadata of the blocks within the same group. One advantage of using block groups is to reduce the distance between the metadata and their corresponding blocks on disk, thereby saving the disk seek overhead.

In Ext3FS, each file is allocated an *inode*, which is a data structure that stores the metadata of the file. In particular, an inode holds a set of block pointers, which store the *block numbers* (or addresses) of the blocks associated with the file. LiveDFS also exploits the design of an inode and uses block numbers to refer to blocks. In particular, if two blocks have the same content, then we set their block numbers to be the same (i.e., both of them point to the same block). This realizes the *block-sharing feature* of a deduplication file system. Note that a shared block may be referenced by a single or different files (or inodes).

To enable inline deduplication, LiveDFS introduces two primitives: *fingerprints* and *reference counts*. LiveDFS identifies each block by a fingerprint, which is a hash of the block content. If the fingerprints are collision-resistant cryptographic hash values (e.g., MD5, SHA-1), then it is practical to assume that two blocks having different contents will return two different fingerprints [23]. Since the fingerprint size is much smaller than the block size, we can easily identify duplicate blocks by checking if they have the same fingerprint. Also, to allow block modification and deletion, LiveDFS associates a reference count with each block, such that it keeps the number of block pointers that refer to the block. We increment the reference count of a block if a new block with the same content is written, and decrement it if a block is deleted.

We now describe how LiveDFS performs file system operations. Since LiveDFS is built on Ext3FS, most of its file system operations are the same as those of Ext3FS, except that LiveDFS integrates deduplication into the write operation. LiveDFS implements *copy-on-write* at the block level. If we update a block that is shared by multiple block pointers, then we either allocate a new block from the disk and write the updated content to the disk, or "*deduplicate*" the updated block with an existing block that has the same content. Note that the fingerprints and reference counts of the blocks are updated accordingly. If the reference count is decremented to zero, then LiveDFS deallocates the block as in Ext3FS.

To preserve the performance of file system operations when enabling deduplication, a critical issue is how to maintain the fingerprints and reference counts of all data blocks in a disk partition, such that we can search and update their values efficiently during deduplication. We elaborate this in Section 2.2.

2.2 Deduplication Design

To enable efficient search of fingerprints during deduplication, one option is to keep all fingerprints in main memory, but this significantly increases the memory cost. For example, if we use a block size of 4KB with 16-byte MD5 fingerprints, then 1TB of data will require 4GB of fingerprint space. As a result, we use an implementation that is similar to approaches in [24, 28] to manage the fingerprints. We introduce two components: a set of *fingerprint stores* and a *fingerprint filter*. We keep the full fingerprints in a set of fingerprint stores, which reside on disk. On the other hand, we use the fingerprint filter, which resides in main memory, to speed up the search of fingerprints. Specifically, we differentiate our work from [24, 28] by carefully placing fingerprints on disk based on the file system layout to further reduce the disk seek overhead during deduplication, as elaborated below.

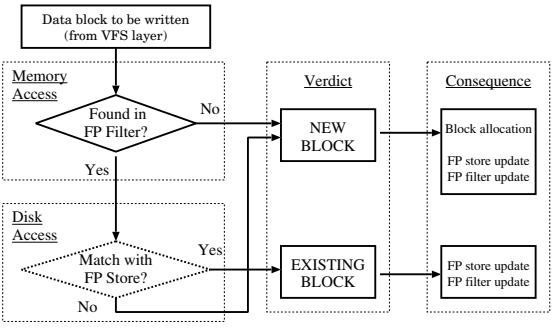


Fig. 2. Every block written to LiveDFS goes through the same decision process

Overview. The main challenge of LiveDFS is about the writing of data blocks. Figure 2 shows how LiveDFS handles that issue. Whenever a data block arrives at LiveDFS (and we name such a block an incoming block thereafter), its fingerprint is generated. We use the fingerprint of the incoming block to determine if the incoming block is unique.

The first checkpoint is the fingerprint filter ("`FP filter`" in Figure 2). The fingerprint filter is a memory-based filter that aims to determine if the incoming block can be deduplicated. If the incoming block is new to the file system, then it can be directly written to the disk. The design of the fingerprint filter will be detailed in later discussion.

Recalling that the fingerprint filter does not store any complete fingerprints, so the next step is to access the corresponding fingerprint store ("`FP store`" in Figure 2) on disk in order to confirm if the incoming block can actually be deduplicated. If the target fingerprint store does not contain the fingerprint of the incoming block, then it implies that the fingerprint filter gives a false-positive result and that the incoming block is unique; otherwise, the block is not unique and can be deduplicated.

In the following, we first elaborate the design of a fingerprint store, followed by that of the fingerprint filter.

Fingerprint store. LiveDFS exploits *spatial locality* for storing fingerprints with respect to the disk layout of the file system. Our insight is that LiveDFS follows Ext3FS and organizes blocks into block groups, each keeping the metadata of the blocks within the same group. In LiveDFS, each block group is allocated a *fingerprint store*, which is an array of pairs of fingerprints and reference counts, such that each array entry is indexed by the block number (i.e., block address) of the respective disk block in the same block group. Thus, each block group in the disk partition has its corresponding fingerprint store. Figure 3 shows how LiveDFS deploys a fingerprint store in a block group. We place the fingerprint store at the front of the data block region in each block group. When LiveDFS writes a new block, we write the content to the disk and update the corresponding fingerprint and reference count for the block. A key observation is that all updates are localized within the same block group, so the disk seek overhead is minimized.

To quantify the storage overhead of a fingerprint store, we consider the case where each fingerprint is a 16-byte MD5 hash and each reference count is of 4 bytes. Note that in Ext3FS, the default block size is 4KB and default block group size is 128MB,

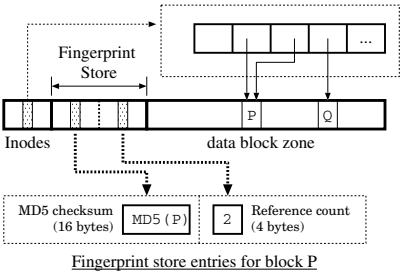


Fig. 3. Deployment of a fingerprint store in a block group

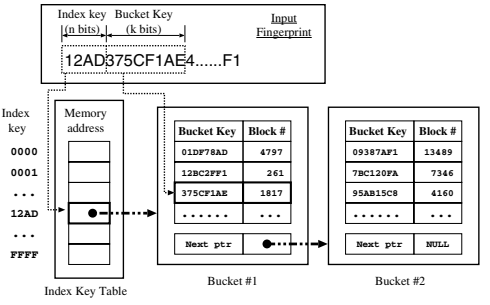


Fig. 4. Design of the fingerprint filter

so there are 32,768 blocks per block group. The fingerprint store will consume 655,360 bytes, or equivalently 160 blocks. This only accounts for 0.49% of the block group size. Hence, the storage overhead of a fingerprint store is limited.

Fingerprint filter. The *fingerprint filter* is an in-memory indexing structure that aims to speed up the search of fingerprints on disk. While there are many possible designs of indexing techniques (see Section 5), we explore a simple design that suffices for our requirements and is shown to work well according to our evaluation (see Section 4).

Figure 4 illustrates the design of the fingerprint filter. The fingerprint filter is a two-level filter. The first-level filter maps the first n prefix bits of a fingerprint called the *index key*, while the second-level filter maps the next k bits called the *bucket key*. We elaborate how n and k can be chosen in later discussion.

We initialize the fingerprint filter when LiveDFS is first mounted. The mounting procedure first allocates the *index key table*, which is an array of 2^n entries, each of which is a memory address that points to the head of a chain of buckets. Then LiveDFS will read through all fingerprint stores in different block groups, and retrieve the index key, the bucket key, and the block number associated with each disk block. We construct a *bucket entry* for each tuple (bucket key, block number) and store the entry in the correct bucket according to the index key. If a bucket is full of entries, then we create a new bucket, so a chain of buckets may be created. We sort the bucket entries in each bucket by the bucket keys, in order for us to efficiently search for a bucket entry using binary search. We emphasize that the initialization of the fingerprint filter is only a one-time process, which is performed when the file system is mounted. We evaluate the mount time using our experimental testbed (see Section 4). We find that the mount time is within 3 minutes for a 512GB harddisk partition that is fully filled with data. In general, the mount time varies linearly with the amount of data, and hence the number of fingerprints, being stored on disk.

We query the fingerprint filter whenever there is a new block to be written to the disk. If the fingerprint of the new block matches one of the "$n + k$"-bit prefixes stored in the filter, then the filter will return the corresponding block number. Note that there may be more than one fingerprint sharing the same "$n + k$" prefix bits, so the filter may return more than one block number. To eliminate false positives, LiveDFS will look up

the corresponding fingerprint store based on each returned block number, and verify if the new block to be written matches the full fingerprint in the fingerprint store. Note that LiveDFS updates the fingerprint filter every time a block is written, modified, or deleted.

Performance impact of parameters. We now elaborate how we set the parameters for the fingerprint filter, which depends on the index key length n and the bucket key length k. These parameters in turn determine the trade-offs of different metrics: (i) the total memory size of the filter, (ii) the false positive rate of each fingerprint filter lookup, and (iii) the bucket chain length. Instead of deriving the optimal choices of the parameters, we consider a special setting which performs well in practice based on our experiments (see Section 4).

In the following, we consider a 1TB partition with of block size 4KB (i.e., 2^{28} blocks in total). We employ the 16-byte MD5 hash algorithm to produce fingerprints. The system under our consideration adopts the 32-bit address space. We consider a special case where n = 19 and k = 24. Note that the following analysis can still apply to other parameter settings as well, e.g., 64-bit addressing.

We first evaluate the memory size of the fingerprint filter. For each data block, there is a 7-byte bucket entry, which stores a bucket key of k = 24 bits and a block number of 32 bits. Since there are a maximum of 2^{28} blocks, we need 1.792GB of memory for all bucket entries. Also, each index key entry is a 32-bit memory address that points to the head of a bucket chain, so we need $2^{19} \times 4$ = 2MB of memory for all index key entries. If we take into account internal fragmentation, then the fingerprint filter needs at most 2GB of memory, which is available for today's commodity configurations.

We emphasize that the memory footprint of LiveDFS is significantly less than those of ZFS [21] and SDFS [20], which assume 64GB and 8GB of memory per 1TB of data with block size 4KB, respectively. Although ZFS and SDFS use longer hashes (SHA-256 and Tiger, respectively) for fingerprints, the reason why they use significantly more memory is that they load the full fingerprints into memory. On the contrary, LiveDFS only loads the fingerprint prefixes into memory, and organizes the full fingerprints near their corresponding data blocks on disk so as to mitigate the disk seek overhead.

We next consider the false positive rate for each fingerprint filter lookup. Recall that there are 2^{28} data blocks. For a given fingerprint x, the probability ϵ that there exists another fingerprint (or data block) that has the same $n + k = 43$ prefix bits as x is: $1 - \left(1 - 2^{-43}\right)^{2^{28}-1} \approx 2^{-15}$. That is, on average every one out of 32,678 blocks will have a fingerprint mapped to more than one block number.

We also evaluate the bucket chain length. If the fingerprint value is uniformly distributed, then the average number of buckets associated with each index key is $2^{28}/2^{19}$ = 512 (for n = 19). Each bucket entry is of 7 bytes. If each bucket is stored as a memory page of size 4KB, then it can hold around 585 bucket entries, and the average bucket length is less than one bucket.

2.3 Prefetching of Fingerprint Store

Whenever LiveDFS writes a block, the fingerprint and the reference count of that block has to be updated. As a result, LiveDFS has to access the corresponding fingerprint store

on disk every time LiveDFS is writing a data block. We improve the performance of our deduplication design by extending the notion of spatial locality for caching fingerprints in memory. Our key observation is that a VM image file generally consists of a large stream of data blocks. If a data block is unique and cannot be deduplicated with existing blocks, then it will be written to the disk following the previously written block. Thus, the data blocks of the same block group are likely to be accessed at about the same time.

In order to further reduce the number of disk seeks, LiveDFS implements a *fingerprint prefetching mechanism*. When LiveDFS is about to access the fingerprint store of a block group, instead of accessing only the target block (which contains the target fingerprint), LiveDFS prefetches the entire fingerprint store of the corresponding block group and store it into the *page cache*, the disk cache of the Linux kernel. Therefore, subsequent writes in the same block group can directly update the fingerprint store in the page cache. The I/O scheduler of the Linux kernel will later flush the page cache into the disk. This further reduces the disk seeks involved. We point out that the consistency of the data content between the page cache and the disk is protected by the underlying journaling mechanism (see Section 2.4). Note that the idea is also used in [24, 28], except that we apply the idea in accordance with the Linux file system design.

The following calculations show that the overhead of our fingerprint prefetching mechanism is small. Suppose that LiveDFS uses MD5 hashes as the fingerprints of data blocks. Then a fingerprint store in a block group consumes 160 blocks (or 640KB space). Today an ordinary 7200-RPM hard disk typically has a data rate of 100MB/s, so it will consume only about 6-7ms for prefetching a whole fingerprint store into memory (i.e., the page cache). This time value is close to the average time of a random disk seek, which can be around 8-10ms [26].

2.4 Journaling

LiveDFS supports *journaling*, a feature that keeps track of file system transactions in a *journal* so that the file system can be recovered to a stable state when the machine fails, e.g., power outage. LiveDFS extends the journaling design in Ext3FS. In particular, we treat every write to a fingerprint store as the file system metadata and have the journal process modifications to the fingerprints and reference counts.

Figure 5 shows the pseudo-code of how LiveDFS updates a fingerprint store for a data block `P` to be written to the file system, while a similar set of procedures are taken when we delete a data block from the file system. Our goal is to update the fingerprint and the reference count associated with the block `P`. First, we obtain the handle that refers to the journal and perform the updates of metadata (e.g., indirect blocks, inodes) for `P` as in Ext3FS (Lines 1-2). If `P` is a new block that cannot be deduplicated with any existing disk block, then LiveDFS first loads the corresponding block that stores `P`'s fingerprint into memory (Line 4), and notify the journal that the fingerprint block is about to be updated via the function `ext3_journal_get_write_access()` (Line 5). After the fingerprint block is updated, we notify the journal that the modification is finished via the function `ext3_journal_dirty_metadata()` (Lines 6-7). Then, we update `P`'s reference count similarly (Lines 9-12). When LiveDFS releases the journal handle (Line 13), the journal will update the fingerprint store on disk *atomically*.

```
function LiveDFS Fingerprint Store Update Block
Input: data block P to be written to the file system
 1: handle = ext3_journal_start()
 2: Perform metadata writes via the journal handle
 3: if P cannot be deduplicated with an existing block then
 4:     Load a fingerprint block fp into memory
 5:     ext3_journal_get_write_access(handle, fp)
 6:     Update fp with the fingerprint of P
 7:     ext3_journal_dirty_metadata(handle, fp)
 8: end if
 9: Load P's reference count cp into memory
10: ext3_journal_get_write_access(handle, cp)
11: Increment the reference count cp by one
12: ext3_journal_dirty_metadata(handle, cp)
13: ext3_journal_stop(handle)
```

Fig. 5. Pseudo-code of how LiveDFS updates the fingerprint store through the journaling system

Not only can the journal improve the file system reliability, but it can also enhance the write performance. The journal defers the disk updates of each write request and combines multiple disk writes in batch. This reduces the disk seeks and improve the write performance. We demonstrate this improvement in Section 4.

3 LiveDFS Implementation and Deployment

LiveDFS is a kernel-space file system running atop Linux. We implemented LiveDFS as a kernel driver module for the Linux kernel 2.6.32, and it can be loaded to the kernel *without* requiring any modification or recompilation of the kernel source code. The deduplication logic is implemented by extending the virtual file system (VFS) address space operations. In particular, we perform fingerprint computation and determine if a block can be deduplicated in the function `writepage()` (in Linux source tree: `fs/ext3/inode.c`), which is called by a kernel thread and will flush dirty pages to the disk.

Since LiveDFS is POSIX-compliant, it can be seamlessly integrated into an open-source cloud platform that runs atop Linux. In this work, we integrate LiveDFS into OpenStack [22], an open-source cloud platform backed by Rackspace and NASA, such that LiveDFS serves as a storage layer for hosting VM images with deduplication. We point out that Eucalyptus [18] has a similar architecture as OpenStack, so we expect that the deployment of LiveDFS in Eucalyptus follows a similar approach.

OpenStack overview. OpenStack is built on three sub-projects *Compute* (named *Nova*), *Object Storage* (named *Swift*), and *Image Service* (named *Glance*). Figure 6 shows a simplified view of an OpenStack cloud, which consists of Nova and Glance only. Nova defines an architecture that uses several controller services that coordinate the VM instances running on different Compute nodes. Glance is a VM image management

system that is responsible for registering, searching, and retrieving VM images. It provides APIs for accessing a storage backend, which could be Object Storage (Swift), Amazon S3, or a local server on which Glance is deployed. Note that OpenStack uses the `euca2ools` command-line tool provided by Eucalyptus to add and delete VM images.

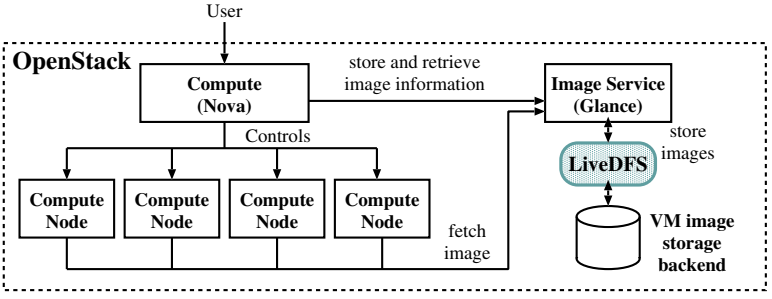


Fig. 6. LiveDFS deployment in an OpenStack cloud

LiveDFS deployment. Figure 6 shows how LiveDFS is deployed in an OpenStack cloud. LiveDFS serves as a storage layer between Glance and the VM image storage backend. Administrators can upload VM images through Glance, and the images will be stored in the LiveDFS partition. When a user wants to start a VM instance, the cloud controller service of Nova will assign the VM instance to run on one of the Compute nodes based on the current resource usage. Then the assigned Compute node will fetch the VM image from Glance, which then retrieves the VM image via LiveDFS.

4 Experiments

In this section, we empirically evaluate our LiveDFS prototype. We first measure the I/O throughput performance of LiveDFS as a disk-based file system. We then evaluate the deployment of LiveDFS in OpenStack-based cloud platform. We justify the performance overhead of LiveDFS that we observe. We compare LiveDFS with Ext3FS, which does not support deduplication.

4.1 I/O Throughput

We measure the file system performance of different I/O operations using synthetic workload based on LiveDFS and Ext3FS. In LiveDFS, we assume that the index key length n is 19 bits and the bucket key length k is 24 bits (see Section 2.2). Note that LiveDFS is built on different design components, including (i) spatial locality, in which we allocate fingerprint stores in different block groups (see Section 2.2), (ii) prefetching of a fingerprint store (see Section 2.3), and (iii) journaling (see Section 2.4). We evaluate different LiveDFS variants that include different combinations of the design components, as shown in Table 1, to see the performance impact of each component.

When spatial locality is disabled, we simply place all fingerprint stores at the end of the disk partition; when prefetching is disabled, we bypass the step of prefetching a fingerprint store into the page cache; when journaling is disabled, we use alternative calls to directly write fingerprints and reference counts to the fingerprint stores on disk.

Table 1. Different LiveDFS variants evaluated in Section 4.1 ($\surd$ = enabled, $\times$ = disabled)

	Spatial locality	Prefetching	Journaling
LiveDFS-J	$\times$	$\times$	$\surd$
LiveDFS-S	$\surd$	$\times$	$\times$
LiveDFS-SJ	$\surd$	$\times$	$\surd$
LiveDFS-all	$\surd$	$\surd$	$\surd$

Experimental testbed. Our experiments are conducted on a Dell Optiplex 980 machine with an Intel Core i5 760 CPU at 2.8GHz and 8GB DDR-III RAM. We equip the machine with two harddisks: a 1TB harddisk of Western Digital WD1002FAEX 7200RPM SATA for our benchmarking, and a 250GB harddisk for hosting our benchmarking tools and the operating system. Our operating system is Ubuntu 10.04.2 server 64-bit edition with Linux kernel 2.6.32.

Evaluation methodology. We use Linux basic system calls `read()` and `write()` to measure the I/O performance. Each experimental result is averaged over 10 runs. In each run, we use the system call `gettimeofday()` to obtain the duration of an operation, and then compute the throughput. At the beginning of each run, we clear the kernel page cache using the command `echo 3 > /proc/sys/vm/drop_caches` so that we can accurately evaluate the performance due to disk accesses.

Experiment A1: Sequential write. We first evaluate the sequential write performance of LiveDFS by writing a 16GB file with all unique blocks (i.e., all blocks cannot be deduplicated with others). The file size is larger than the 8GB RAM in our test machine, so that not all requests are kept in the kernel buffer cache.

Figure 7 shows the throughput of different LiveDFS variants and Ext3FS. First, considering LiveDFS-J and LiveDFS-SJ, we observe that LiveDFS-SJ has a higher throughput than LiveDFS-J by 16.1MB/s (or 37%). Thus, spatial locality by itself can improve the throughput, by putting the fingerprint stores close to their blocks on disk.

We note that LiveDFS-S (without journaling) has the lowest throughput among all variants. Specifically, LiveDFS-SJ increases the throughput of LiveDFS-S by 34.6MB/s (or 78.6%). Since journaling combines write requests and flushes them to the disk in batch (see Section 2.4), journaling can effectively minimize the disk accesses in addition to providing robustness against system crashes.

We observe that LiveDFS-all further improves the throughput of LiveDFS-SJ via prefetching (by 13.8MB/s, or 18%). Now, comparing LiveDFS-all and Ext3FS, we observe that LiveDFS-all is slightly less than Ext3FS's throughput by 3.7MB/s (or 3.8%), mainly due to the increase in the block group accesses. Because we introduce a fingerprint store to each block group, LiveDFS has fewer data blocks per block group than Ext3FS. However, we show that each design component of LiveDFS can reduce disk accesses and increase the throughput of LiveDFS close to that of Ext3FS.

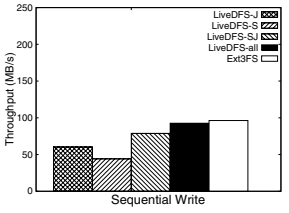


Fig. 7. Throughput of sequential write

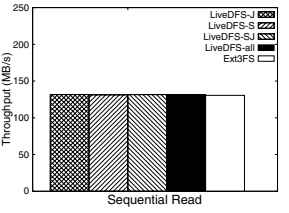


Fig. 8. Throughput of sequential read

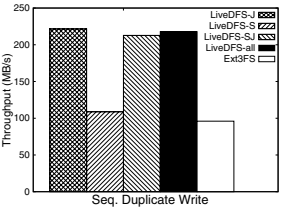


Fig. 9. Throughput of sequential duplicate write

Experiment A2: Sequential read. In this experiment, we evaluate the sequential read performance of LiveDFS by reading the stored 16GB file created in Experiment A1. Figure 8 shows the results. LiveDFS and Ext3FS. We observe that all LiveDFS variants have almost the same throughput as Ext3FS, mainly because LiveDFS uses the same execution path as that of Ext3FS for reading data blocks.

Experiment A3: Sequential duplicated write. In this experiment, we write another 16GB file that has the identical content to the one in Experiment A1. Figure 9 shows the results. We observe that all LiveDFS variants (except LiveDFS-S without journaling) significantly boost the throughput of Ext3FS by around 120MB/s. The reason is that LiveDFS only needs to read the fingerprints of data blocks and update the reference counts, without re-writing the same data blocks (with larger size) to the disk. We note that the write-combining feature of journaling plays a significant role in boosting the throughput of LiveDFS, as LiveDFS-S does not achieve the same improvement.

Experiment A4: Crash recovery with journaling. In this experiment, we consider how deduplication affects the time needed for a file system to recover from a crash using journaling. We set the journal commit interval to be five seconds when the file system is first mounted. We then write a 16GB file with unique blocks sequentially into LiveDFS, and unplug the power cable in the middle of writing. The file system is therefore inconsistent. After rebooting the machine, we measure the time needed for our file system check tool `fsck`, which is modified from the Ext2FS utility `e2fsprogs` to recover the file system.

We observe that LiveDFS ensures the file system correctness using journaling. However, it generally uses a longer recovery time than Ext3FS. On average (over 10 runs), LiveDFS uses 14.94s to recover, while Ext3FS uses less than 1s. The main reason is that LiveDFS needs to ensure that the fingerprints in the fingerprint store match the new data blocks being written since the last journal commit interval. Such additional fingerprint checks introduce overhead. Since system crashes are infrequent, we expect that the recovery time is acceptable, as long as the file system correctness is preserved.

4.2 OpenStack Deployment

We now evaluate LiveDFS when it is deployed in an OpenStack-based cloud. Our goal is to justify the practicality of deploying LiveDFS in a real-life open-source cloud for

VM image storage with deduplication. Specifically, we aim to confirm that LiveDFS achieves the expected storage savings as observed in prior studies [13, 11], while achieving reasonable I/O performance of accessing VM images.

System configuration. Our experiments are conducted in an OpenStack cloud platform consisting of three machines: a Nova cloud controller, a Glance server, and a Compute node. The Nova cloud controller is equipped with an Intel Core 2 Duo E7400 2.8GHz CPU, while both the Glance server and the compute node are equipped with an Intel Core 2 Quad Q9400 2.66GHz CPU. All machines are equipped with 4GB DDR-II RAM, as well as two harddisks: a 1TB 7200RPM harddisk for storing VM images and a 250GB harddisk for hosting the operating system and required software. All machines use Ubuntu 10.04.2 server 64-bit edition as the operating system. Furthermore, all three machines are inter-connected by a Gigabit Ethernet switch, so we expect that the network transmission overhead has limited performance impact on our experiments.

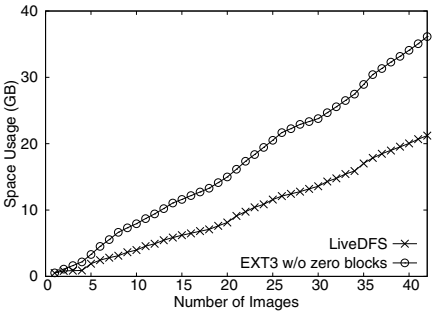
All VM images are locally stored in the Glance server. We deploy either LiveDFS or Ext3FS within the Glance server, which can then access VM images via the deployed file system using standard Linux file system calls. We also deploy Kernel-based Virtual Machine (KVM) as the default hypervisor in the compute node. By default, we assume that LiveDFS enables all design components (i.e., spatial locality, prefetching, and journaling).

Our cloud testbed consists of only one Compute node, assuming that in most situations there is at most one Compute node that retrieves a VM image at a time. We also evaluate the scenario when a Compute node retrieves multiple VM images simultaneously (see Experiment B3).

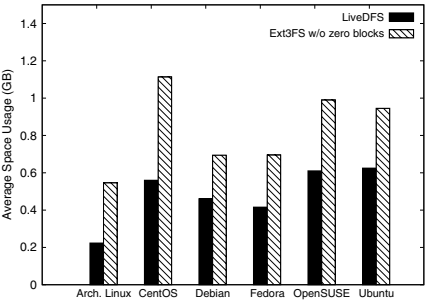
Dataset. We use deployable VM images to drive our experiments. We have created 42 VM images in Amazon Machine Image (AMI) format. Table 2 lists all the VM images. The operating systems of the VM images include ArchLinux, CentOS, Debian, Fedora, OpenSUSE, and Ubuntu. We prepare images of both x86 and x64 architectures for each distribution, using the recommended configuration for a basic server. Networked installation is chosen so as to ensure that all installed software packages are up-to-date. Each VM image is configured to have size 2GB. Finally, each VM image is created as a single monolithic flat file.

Table 2. The collection of virtual machine images involved in the experiments

Distribution	Version (x86 & x64)	Total
ArchLinux	2009.08, 2010.05	4
CentOS	5.5, 5.6	4
Debian	5.0.8, 6.0.1	4
Fedora	11, 12, 13, 14	8
OpenSUSE	11.1, 11.2, 11.3, 11.4	8
Ubuntu	6.06, 8.04, 9.04, 9.10, 10.04, 10.10, 11.04	14



(a) Cumulative space usage.



(b) Average space usage of a VM image for different distributions.

Fig. 10. Space usage of VM images (excluding zero-filled blocks)

Experiment B1: Storage efficiency. We first validate that LiveDFS can save space for storing VM images. Note that each VM image typically has a large number of zero-filed blocks [11]. One main source of zero-filled blocks, according to our created VM images, is due to the unused space of the VM. To reflect the true saving achieved by deduplication, we exclude counting the zero-filled blocks in our evaluation.

Figure 10(a) shows the cumulative space usage of storing the VM images using LiveDFS and Ext3FS. Overall, LiveDFS saves at least 40% of space over Ext3FS (for non-zero-filled blocks). If we count the zero-filled blocks as well, then LiveDFS still uses around 21GB of space (as all zero-filled blocks can be denoted by a single block), while Ext3FS consumes 84GB of space for the 42 2-GB VM images. In this case, we can even achieve 75% of saving.

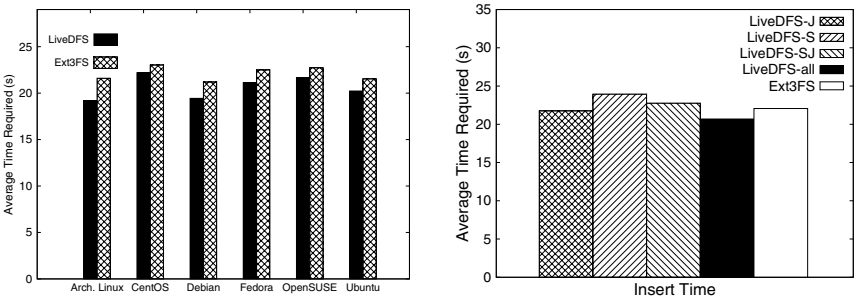
Figure 10(b) shows the average space usage of a VM image for each Linux distribution. The space savings range from 33% to 60%. It shows that different versions of VM images of the same Linux distribution have a high proportion of identical data blocks that can be deduplicated. Therefore, our LiveDFS implementation conforms to the observations that deduplication is effective in improving the storage efficiency of VM images [13,11]. We do not further investigate the deduplication effectiveness of VM images, which has been well studied in [13,11].

Experiment B2: Time for inserting VM images. In this experiment, we evaluate the time needed for inserting VM images into our Glance server. We execute the commands `euca-bundle-image`, `euca-upload-bundle`, and `euca-register`[1] to insert the VM images from the cloud controller to the Glance server (over the Gigabit Ethernet switch). We repeat the test five times and obtain the average. Our goal is to measure the practical performance of writing VM images using LiveDFS.

Figure 11(a) shows the average insert time for individual distributions (over five runs). Overall, LiveDFS consumes less time than Ext3FS in inserting VM images. The reason is that LiveDFS does not write the blocks that can be deduplicated to the disk, but instead it only updates the smaller-size reference counts. Figure 11(b) shows the

[1] Those `euca-*` commands come with `euca2ools`, the command-line tool of Eucalyptus for VM image management.

average insert time for all 42 VM images using different LiveDFS variants as defined in Section 4.1. Although this time the differences among the different LiveDFS variants are not as significant as seen in Section 4.1, we observe that enabling all design components (i.e., LiveDFS-all) still gives the least insert time.



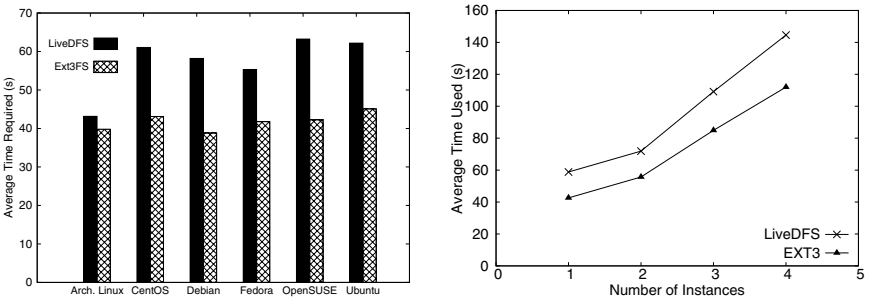
(a) Individual distribution. (b) Different LiveDFS variants.

Fig. 11. Average time required for inserting a VM image to the Glance server

Experiment B3: Time for VM startup. We now evaluate the time needed to launch a single or multiple VM instances. We assume that all VM images have been inserted into the file system. We then execute the `euca-run-instances` command in the Nova cloud controller to start a VM instance in the Compute node, which fetches the corresponding VM image from the Glance server. We measure the time from the command being issued until the time when the `euca-run-instances` command invokes the KVM hypervisor (i.e., when the VM starts running). Our goal is to measure the practical performance of reading VM images using LiveDFS.

Figure 12(a) shows the time needed to launch a single VM instance for different distributions in the Compute node. We observe that LiveDFS has lower throughput performance than Ext3FS. The main reason is that deduplication introduces *fragmentation* [24]. That is, in deduplication, some blocks of a file may be deduplicated with the blocks of a different file, so the actual block allocation of a file on disk is no longer in the sequential order as seen in the ordinary file system without deduplication. Fragmentation is an inherent problem in deduplication, as it remains an open issue to be solved [24]. To see how fragmentation affects the practical performance, we note that in LiveDFS, its increase in VM startup time ranges from 3s to 21s (or 8% to 50%) for a VM image of size 2GB. Currently, the cloud nodes are connected over a Gigabit Ethernet switch. We expect that the VM startup penalty will become less significant if we deploy the cloud in a network with less available bandwidth (e.g., the cloud nodes are connected by multiple switches that are shared by many users), as the network transmission overhead will dominate in such a setting.

We now consider the case when we launch multiple VM instances in parallel. In the Compute node, we issue multiple VM startup commands simultaneously, and measure the time for all VM instances to start running. Figure 12(b) shows the time required for starting one to four VM instances in the Compute node. The overhead of LiveDFS remains consistent (at around 30%), regardless of the number of VM instances being launched. The observation conforms to that of launching a single VM instance.



(a) Startup time for a VM instance for different distributions.

(b) Startup time for multiple VM instances using the Ubuntu distribution.

Fig. 12. Average time for VM startup

5 Related Work

Existing deduplication techniques are mainly designed for backup systems. On the other hand, LiveDFS applies deduplication in a different design space and is designed for VM image storage in a cloud platform. Also, LiveDFS seeks to be POSIX-compliant, so its implementation details take into account the Linux file system layout and are different from existing deduplication systems. In this section, we review the related work in deduplication in more detail.

Backup systems. Venti [23] is the first work of content addressable storage (CAS) for data backup. Foundation [24] is another CAS system built upon Venti, and improves Venti with the new compare-by-value mode. It uses the Bloom filter as an in-memory indexing structure to identify new or existing fingerprints. LiveDFS uses the fingerprint filter as an in-memory indexing structure. However, the fingerprint filter not only identifies the existence of fingerprints as in the Bloom filter, but it also specifies where the fingerprint is stored on disk. Data Domain [28] also uses the Bloom filter [4] for in-memory indexing, and uses locality preserved caching to reduce random I/Os for duplicated data. Sparse Indexing [14] trades deduplication opportunities for reduced memory usage in backup systems by only deduplicating data with a few of the most similar previous copies in different backup streams. Bimodal Content Defined Chunking [12] reduces the memory overhead of chunk indexing by using chunks with different granularities in different regions of a backup. Note that the above systems do not consider the scenario where data can be modified or deleted, while LiveDFS addresses this by associating a reference count with each data block in its deduplication design.

Usage of flash memory. Dedupv1 [15] and ChunkStash [8] use flash memory and solid state drives (SSDs) to relieve the memory constraints of deduplication. They exploit the fast random I/O feature of flash memory by putting the fingerprints into the flash rather than in main memory. They show that they achieve competent I/O performance while requiring a small memory capacity. On the other hand, LiveDFS targets a commodity server that is not necessarily equipped with SSDs.

Scalable storage. Extreme Binning [3], HydraFS [27] and DeDe [7] are scalable storage systems that support data deduplication. Extreme Binning exploits *file similarity* rather than chunk locality. HydraFS is a file-system built atop HYDRAstor [9]. DeDe is a storage system that performs *out-of-order* deduplication. Their deployment platforms are based on a distributed environment, while LiveDFS is designed to be deployed on a single commodity server.

Deduplication-enabled file systems. ZFS by Sun Microsystems[21] and SDFS by Opendedup [20] are file systems supporting inline deduplication. However, as stated in Section 2.2, they need a large memory capacity for enabling deduplication, mainly because they assume that the entire set of fingerprints is loaded into memory. In terms of deployment, ZFS and SDFS are mainly designed for enterprise-scale systems, while LiveDFS is designed as a kernel-space file system for commodity servers. Btrfs [5] is an open-source file system developed by Oracle that supports deduplication. While Btrfs is a Linux kernel module like LiveDFS, it only supports offline deduplication [19] instead of inline at the time of this writing.

VM image storage. It has been shown [13,11] that deduplication can significantly save the storage space for VM images. However, there remain open issues of deploying deduplication in a VM storage system. Nath *et al.* [17] evaluate a deployed CAS system for storing VM images. They mainly focus on storage efficiency and network load, but do not evaluate the read/write throughput of accessing VM images. Liguori *et al.* [13] deploy a CAS system based on Venti [23] for storing VM images, but its read/write throughput is limited by the Venti implementation. Lithium [10] is a cloud-based VM image storage system that aims for fault tolerance, but it does not consider deduplication. To our knowledge, LiveDFS is the first practical system that deploys deduplication for VM image storage in a real cloud platform.

6 Conclusions and Future Work

We propose LiveDFS, a live deduplication file-system that is designed for VM image storage in an open-source cloud with commodity configurations. LiveDFS respects the file system design layout in Linux and allows general I/O operations such as read, write, modify, and delete, while enabling inline deduplication. To support inline deduplication, LiveDFS exploits spatial locality to reduce the disk access overhead for looking up fingerprints that are stored on disk. It also supports journaling for crash recovery. LiveDFS is implemented as a Linux kernel driver module that can be deployed without the need of modifying the kernel source. We integrate LiveDFS into a cloud platform based on OpenStack and evaluate the deployment. We show that LiveDFS saves at least 40% of storage space for different distributions of VM images, while its performance overhead in read/write throughput is minimal overall. Our work demonstrates the feasibility of deploying deduplication into VM image storage in an open-source cloud.

In this work, we mainly focus on deduplication on a single storage partition. Since a cloud platform is typically a distributed system, we plan to extend LiveDFS in a distributed setting (e.g., see [16]). One challenging issue is to balance the trade-off between storage efficiency and fault tolerance. On one hand, deduplication reduces the

storage space by removing redundant data copies; on the other hand, it sacrifices fault tolerance with the elimination of redundancy. We pose this issue as future work.

The source code of LiveDFS is published for academic use at: `http://ansrlab.cse.cuhk.edu.hk/software/livedfs`.

Acknowledgments. This work was supported in part by grant ITS/297/09 from the Innovation and Technology Fund of the HKSAR.

References

1. Amazon EC2, `http://aws.amazon.com/ec2`
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A View of Cloud Computing. Comm. of the ACM 53(4), 50–58 (2010)
3. Bhagwat, D., Eshghi, K., Long, D., Lillibridge, M.: Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In: Proc. IEEE MASCOTS, pp. 1–9. IEEE (2009)
4. Bloom, B.H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. Communications of the ACM (1970)
5. Btrfs, `http://btrfs.wiki.kernel.org`
6. Cao, M., Tso, T., Pulavarty, B., Bhattacharya, S., Dilger, A., Tomas, A.: State of the art: Where we are with the ext3 filesystem. In: Proc. of the Ottawa Linux Symposium, OLS (2005)
7. Clements, A., Ahmad, I., Vilayannur, M., Li, J.: Decentralized deduplication in SAN cluster file systems. In: Proc. USENIX ATC (2009)
8. Debnath, B., Sengupta, S., Li, J.: ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In: Proc. USENIX ATC (2010)
9. Dubnicki, C., Gryz, L., Heldt, L., Kaczmarczyk, M., Kilian, W., Strzelczak, P., Szczepkowski, J., Ungureanu, C., Welnicki, M.: Hydrastor: A scalable secondary storage. In: Proc. USENIX FAST (2009)
10. Hansen, J.G., Jul, E.: Lithium: Virtual Machine Storage for the Cloud. In: Proc. of ACM SOCC (2010)
11. Jin, K., Miller, E.L.: The effectiveness of deduplication on virtual machine disk images. In: Proc. ACM SYSTOR (2009)
12. Kruus, E., Ungureanu, C., Dubnicki, C.: Bimodal content defined chunking for backup streams. In: Proc. USENIX FAST, page 18. USENIX Association (2010)
13. Liguori, A., Van Hensbergen, E.: Experiences with Content Addressable Storage and Virtual Disks. In: WIOV 2008 (2008)
14. Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., Camble, P.: Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In: Proc. USENIX FAST (2009)
15. Meister, D., Brinkmann, A.: Dedupv1: Improving deduplication throughput using solid state drives (SSD). In: Proc. IEEE MSST (2010)
16. Muthitacharoen, A., Chen, B., Mazières, D.: A low-bandwidth network file system. In: Proc. of ACM SOSP (2001)
17. Nath, P., Kozuch, M.A., O'Hallaron, D.R., Harkes, J., Satyanarayanan, M., Tolia, N., Toups, M.: Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines. In: Proc. USENIX ATC (2006)

18. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The Eucalyptus Open-source Cloud Computing System. In: Proc. of IEEE CCGrid (2009)
19. Offline Deduplication for Btrfs, http://www.spinics.net/lists/linux-btrfs/msg07818.html
20. OpenDedup. NBU for vSphere (December 2010), http://code.google.com/p/opendedup/downloads/detail?name=SDFScture.pdf
21. OpenSolaris. ZFS Dedup FAQ (Community Group zfs.dedup) - XWiki (December 2010), http://hub.opensolaris.org/bin/view/Community+Group+zfs/dedup
22. OpenStack, http://www.openstack.org
23. Quinlan, S., Dorward, S.: Venti: a new approach to archival storage. In: Proc. USENIX FAST (2002)
24. Rhea, S., Cox, R., Pesterev, A.: Fast, inexpensive content-addressed storage in foundation. In: Proc. USENIX ATC (2008)
25. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In: Proc. of ACM CCS (2009)
26. Seagate. 7200-RPM Drive Specification Comparison, http://www.seagate.com/docs/pdf/whitepaper/mb578_7200_drive_specification_comparison.pdf
27. Ungureanu, C., Atkin, B., Aranya, A., Gokhale, S., Rago, S., Calkowski, G., Dubnicki, C., Bohra, A.: HydraFS: a High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In: Proc. of USENIX FAST (2010)
28. Zhu, B., Li, K., Patterson, H.: Avoiding the disk bottleneck in the data domain deduplication file system. In: Proc. USENIX FAST (2008)

Scalable Load Balancing in Cluster Storage Systems

Gae-won You[1], Seung-won Hwang[1], and Navendu Jain[2]

[1] Pohang University of Science and Technology, Republic of Korea
[2] Microsoft Research Redmond, United States
{gwyou,swhwang}@postech.edu, navendu@microsoft.com

Abstract. Enterprise and cloud data centers are comprised of tens of thousands of servers providing petabytes of storage to a large number of users and applications. At such a scale, these storage systems face two key challenges: (a) hot-spots due to the dynamic popularity of stored objects and (b) high reconfiguration costs of data migration due to bandwidth oversubscription in the data center network. Existing storage solutions, however, are unsuitable to address these challenges because of the large number of servers and data objects. This paper describes the design, implementation, and evaluation of Ursa, which scales to a large number of storage nodes and objects and aims to minimize latency and bandwidth costs during system reconfiguration. Toward this goal, Ursa formulates an optimization problem that selects a subset of objects from hot-spot servers and performs *topology-aware* migration to minimize reconfiguration costs. As exact optimization is computationally expensive, we devise scalable approximation techniques for node selection and efficient divide-and-conquer computation. Our evaluation shows Ursa achieves cost-effective load balancing while scaling to large systems and is time-responsive in computing placement decisions, *e.g.*, about two minutes for 10K nodes and 10M objects.

Keywords: Load balancing, storage, optimization, linear programming.

1 Introduction

This paper describes a scalable data management middleware system Ursa that aims to improve load balancing for cloud storage services services in the spirit of web email data stores (*e.g.*, Hotmail or Yahoo! Mail) over utility storage systems (*e.g.*, Amazon S3 [1] and Windows Azure [2]).

As cloud services continue to grow rapidly, large-scale cloud storage systems are being built to serve data to billions of users around the globe. The primary goal for these systems is to provide scalable, good performance, and high-availability data stores while minimizing operational expenses, particularly the bandwidth cost. However, diverse I/O workloads can cause significant data imbalance across servers resulting in hot-spots [14]. Fig. 1 shows the CPU utilization across clusters in Hotmail. From this figure, we can observe that I/O workloads are unbalanced over clusters and they are skewed to a few object partitions resulting in "hot nodes," causing high delays to end users. As a result, these services typically shuffle terabytes of data per day to balance load across clusters [22].

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 101–122, 2011.

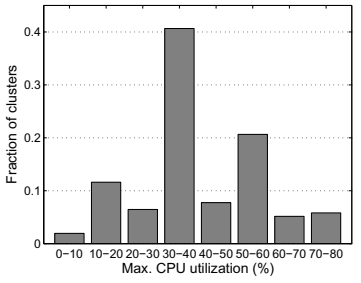


Fig. 1. CPU utilization in Hotmail clusters. CPU utilization is highly skewed to a few object partitions.

The same challenge has been raised and actively discussed in the context of building database on clouds [5, 6, 7, 17, 20, 21]. However, all these works either do not address dynamic load reconfiguration or assume the source and target nodes of dynamic migration are known by an oracle. In contrast, we aim at achieving scalable and dynamic reconfiguration at the storage layer by identifying cost-optimal source-target pairs. Toward this goal, we highlight the following two key challenges for our system:

1. Be scalable to large numbers of nodes and objects: The system should scale to a large number of nodes storing data from participating users and applications. Cloud scale systems today have tens of thousands of nodes and billions of objects corresponding to petabytes of disk space, and these numbers will increase over time. Similarly, the system should support a broad range of applications which may track per-user data (*e.g.*, mail folders) or per-object access patterns (*e.g.*, video content), or perform analytical processing like MapReduce [8] on large datasets (*e.g.*, scientific traces and search index generation from crawled web pages).

2. Minimize reconfiguration costs: The system should aim to minimize the reconfiguration costs in terms of both bandwidth and latency. As workloads change over time, load balancing incurs bandwidth overhead to monitor the load of individual objects and servers to find hot-spots as well as to migrate data to alleviate hot-spots. Similarly, data movement risks a long reconfiguration time window, typically proportional to the migrated data volume, during which reads may incur a high delay and writes may need to be blocked to ensure consistency. Finally, reconfigurations may interfere with foreground network traffic risking performance degradation of running applications.

To our knowledge, all prior techniques have aimed to solve these challenges individually. To address load imbalance, many techniques perform dynamic placement of individual data objects [3, 25] or distribute objects randomly across the cluster (*e.g.*, based on hashing [18]). However, to adaptively re-distribute objects, we need to know load patterns for billions of objects. Optimizing reconfiguration costs for these patterns calls for offline solvers [11, 16] (*e.g.*, knapsack or linear programming based) to make migration decisions. However, as such optimization is inherently expensive, these approaches are suitable at small scale and less effective when systems grow to a large scale. Meanwhile, approaches trading effectiveness to achieve scalability, for instance, by using a greedy simulated annealing or an evolutionary algorithm [16], suffer from high reconfiguration costs; we discuss these approaches in detail in related work.

Our goal is to show that a simple and adaptive framework can significantly reduce reconfiguration costs for industry-scale load balancing in cloud storage systems. While designing Ursa, we keep it implementable with mechanisms in existing storage systems–meaning our design should avoid complexity, can be evaluated on physical hardware, and can be deployed to our data centers today.

Our approach Ursa is designed to address the challenges listed above, using the following two key ideas:

1. Workload-driven optimization approach: First, to minimize the reconfiguration costs, Ursa formulates the problem as a workload-aware integer linear programming (ILP) problem whose goal is to eliminate hot-spots while minimizing the data movement cost. Based on our analysis of production workloads (Section 2.1), Ursa leverages the fact that the number of hot-spots is small compared to system size (*i.e.*, power law distribution) and hence optimizes load balancing for only the hot-spots rather than uniformly distributing the entire load across all servers.

2. Scalable approximation techniques: Second, to achieve scalability, Ursa applies an efficient divide-and-conquer technique to optimize for a subset of source/target nodes based on the workload and network topology. To further reduce migration costs under high bandwidth oversubscription prevalent in data center networks [13], Ursa migrates data in a topology-aware manner, *e.g.*, first finds placement within the same rack, then a neighboring rack, and so on. When a neighboring set of a hot-spot server overlaps with another, Ursa performs joint data placement of objects from hot-spots in the combined neighborhood set.

We have developed a prototype of Ursa and evaluated it on traces from Hotmail. Our evaluation shows that, compared to state-of-the-art heuristics [16], Ursa achieves scalable and cost-effective load balancing and is time-responsive in computing placement decisions (*e.g.*, about two minutes for 10K nodes and 10M objects).

2 Preliminaries

In Section 2.1, we first explain our target application scenarios, based on which we formally discuss our problem in Section 2.2.

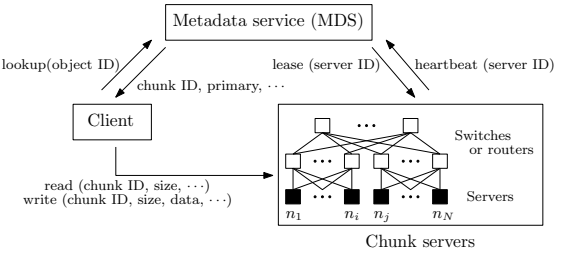


Fig. 2. Overall architecture; the data center network organized as a spanning tree topology [13]

2.1 Motivation

Our target system model for Ursa is that of a replicated, cluster-based object store similar to current systems such as GFS [12] and Azure blob store [2]. The object store is comprised of tens of thousands of storage nodes, typically commodity PCs, which provide read/write accesses to data. The nodes are inter-connected in a data center network which is organized as a spanning tree topology [13]. As a result, network distance

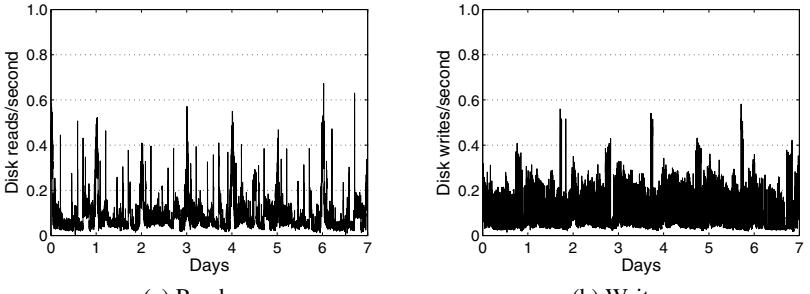
between nodes can vary significantly *e.g.*, two nodes in the same rack or cluster have higher bandwidth connectivity than those in different ones, which affects migration costs between nodes. Fig. 2 illustrates the overall architecture.

Data is stored in units of chunks (or partitions). A chunk is of some fixed size, *e.g.*, 64KB in Google's Bigtable [4] and 64MB in GFS [12]. Each chunk has l replicas (1 primary and $l-1$ secondary replicas) to enable fault tolerance; a typical value of l is 3. At any time a storage server will be the primary for some of the chunks stored on it. For fault tolerance, each replica of a chunk is typically placed in a separate fault domain, *e.g.*, a node or a rack.

All client requests are first sent to a metadata service which maps a data object to its constituent chunks, and each chunk to its current primary. The metadata service periodically polls each server to track availability and uses leases to maintain read/write consistency. A read request is sent by the primary to the local replica. For a write request, the primary first determines request ordering, then sends the request to all replicas, and finally acknowledges 'accept' to the client when each of these steps is completed.

Load balancing is achieved by uniformly spreading chunks over storage servers and choosing primaries for each chunk randomly from the available replicas of that chunk. However, in balancing load under dynamic workloads, these storage systems face several challenges described using a case-study of a large cloud email service.

Case-study on Hotmail: Hotmail is a cloud service with millions of users and is comprised of tens of thousands of nodes storing petabytes of data (*e.g.*, emails and attachments). In Hotmail, most writes are message writes. When a message arrives, it is simultaneously written to multiple replicas. Hotmail is read-heavy, which is the source of disk latency, and its workload has a heavy tailed distribution with large size email attachments which result in significant disk I/O load.



(a) Reads (b) Writes

Fig. 3. Normalized Hotmail weekly loads

We observed the (normalized) volume of read and write requests for a typical server for a week, as reported in Fig. 3. First, we observed a significant variation in the request volume by about 10x. Second, the requests are imbalanced across partitions and correspondingly across storage nodes hosting them. For instance, recall from Fig. 1 that CPU utilization is highly skewed to a few partitions, which become hot-spots and bottlenecks

for the entire system. We leverage these observations in designing our load balancing algorithm by spreading load away from hot-spots and minimizing reconfiguration costs to migrate data under dynamic load patterns.

2.2 Problem Statement

Based on these observations, we formally define the problem of load balancing. In particular, we show how to integrate load balancing in a cloud storage system.

We first define some notations to be used in this paper. Let $\{n_1, \cdots, n_N\}$ be a set of nodes, where N is the number of nodes and each node n_i contains replicas r_j's of some partitions. Each replica r_j has a workload L_j according to the request on the replica and each node n_i has the sum of loads of all replicas that it contains, *i.e.*, $L_{*i} = \sum_{r_j \in n_i} L_j$.

Fig. 4 evaluates the response time for a varying volume of requests in a VM on Windows Azure. We assume various block sizes, *i.e.*, 8-64MB, as the unit of each read or write request. We measure the response time of the 99th percentile among 1,000 requests. As expected, we observe from Fig. 4 results that the response time increases significantly when the request load exceeds a threshold, *e.g.*, more than 10 seconds at 50 reads/sec in read requests and 20 writes/sec in write requests for 32MB block size.

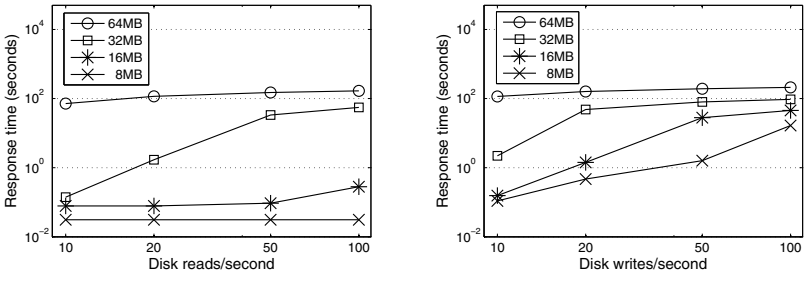


Fig. 4. Response time over the requests per second (the y-axis is on log scale)

This suggests that a "hot-spot," swamped with requests, incurs significantly higher response time and becomes a bottleneck, which motivates us to alleviate such hot-spots. Formally, hot-spots are defined as a node n_i with an aggregate load, *e.g.*, total I/O operations per second (IOPS) or bytes transferred per second for reads and writes, beyond some threshold C_i. We can define C_i empirically as illustrated in Fig. 4. Based on the observation, we define the following load capacity constraint:

Constraint 1 (Load Capacity (C1)). *For every node n_i in the storage system, keep the total load $L_{*i} \leq C_i$.*

Meanwhile, to provide fault tolerance, two replicas of the same object should not be in the same failure domain, *e.g.*, the same node or rack. We use the fault tolerance constraint for each node as follows:

Constraint 2 (Fault Tolerance (C2)). *Every node n_i should contain at most one replica of a given chunk (or partition).*

To achieve load balancing, our goal is to generate a sequence of operations satisfying C1 and C2, while optimizing the reconfiguration cost. The two available knobs for data migration are:

- **Swap:** This operation simply switches the role of the primary replica of a chunk with one of the secondary replicas.
- **Move:** This operation physically migrates the chunk data between two nodes.

For instance, to avoid some nodes being swamped with requests, Hotmail "moves" users between disk groups to maintain a high full watermark on each disk group, and servers "swap" primary and secondary roles.

This paper mainly focuses on optimizing moves, as they involve higher migration costs, which depends on network topology between servers, *e.g.*, the lowest cost to move data is within the same rack, relatively higher within neighboring racks, and more far away. To logically visualize this problem in 2D, Fig. 5 emulates bandwidth costs using an L2-metric, or Euclidean distance, based on inter-node distance (a circle of radius r denotes a logical partition of nodes, say a rack). In this figure, a smaller distance between two nodes is more desirable meaning it incurs lower migration cost.

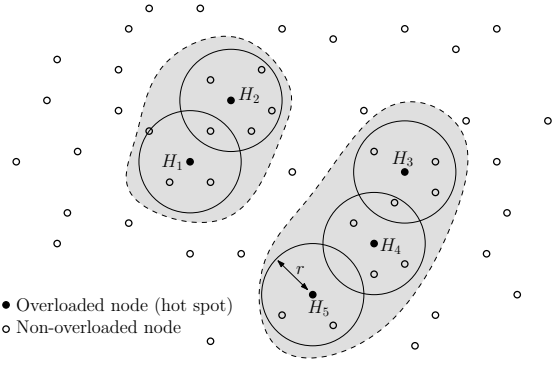


Fig. 5. Logical view of topology-dependent bandwidth costs for hot-spots H_1 to H_5

To summarize, our load balancing problem can formally be defined as follows:

Definition 1 (Load Balancing). *Find the optimal placement $\mathcal{P}'$ satisfying C1 and C2 from initial placement $\mathcal{P}$ by generating a cost-optimal swap and move sequence.*

3 Algorithms for Load Balancing

In this section, we describe our optimization problem for load balancing more formally. Specifically, Section 3.1 formulates our goal and constraints as a linear programming (LP) problem [15], which is not scalable in our target scenario. Section 3.2 describes a two-phase approximation algorithm for near-optimal solutions.

While a joint-optimization of combining two available knobs, swap and move operations, seems attractive, we only consider the move operation for two reasons in this

formulation. First, the number of decision variables will increase significantly making the computation prohibitive. Second, from a practical view, operators prefer to invoke cheap swap operations before they consider expensive data migrations. Thus, prior to applying the move operations, to cool down our entire system, we can iteratively swap the primaries in the hottest node and the secondaries in the cold node until no swap operation is applicable. After that, with respect to the remaining hot nodes, *i.e.*, on the swapped system, we apply our following strategies with only the move operations.

3.1 Exact Solution: ILP Formulation

LP is a mathematical model, populating "decision variables" with values achieving the best outcome (lowest migration cost in our problem). There can be multiple optimal solutions in terms of cost model, but our model is guaranteed to return one of such solutions. In our problem, decision variables are defined as a binary hyper-matrix Y representing optimal placement of the replicas. If the k-th replica of the j-th partition is best to be placed on the i-th node after migration, Y_{ijk} will be set to 1, and 0 otherwise. As decision variables are all integers, our problem is an integer linear programming (ILP) problem.

Table 1 summarizes all notations. In particular, current placement A_{ijk} can be represented in the same way as Y_{ijk}. With respect to A_{ijk} and Y_{ijk} placements, the bandwidth cost to move the k-th replica of the j-th partition from the node containing the replica to the i-th node can be defined. Note that $\overline{A}$ is a complement representation of A that converts 0 to 1 and 1 to 0. $\overline{A}_{ijk}Y_{ijk} = 1$ means the movement of the k-th replica of the j-th partition to the i-th node. Our goal is to find Y minimizing the total cost for such movements, which can be formally stated as an objective function in Eq. (1).

Constraints C1 and C2 discussed in Section 2 can be formally stated as Eq. (2) and (5) respectively.

$$\textbf{minimize} \quad \sum_{i}\sum_{j}\sum_{k} \overline{A}_{ijk}Y_{ijk}B_{ijk} \tag{1}$$

$$\textbf{subject to} \quad \forall i : \sum_{j}\sum_{k} Y_{ijk}L_{jk} \leq C_i \tag{2}$$

Table 1. Notations for optimal ILP model

Notation	Description
Y_{ijk}	1 if k-th replica of j-th partition is on i-th node after move, 0 otherwise
A_{ijk}	1 if k-th replica of j-th partition is on i-th node before move, 0 otherwise
B_{ijk}	Bandwidth cost to move k-th replica of j-th partition from the node containing the replica into i-th node
C_i	Load capacity of i-th node
L_{jk}	Load of k-th replicas of j-th partition
l	Number of replicas
F_q	Set of nodes with the same fault domain index q

$$\forall j, \forall k : \sum_i Y_{ijk} = 1 \tag{3}$$

$$\forall j : \sum_i \sum_k Y_{ijk} = l \tag{4}$$

$$\forall q, \forall j : \sum_{i \in F_q} \sum_k Y_{ijk} \leq 1 \tag{5}$$

$$\forall i, \forall j, \forall k : Y_{ijk} = 0 \text{ or } 1 \tag{6}$$

In the above ILP formulation, Eq. (5) can incorporate placement constraints for fault domains (*e.g.*, a node, rack, or cluster) where replicas of the same partition should be placed in different fault domains to ensure high availability. Note that our LP model does not have to keep track of whether a replica is a primary or a secondary. The model incorporates only move operations because the primary-secondary swaps are already completed before applying the model.

Our formulation suggests that obtaining an exact solution is not practical, especially in our target scenario of 10M partitions, 3 replicas, and 10K nodes. Y_{ijk} holds 300 billion variables, which is prohibitive in terms of both computation and memory usage. This observation motivates us to develop an efficient approximation in the following section.

3.2 Approximation

In this section, we present two complementary approaches for efficient computation: (1) reducing the number of decision variables and (2) speeding up the computation.

Reducing Number of Decision Variables: Two-Phase Approach. To reduce the number of decision variables Y_{ijk}, representing all possible moves of the k-th replica of the j-th partition on the i-th node, we decide to restrict the moves (by restricting source and target nodes) to yield an approximate yet efficient solution, based on two key observations; we empirically evaluate the efficiency of our approach with respect to the offline optimal in Section 5.

First, as observed in Section 2, loads are skewed to a few hot-spots, *e.g.*, power law distribution. This suggests that we can safely localize the **source node** of a move to hot-spots. Second, only if we knew the maximal migration cost r_{max} of a single object a priori, we could easily eliminate any **target node** incurring higher costs than r_{max} from the model without compromising the optimality. However, as r_{max} is obtained after optimization, we start with its lower bound estimation r and iteratively increase the value by Δr until it converges to r_{max}, *i.e.*, a linear search, as in 'phase 2' described below. As over-estimating the value incurs high LP computation cost, a binary or an evolutionary search was not suitable for this purpose.

To reflect these two observations, our two-phase approach consists of (1) selecting a set of objects to move from all hot-spots, and (2) deciding a set of target nodes to which the selected objects are to be migrated.

Phase 1: Restricting Source Nodes/Objects

As discussed above, we restrict moves to those from hot-spots (with loads higher than C_i). From each hot-spot i, we then need to decide which set S_i of objects to move in order to reduce the loads below C_i. There can be various strategies as follows:

- Highest-load-first: Starting from the object with the highest request load, add objects to S_i in descending order of loads until the load is below the C_i threshold.
- Random: Selecting random objects from node i and adding to S_i until the load is below the C_i threshold.

Between these two strategies, we take the highest-load-first to minimize the number of objects to move. In Section 5, we show that this approach works well in practice.

Phase 2: Target Node Selection

As discussed above, we can safely restrict our search for destination nodes to those with a migration cost less than r_{max}. As this value is unknown, we perform a linear search starting with a small r, *e.g.*, the cost of migrating into the nodes in the same rack, and run the ILP model. If a feasible solution exists, we stop searching. Otherwise, we expand the radius further $r + \Delta r$, and resume running the ILP model.

We now discuss how to formulate the ILP model from the revised strategy. Table 2 summarize decision variables and all constants. Based on these, we revise the ILP model in the previous section as follows:

$$\textbf{minimize} \quad \sum_i \sum_j X_{ij} B_{ij} \tag{7}$$

$$\textbf{subject to} \quad \forall j : \sum_i X_{ij} L_i - \sum_{i' \in S_j} \sum_{j'} X_{i'j'} L_{i'} + L_{*j} \leq C_j \tag{8}$$

$$\forall i : \sum_{j \in R(i,r)} X_{ij} = 1 \tag{9}$$

$$\forall i, \forall j \notin R(i,r) : X_{ij} \leq 0 \tag{10}$$

$$\forall p, \forall q : \sum_{i \in G_p} \sum_{j \in F_q} X_{ij} \leq 1 \tag{11}$$

$$\forall i, \forall q : \sum_{j \in F_q} X_{ij} \leq I_{iq} \tag{12}$$

$$\forall i, \forall j : X_{ij} = 0 \text{ or } 1 \tag{13}$$

In this revision, the decision variables are defined as a binary matrix X representing optimal movements of the replicas. If the i-th object moves to the j-th node, X_{ij} will be

Table 2. Notations for approximate ILP model

Notation	Description
X_{ij}	1 if i-th object moves to j-th node, 0 otherwise
C_j	Load capacity of j-th node
B_{ij}	Bandwidth cost for moving i-th object into j-th node
S_j	Set of objects selected to move from j-th node
L_i, L_{*j}	Load of i-th object, total load of j-th node
$R(i,r)$	Nodes within radius r from the node i-th object belongs to
G_p	Set of selected objects with partition index p to move from the overloaded nodes
F_q	Set of nodes with the same fault domain index q
I_{iq}	0 if q-th domain contains the same partition as i-th object, 1 otherwise

set to 1, and 0 otherwise. Using this representation, an objective function for minimizing such movement cost is formally stated in Eq. (7).

Table 2 describes all notations for representing the following constraints–Eq. (8) represents the capacity constraint for each j-th node. In this constraint, the first two terms represent the incoming load to the j-th node and outgoing load from the j-th node, respectively. Eq. (9) and (10) are the constraints for restricting the target nodes within the search radius r. Eq. (11) and (12) are the constraints to ensure fault tolerance by preventing objects with the same partition from being placed on the same fault domain similarly to Eq. (5) in the optimal ILP formulation. In particular, G_p denotes the group of all object replicas belonging to partition index p in the set of selected objects. Thus, Eq. (11) imposes a constraint such that candidate objects having the same partition index cannot be placed on the same fault domain while Eq. (12) imposes a constraint such that a candidate object cannot be placed on a fault domain which already has a replica from the same partition. Note that these two constraints are useful when a set of candidate objects is merged from nodes with overlapping search regions. For a single node, these constraints hold trivially as at most one copy of a replica partition can be hosted on the node.

In this formulation, we can significantly reduce the number of decision variables compared to the previous formulation, because we only consider the selected objects in hot-spots and constrain the search space within a specific radius. For instance, in the same scenario of 10M partitions, 3 replicas, and 10K nodes, if the number of the selected objects in hot-spots is 10K$\times$30 (1% of 3K objects per node on average) and the number of nodes within a typical rack is 40, the number of variables is about 12M, which is significantly less than the 300 billion in the previous model.

Speeding-up Computation. We next describe how to speed up the computation using the following two approaches: (1) Divide-and-Conquer (D&C), and (2) relaxation to the LP model.

(1) Divide-and-Conquer: We formulate the ILP model to move all selected objects from the hot-spots to the candidate nodes within a specified cost radius. However, we observe that the computation for a hot-spot can be separated if the neighborhood defined by its cost radius does not overlap with that of others. That is, we can divide the given problem into sub-problems of optimizing moves for "disjoint" groups of hot-spots with overlaps, and merge the solutions without loss of accuracy, which shows significantly higher performance empirically. Fig. 5 illustrates the D&C approach for two disjoint groups of overlapping hot-spots. Unlike running ILP once for all hot-spots, D&C runs the ILP model independently and in parallel for two grey groups on $\{H_1, H_2\}$ and $\{H_3, H_4, H_5\}$.

(2) Relaxation to LP: The ILP model, which requires decision variables to be integers, is more expensive than LP. We thus relax the model by changing the decision variables from binary to real numbers between 0 and 1, *i.e.*, by using constraint $\forall i, \forall j : 0 \leq X_{ij} \leq 1$ instead of Eq.(13). However, such relaxation can often return fractional values between 0 and 1 for some target nodes, *e.g.*, $X_{31} = 0.5$, $X_{32} = 0.3$, and $X_{33} = 0.2$. We thus use these scores to prioritize the target nodes by picking the one with the highest score. Meanwhile, if the movement causes violation of the constraints C1 and C2, we

Algorithm 1. Two-phase approximation: cost optimization for load balancing

```
1  M ← {}; // Initialize movement set
2  r ← r_0; // Initialize search radius
   // Merge the objects to move whose search regions are
   // overlapped
3  {S'_1, ··· , S'_n'} ← MergeObjects (S_1, ··· , S_n, r);
4  for k ← 1 to n' do
       // Find target nodes within r from the nodes
       // containing each group k
5      TN_k ← FindTargetNodes (k, r);
6      {X_ij | ∀i ∈ S'_k, ∀j ∈ TN_k} ← SolveLP (S'_k, TN_k, r);
7      if LP solution is feasible then
8          foreach i ∈ S'_k do
9              Let {X_i1, ··· , X_im} be a sorted list s.t. X_ij > 0;
10             for j ← 1 to m do
11                 if L_i + L_*j ≤ C_j then
12                     M ← M ∪ {(i, j)};
13                     break;
14 if ∃S'_k, LP solution is infeasible then r ← r + Δr;
15 if there still exist object(s) to move then goto line 3;
```

have to bypass it. In this example, we first check whether moving the object id 3 to the node id 1 (as represented by X_{31}) is feasible to satisfy the constraints. If it is feasible, we move it. Otherwise, we iteratively check the next movement on X_{32} with the second highest score, and so on.

In summary, Algorithm 1 formally explains the optimization. There are three functional modules: (1) `MergeObjects`, merging the objects to move whose search radii overlap, (2) `FindTargetNodes`, finding the target nodes for each overlapping group, and (3) `SolveLP`, running the LP model for each overlapping group.

4 Implementation of Ursa on Windows Azure

This section describes our implementation of Ursa by using APIs provided by a real cloud system Windows Azure for deploying multiple servers in a distributed cloud environment. Our system consists of three main modules: (1) a resource manager that performs load balancing, (2) a load generator/trace replayer, and (3) a set of storage nodes. Fig. 6 illustrates the overall architecture. The resource manager periodically collects load statistics of objects requested to the storage nodes and their location information, computes load balancing decisions by applying different algorithms such as ILP/LP models, and finally it generates messages to actuate the data migration operations and conveys them through a message queue to individual storage nodes. The load generator outputs storage requests by generating read and write messages for objects in the storage nodes according to Hotmail read/write requests logged. Similarly, the trace replayer takes as input a log of the read and write requests for running applications (*e.g.*, Hotmail) and issues I/O requests as per their temporal patterns. The location information for

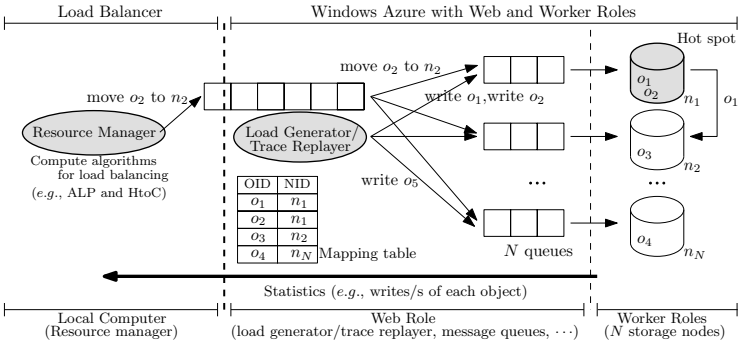


Fig. 6. Overall architecture of Ursa's prototype implementation on Windows Azure

all objects is maintained in a mapping table corresponding to the MDS lookup table in Fig. 2, which is used to assign each request message to the corresponding storage node. The storage nodes receive the requests through their corresponding message queues. After processing the requests, the storage nodes send request acknowledgements which are used to compute load statistics for the resource manager. Each storage node handles both the read/write requests and data migration operations. Alternatively, our architecture can support a separate queue per storage node for data migration operations, similar to the principle of separating control and data plane in distributed systems. To actuate data migration between nodes, the source node transfers the corresponding objects to the destination node over a TCP channel; the mapping table is updated after migration is completed. For correctness, write requests on objects being migrated are blocked until the migration operations are completed.

We implement the resource manager as a module running on a local computer, the load generator as a web role instance, and the storage nodes as worker role instances, where the web role and the worker role compose the application of Windows Azure. The web role instance is the interface to pass the migration operations decided by the resource manager and the request messages generated from the load generator/trace replayer to the storage nodes. We instantiate the web role as a large VM with 4 CPU cores, 7 GB memory, and 1,000 GB disk because it interacts with multiple worker role instances. The worker role instances execute write/read requests on data objects delivered from their queues, or move their objects to other worker role instances as specified by the resource manager when hot-spots occur. We instantiate them as small VMs with 1 CPU core, 1.75 GB memory, and 20 GB disk.

5 Experiments

Section 5.1 validates the effectiveness of our load balancing approach on a simulator over large-scale datasets up to ten thousand nodes and thirty million objects. Section 5.2 then evaluates our framework on the real system implemented on Windows Azure.

5.1 Evaluation on Simulator

This section evaluates the effectiveness of the load balancer by synthetically generating load statistics based on Hotmail traces. We first report our experimental settings and then validate the effectiveness and the scalability of our approach over synthetic datasets.

Settings. We synthetically generate several datasets (denoted as the naming convention of [number of storage nodes]-[number of partitions]) from 0.1K-0.1M to 10K-10M to closely emulate the statistics aggregated from storage servers in Hotmail clusters. More specifically, given the parameter pairs X-Y, we generate the dataset to mimic Hotmail traces by the following procedure:

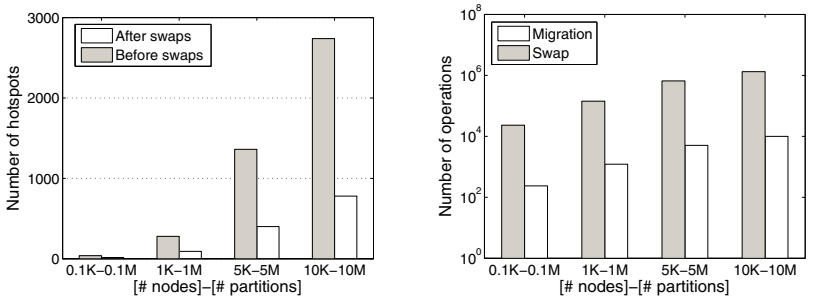
1. Generate Y partitions where every partition has $l = 3$ replicas, and assign workload values following the power-law distribution into partitions. We assume that the workload of primaries is two times higher than their secondaries corresponding to the read to write ratio observed in our case-study workload.
2. Randomly distribute X nodes in two-dimensional Euclidean space. We attempt to reproduce the bandwidth cost for passing a message between two servers that may be hosted either on the same rack, neighboring racks, or topologically far apart so that it is proportional to the Euclidean distance.
3. Randomly distribute all replicas into nodes satisfying constraints such that no node can include two replicas corresponding to the same partition.
4. Assign load capacities C_i's to X nodes. In particular, when the total workload of the node n_i is L_{*i} by all the replicas assigned to n_i, the distribution of the ratio L_{*i}/C_i over X nodes follows the distribution illustrated in Fig. 1 under the assumption that the total workload of the node is proportional to the CPU utilization. The nodes with L_{*i}/C_i higher than 0.7 are regarded as hot-spots.

We then measure (1) the running time (the placement decision time without the reconfiguration time for actually passing the messages) and (2) the migration cost (sum of the distances between two nodes to move objects) and (3) the number of migration operations. Specifically, we measure the reconfiguration cost as the cost of migrations (and not swaps), because we apply the same swap optimization strategy for all algorithms, namely iteratively swapping the primary in the hottest node and the secondary in a cold node until no swap operation is applicable. In particular, we apply the swap operations with 90% success probability to emulate a small fraction of out-of-date secondaries. The swap operations for the out-of-date secondaries may require more expensive cost than the migration operations to synchronize their state. That is, we regard the swap operations for such secondaries as the failed operations. As we consider only migrations, all the message sizes are the same and thus the number of migration operations can also be interpreted as the bandwidth cost, which would be directly proportional to these numbers. The bandwidth cost is an important metric for reconfiguration costs as data center networks tend to have high bandwidth oversubscription [13].

We compare our best approach (especially the approximation, denoted as ALP (for Approximate LP), in Section 3) with a natural baseline called HtoC (for Hottest to Coldest), which iteratively eliminates hot-spots until no hot-spots exist by moving the

hottest objects in a hot-spot node into the coldest node. (Note that, though we use a single machine, ALP can be easily parallelized for each non-overlapping group of the search regions.) HtoC is a simplified variation of the greedy algorithm suggested by Kunkle and Schindler [16], as the proposed algorithm "as is" cannot apply to a larger scale setup because of its complex objective function of minimizing an imbalance. As another baseline, we adopt a metaheuristic optimization algorithm using a simulated annealing technique called SA (for Simulated Annealing), similar to an evolutionary algorithm approach in [16]. SA randomly moves the objects into other nodes to minimize the load variance. We note that SA has eight parameters to tune and failing to optimize these parameters may lead SA to generate excessive migrations. For our evaluations, we empirically tune these parameters including setting the maximum number of migrations as 10K and the maximum running time as 30 minutes.

Results. We first motivate the necessity of migration operations, then evaluate our approximation techniques to speed up our ILP formulations, and finally compare our best approach with the baselines.

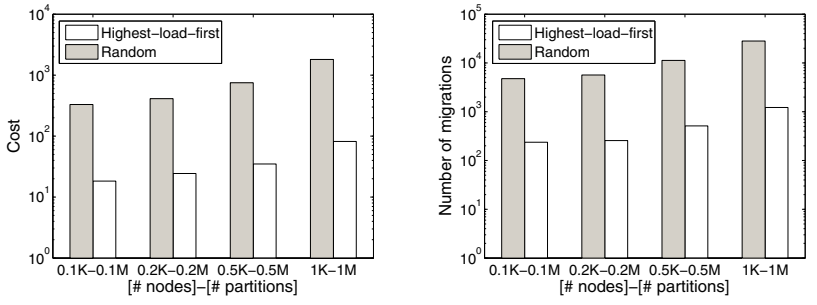


(a) Number of hot-spots (b) Number of operations

Fig. 7. Effect of swap and migration operations

To validate the necessity of the migration operations, we first evaluate the number of remaining hot-spots after applying only swap operations. Fig. 7 shows such results over various scales of datasets. Observe from Fig. 7(a) that there are still about one third of the initial hot-spots even after swaps, and Fig. 7(b) shows the numbers of swap and migration operations respectively until all hot-spots are eliminated.

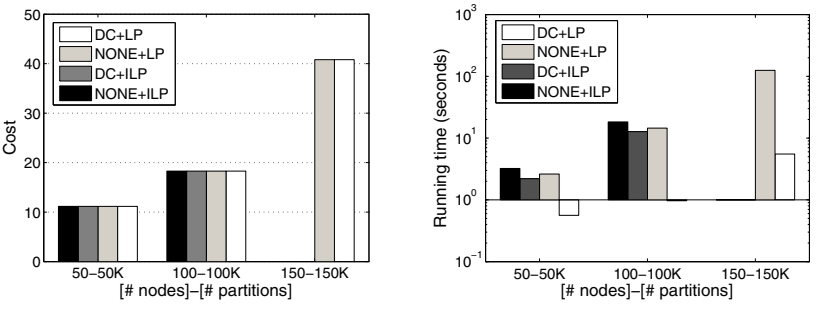
Fig. 8 shows the effect of the object selection strategies in our two-phase approach for reducing the number of decision variables. In particular, we compare two simple approaches, highest-load-first and random. Fig. 8(a) shows the total cost for migration operations. Observe that the highest-load-first strategy is much cheaper than the random in all settings, which is explained by the number of migration operations in Fig. 8(b). As the workload of objects follows the power-law distribution, the highest-load-first selects far fewer objects with the highest load than the random. Thus, in all experiments hereafter, we use the highest-load-first strategy in the first phase.



(a) Cost (b) Number of migrations

Fig. 8. Effect of selecting source objects

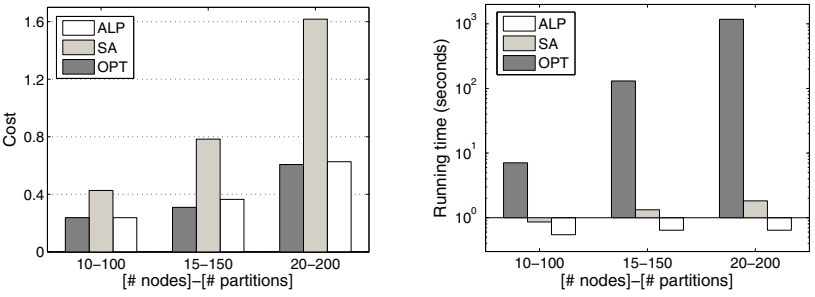
Fig. 9 shows the effect of our two speedup techniques suggested in Section 3.2, divide-and-conquer and relaxation from ILP to LP. We evaluate combinations of these two techniques (denoted as [NONE or DC]+[ILP or LP]) with respect to the migration cost and the running time in Fig. 9(a) and (b), respectively. Observe that, while all combinations show the same or almost the same cost, DC+LP applying both optimization techniques shows much shorter running time than the others. We do not plot the results of NONE+ILP and DC+ILP in 150-150K configuration because their running times become too large (more than two days).



(a) Cost (b) Running time

Fig. 9. Effect of speeding up computation

Fig. 10 compares the optimal cost computed by the optimal ILP model in Section 3.1 (denoted as OPT) with our best approach ALP, *i.e.*, DC+LP. Observe that ALP shows a similar cost to OPT in Fig. 10(a) while ALP exhibits a significantly higher scalability than OPT in the running time of Fig. 10(b). Note that we evaluate small-scale datasets due to the low scalability of OPT.



(a) Cost (b) Running time

Fig. 10. Performance comparison of ALP with respect to SA and the optimal solution

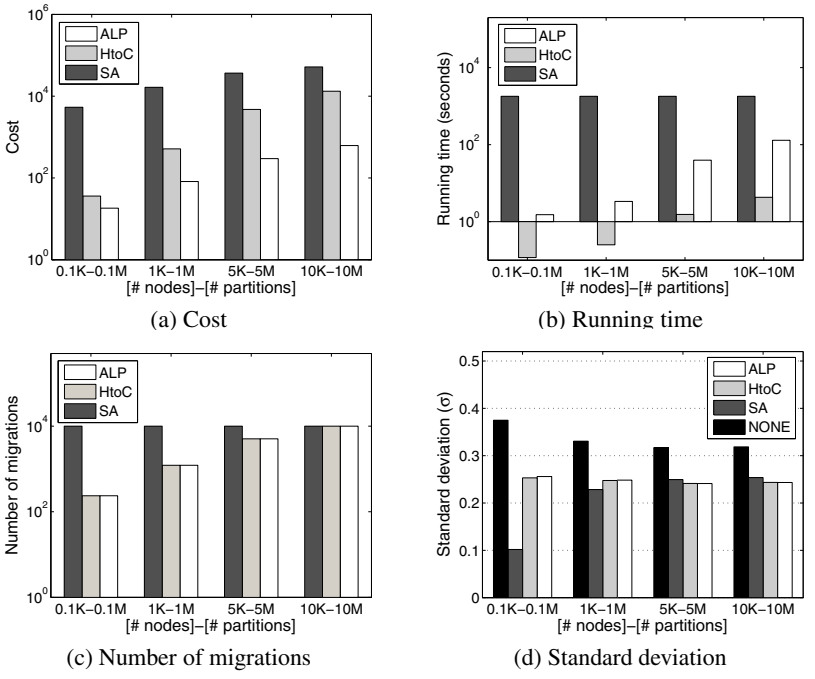
Fig. 11 shows the effect over large-scale datasets: Fig. 11(a) and (b) show the total cost and the running time of ALP. Observe that ALP is a clear winner in all settings except for the running time. In terms of cost, ALP is 80-300 times cheaper than SA and 2-20 times cheaper than HtoC. Meanwhile, the running time of ALP is comparable to simple heuristics and remains reasonable, *i.e.*, 130 seconds, even in the largest setting of 10K-10M. Fig. 11(c) shows the effect over the number of migrations (proportional to the bandwidth cost). ALP and HtoC have the same number of operations, because both adopt the same method, highest-load-first, for selecting the objects to be migrated from the hot-spots. Fig. 11(d) shows the standard deviation σ over the loads of storage nodes. NONE depicts σ before the reconfiguration. SA has the lowest σ at small scale 0.1K-0.1M according to the goal of minimizing σ. However, as the scale increases, it becomes less effective. In particular, in 5K-5M and 10K-10M configurations, we observe that, when ALP and HtoC eliminate hot-spots, they result in slightly lower σ than SA.

5.2 Evaluation on Windows Azure

This section describes the evaluation of our prototype on Windows Azure described in Section 4. We first report our experimental settings and then discuss the results.

Settings. We simplify some environmental settings for convenience of evaluation. More specifically, we assume that all objects have the same block size, *e.g.*, 32MB, and the load generator only generates the write messages on those objects. (When the read and write requests are blended, we can easily quantify the capacity of the node as a weighted sum w_1·reads/s + w_2·writes/s.) Under these assumptions, before executing the above setup on Azure, we measure the load capacity C_i of the storage node in the unit of writes/s. We first find a threshold value T writes/s such that, for the storage node, the response time rapidly increases, and set $C_i = 0.7T$. We then generate T writes/s messages for the hot-spots (20% of all the nodes) and $0.4T$ writes/s messages for the other nodes. Meanwhile, in this setting, we do not distinguish between primary and secondary replicas in order to focus on the migration operation incurring significantly higher cost than the swap operation.

We perform evaluations in the following two scenarios–**Random** and **Controlled**. In **Random**, we leave Windows Azure to deploy nodes, and as a result, have nodes deployed randomly in the data center network (*i.e.*, similar migration costs between each



(a) Cost (b) Running time

(c) Number of migrations (d) Standard deviation

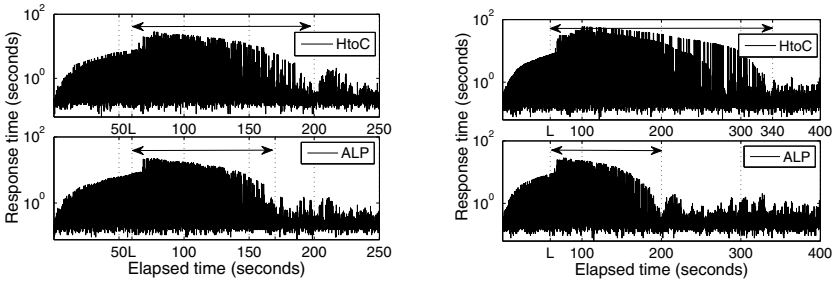
Fig. 11. Comparing (a) the cost, (b) running time, (c) number of migrations, and (d) standard deviation under different configurations for ALP, HtoC, and SA

source-destination pair). In contrast, in **Controlled**, we emulate data center scenarios where we have control over which nodes go to which racks. In these scenarios, migration costs can vary based on the underlying rack assignment. More specifically, in our system, the nodes of a data center network form a hierarchical structure of the spanning tree topology [13]. Thus, as traffic goes up through switches and routers, the oversubscription ratio increases rapidly, which means that movement of the data across rack or cluster causes significant delay compared to the movement within the same rack or cluster. As Windows Azure does not allow user-specified deployment of VMs to racks, we emulate arbitrary rack assignment by adding time delays.

We evaluate the effectiveness of our approach over various scales of datasets, *e.g.*, 30-900 (30 storage nodes and 900 partitions). Note that we assume the number of replicas is three in all setups. Thus, 30-900 implies that the number of total objects is 2,700.

Running experiments in Windows Azure enables us to measure "elapsed time" which includes the time for running ALP and the time it takes for the system to stabilize (*i.e.*, until the system responds to a request within 1 second). We thus use two metrics in this section: (1) the elapsed time and (2) the bandwidth cost (measured as the number of operations as discussed before.)

We then compare our proposed method with HtoC, a winner solution among the baseline approaches as shown in the experiment on our simulator.

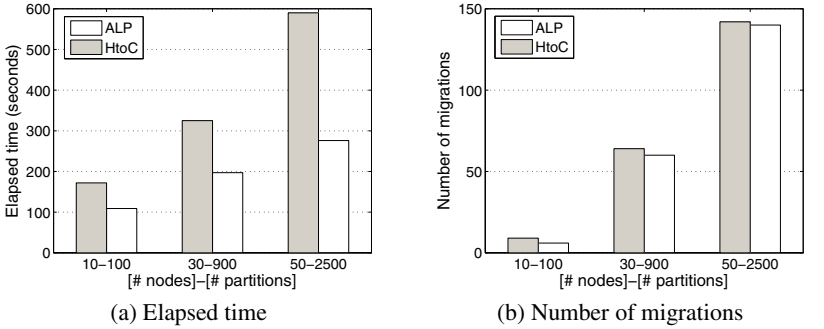


(a) Random (b) Controlled

Fig. 12. Response time over elapsed time

Results. Fig. 12 shows the response time of each request over elapsed time with the dataset 30-900. We observe that the response time rapidly increases before the load balancer is called at 60 second point marked as 'L' on x-axis. After the call, the response time starts to be stable passing through the reconfiguration by using two approaches. We argue **Random**, where migration costs are more or less the same, is not a favorable scenario for our proposed system, as a randomly selected node with no computation and the one selected from our optimization computation would incur similar migration costs (*i.e.*, little margin for optimization). However, even in this unfavorable setting, ALP achieves stable state about 30 seconds earlier than HtoC in Fig. 12(a). As expected, our performance gain is much clearer in **Controlled**, where migration costs differ significantly. As shown in (b), ALP is about 140 seconds faster than HtoC.

Fig. 13 presents our results in varying settings of **Controlled** scenarios, *i.e.*, 10-100, 30-900, and 50-2500. We observe from Fig. 13(a) that the performance gap between the two approaches increases as the data scale increases, *e.g.*, two times faster in 50-2500, which suggests the scalability of our approach. Note from Fig. 13(b) that two approaches have a similar number of migrations because both select the objects to move from the hot-spots using the highest-load-first.



(a) Elapsed time (b) Number of migrations

Fig. 13. Elapsed time and number of migrations for reconfiguration in Controlled setting

6 Related Work

As cloud systems have become popular as cost-efficient and agile computing platforms, there have been various efforts to efficiently manage them. For instance, pMapper [24] performs cost-aware server consolidation and dynamic VM placement to minimize power. Rhizoma [26] leverages constraint logic programming (CLP) to minimize cost for deploying and managing distributed applications while satisfying their resource requirements and performance goals. However, these approaches do not address load balancing in cluster storage systems or perform migration in a topology-aware manner.

Meanwhile, while many existing methods for load balancing perform dynamic placement of individual data objects [3,16,25,27], they aim at evenly distributing loads over all nodes. However, finding load balancing schemes with minimal re-configuration costs is a variant of the bin-packing or knapsack problem, which is inherently expensive and solvable offline [11]. As a result, existing solutions cannot scale to cloud-scale systems with more than ten thousand nodes incurring prohibitive reconfiguration costs. For instance, Kunkle and Schindler [16] tackle this problem by applying a greedy search, an evolutionary algorithm, and an optimal ILP model. However, these algorithms are deployed and evaluated in a small scale system, *e.g.*, six nodes, and it is non-trivial to scale these approaches for a larger scale.

In clear contrast, our approach aims to scale up to tens of thousands of nodes by eliminating hot-spots to minimize the reconfiguration cost. Similarly motivated by the challenge for scaling down peak loads, Everest [19] proposed to off-load the workload from overloaded storage nodes to under-utilized ones. A key difference is Everest targets to flash crowds with bursty peaks. To optimize for workload of this type, Everest off-loads writes to pre-configured virtual storage nodes temporarily then later reclaims to original nodes lazily when the peak subsides. Meanwhile, Ursa has a complementary strength of handling long-term changes in load and making topologically-aware migration decisions.

Ursa has advantages for read-heavy workloads, such as Hotmail workload studied in this paper. In Everest, only writes are off-loaded and such write offloading can increase read latency, as a write offloaded to multiple locations with data versioning, *i.e.*, N-way offloading, requires the following read request to find the latest write (which is bottlenecked by the slowest server). Another restrictive assumption of Everest for Hotmail workloads is that reads to off-loaded writes (recent data) are rare, while such pattern can be frequently observed in many applications, such as email where a user immediately checks for incoming mail.

Data migration challenge has been raised in the context of database systems built on cloud computing environments as well [5,6,7,9,10,17,20,21]. These systems can be categorized into those using replication strategies [5,17,20,21] or limited dynamic data migration [6,7,9,10]. However, existing dynamic migration approaches assume some oracle to determine the source and target nodes of migration and leave the problem of identifying cost-optimal dynamic reconfiguration as future work, while we identify a scalable and efficient solution for such reconfiguration.

7 Conclusion and Future Work

We have presented a new middleware system Ursa for load balancing. Ursa formulates load balancing as an optimization problem and develops scalable optimization techniques. Our evaluations using traces from Hotmail are encouraging: Ursa scales to large systems while reducing reconfiguration cost and can compute placement decisions within about two minutes for 10K nodes and 10M objects in cloud-scale systems.

As future work, we consider the following issues:

First, regarding **interference during reconfiguration**, reconfiguration does consume resources, CPU and bandwidth, that could otherwise be used to serve requests. To control interference, we can build a mechanism, such as TCP Nice [23] developed for background data transfer, to automatically tune the right balance between interference and resource utilization. For workloads with many sudden transient changes, *e.g.*, flash crowds on the Internet, interference and reconfiguration costs may become significant. One approach would be to specify a resource budget for reconfigurations to ensure that interference to applications is within acceptable bounds.

Second, regarding **the unit of migration**, in this paper, we regard an object as the unit of migration, but supporting a varying degree of granularity may enable further optimization. For instance, partitions or chunks may be collected into a group, such that a single MDS message is sufficient to migrate or swap the whole group. That is, adopting a coarser unit can reduce the number of MDS operations.

Lastly, regarding **trade-offs between reconfiguration cost and speed**, our framework does have hidden knobs controlling this trade-off. For instance, in divide-and-conquer computation, a small initial radius r would incur less computation cost for LP but may require more rounds of LP computation, while larger value requires a single but more costly LP computation with possibly high reconfiguration cost. Meanwhile, divide-and-conquer computation does not compensate accuracy if (1) regions do not overlap and (2) a region has an infeasible solution. Though divide-and-conquer can still be used for faster LP computation when these conditions are violated, the process will identify sub-optimal results. Also, when using the greedy highest-load-first strategy, we may pick a very hot object that can only be migrated to a cold node far away, whereas we could pick several relatively less hot objects incurring lower migration cost. A joint optimization minimizing the overall cost would avoid this case and thus reduce reconfiguration cost, but may incur higher LP running time.

Acknowledgments. We thank our shepherd Steven Ko and the anonymous reviewers for their feedback. We are grateful to Hua-Jun Zeng for providing us with an initial simulator code and Boon Yeap for helping us understand the Hotmail datasets. This research was supported by Microsoft Research and the MKE (The Ministry of Knowledge Economy), Korea, under IT/SW Creative research program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2010-C1810-1002-0003).

References

1. Amazon S3, http://aws.amazon.com/s3/
2. Windows azure, http://www.microsoft.com/windowsazure/

3. Abd-El-Malek, M., Courtright II, W.V., Cranor, C., Ganger, G.R., Hendricks, J., Klosterman, A.J., Mesnier, M., Prasad, M., Salmon, B., Sambasivan, R.R., Sinnamohideen, S., Strunk, J.D., Thereska, E., Wachs, M., Wylie, J.J.: Ursa Minor: Versatile Cluster-based Storage. In: Proc. of FAST (2005)
4. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. In: Proc. of OSDI (2006)
5. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: a Workload-Driven Approach to Database Replication and Partitioning. In: Proc. of VLDB (2010)
6. Curino, C., Jones, E., Zhang, Y., Wu, E., Madden, S.: Relational Cloud: The Case for a Database Service. Technical Report MIT-CSAIL-TR-2010-014, MIT (2010)
7. Das, S., Nishimura, S., Agrawal, D., Abbadi, A.E.: Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms. Technical report, UCSB (2010)
8. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proc. of OSDI (2004)
9. Elmore, A., Das, S., Agrawal, D., Abbadi, A.E.: Who's Driving this Cloud? Towards Efficient Migration for Elastic and Autonomic Multitenant Databases. Technical report, UCSB (2010)
10. Elmore, A., Das, S., Agrawal, D., Abbadi, A.E.: Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In: Proc. of SIGMOD (2011)
11. Eric, E.A., Spence, S., Swaminathan, R., Kallahalla, M., Wang, Q.: Quickly Finding Near-optimal Storage Designs. ACM Transactions on Computer Systems 23, 337–374 (2005)
12. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google File System. SIGOPS Operating System Review 37, 29–43 (2003)
13. Greenberg, A.G., Hamilton, J.R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D.A., Patel, P., Sengupta, S.: VL2: A Scalable and Flexible Data Center Network. In: Proc. of SIGCOMM (2009)
14. Gulati, A., Kumar, C., Ahmad, I., Kumar, K.: BASIL: Automated IO Load Balancing Across Storage Devices. In: Proc. of FAST (2010)
15. Hiller, F.S., Lieberman, G.J.: Introduction to Operations Research, 8th edn. McGraw-Hill (2005)
16. Kunkle, D., Schindler, J.: A Load Balancing Framework for Clustered Storage Systems. In: Sadayappan, P., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2008. LNCS, vol. 5374, pp. 57–72. Springer, Heidelberg (2008)
17. Lang, W., Patel, J.M., Naughton, J.F.: On Energy Management, Load Balancing and Replication. SIGMOD Record 38, 35–42 (2010)
18. Litwin, W.: Linear Hashing: A New Tool for File and Table Addressing. In: Proc. of VLDB (1980)
19. Narayanan, D., Donnelly, A., Thereska, E., Elnikety, S., Rowstron, A.: Everest: Scaling Down Peak Loads through I/O Off-loading. In: Proc. of OSDI (2008)
20. Savinov, S., Daudjee, K.: Dynamic Database Replica Provisioning through Virtualization. In: Proc. of CloudDB (2010)
21. Tam, H.V., Chen, C., Ooi, B.C.: Towards Elastic Transactional Cloud Storage with Range Query Support. In: Proc. of VLDB (2010)
22. Thereska, E., Donnelly, A., Narayanan, C.: Sierra: A Power-proportional, Distributed Storage System. In: Technical Report MSR-TR-2009-153 (2009)
23. Venkataramani, A., Kokku, R., Dahlin, M.: TCP Nice: A Mechanism for Background Transfers. In: Proc. of OSDI (2002)

24. Verma, A., Ahuja, P., Neogi, A.: pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 243–264. Springer, Heidelberg (2008)
25. Weil, S.A., Brand, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: A Scalable, High-performance Distributed File System. In: Proc. of OSDI (2006)
26. Yin, Q., Schüpbach, A., Cappos, J., Baumann, A., Roscoe, T.: Rhizoma: A Runtime for Self-deploying, Self-managing Overlays. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 184–204. Springer, Heidelberg (2009)
27. Zeng, L., Feng, D., Wang, F., Zhou, K.: A Strategy of Load Balancing in Objects Storage System. In: Proc. of CIT (2005)

Predico: A System for What-if Analysis in Complex Data Center Applications

Rahul Singh[1], Prashant Shenoy[1], Maitreya Natu[2],
Vaishali Sadaphal[2], and Harrick Vin[2]

[1] Dept. of Computer Science, University of Massachusetts,
Amherst, USA
{rahul,shenoy}@cs.umass.edu
[2] Tata Research Development and Design Center,
Pune, India
{maitreya.natu,vaishali.sadaphal,harrick.vin}@tcs.com

Abstract. Modern data center applications are complex distributed systems with tens or hundreds of interacting software components. An important management task in data centers is to predict the impact of a certain workload or reconfiguration change on the performance of the application. Such predictions require the design of "what-if" models of the application that take as input hypothetical changes in the application's workload or environment and estimate its impact on performance.

We present Predico, a workload-based what-if analysis system that uses commonly available monitoring information in large scale systems to enable the administrators to ask a variety of workload-based "what-if" queries about the system. Predico uses a network of queues to analytically model the behavior of large distributed applications. It automatically generates node-level queueing models and then uses model composition to build system-wide models. Predico employs a simple what-if query language and an intelligent query execution algorithm that employs on-the-fly model construction and a change propagation algorithm to efficiently answer queries on large scale systems. We have built a prototype of Predico and have used traces from two large production applications from a financial institution as well as real-world synthetic applications to evaluate its what-if modeling framework. Our experimental evaluation validates the accuracy of Predico's node-level resource usage, latency and workload-models and then shows how Predico enables what-if analysis in two different applications.

1 Introduction

Today online server applications have become popular in domains ranging from banking, finance, e-commerce, and social networking. Such server applications run on data centers and tend to be complex distributed systems with tens or hundreds of interacting software components running on large server clusters. As an example, consider an online stock trade processing application of a major financial firm. This application consists of 471 separate software components that process incoming stock trades at low latencies. Figure 1 shows another application which disseminates stock prices and market news to the terminals of stock traders; this application consists of 8970 components.

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 123–142, 2011.

The components are depicted as nodes of the graph and process stock data and news from a multitude of sources, filter, aggregate, and then disseminate updates for each company to desktops that subscribe to such streams. Such data center applications differ significantly in scale and complexity when compared to traditional multi-tier web applications.

Typical data center applications evolve over time as new functionality is added, its workload volume grows, or its hardware or software is updated. To deal with such changes, an important management task for administrators is to predict the impact of any planned (or hypothetical) change on the performance of individual components or the entire system. This task, which is referred to as *what-if analysis*, requires the design of what-if models that take as input a potential change in the application workload or its settings and predict the impact of that change on application behavior. However, given the complexity of today's data center application, manual design of such what-if models is no longer feasible since data center administrators may not be able to comprehend the behavior of a complex system of tens of interacting components. Consequently, a what-if analysis system must be able to automatically derive such models from prior observations of application's behavior. Further, the system must be able to scale to large complex applications with hundreds of interacting components, while allowing rich what-if analysis efficiently. While a number of modeling techniques have been proposed for distributed or multi-tier web applications [10,14,17,13,7,12], such models are not directly targeted for what-if analysis or are not designed to scale to larger data center applications such as the ones illustrated in Figure 1.

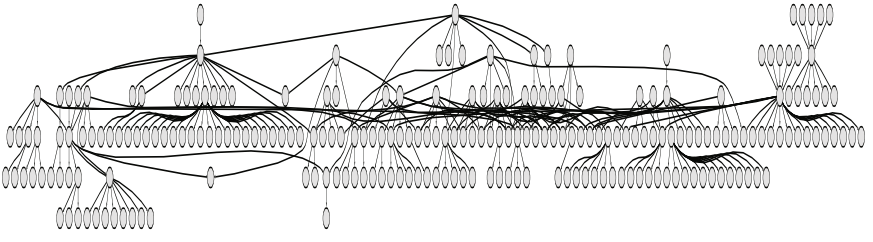


Fig. 1. Stock Price and Market Data Dissemination Application; only a subset of the application is shown for brevity

In this paper, we present Predico, a what-if analysis system to predict the impact of workload changes on the behavior of data center applications. Predico makes the following contributions:

– *Modeling of complex data center applications:* Predico employs a queuing-theoretic framework to model large distributed data-center applications. Our modeling framework is based on a network of queues and captures the dependence between the workload of each component of the application and the corresponding resource utilization, request latency and the outgoing workloads to other components. Predico uses monitoring data and request logs to estimate the parameters of such a model

and employs model composition to create larger system-level models for groups of interacting application components.
- *Intelligent query execution:* Predico uses a novel change propagation algorithm that uses these models to execute a what-if query and determine the impact of a workload change on other components. This algorithm first computes an influence graph to determine which application components are impacted by the specified what-if query and then uses a change propagation technique to propagate the specified workload change through each component in the influence graph.
- *Prototype Implementation:* We have implemented a prototype of our Predico what-if modeling framework. Our prototype incorporates a What-If Query Language (WIFQL) that can be used by administrators to pose queries. Since our prototype needs to handle large data center applications with hundreds of interacting components, we implement several optimizations to scale the modeling framework to such large applications. Specifically Predico uses on-the-fly model construction and employs a cache of previously constructed models to reduce model computation overhead.
- *Evaluation based on real traces and real-world synthetic applications:* We conduct an experimental evaluation of Predico using traces of two large production applications from a financial institution as well as realistic synthetic applications. Our experimental results validate the accuracy of Predico's modeling framework in building node-level resource usage, latency and workload models and illustrate Predico's ability to enable accurate what-if analysis.

2 Background and Problem Formulation

Our work assumes a large distributed application with N interacting components. We assume that the application is structured as a directed acyclic graph (DAG), where each vertex represents a software component and edges capture the interactions (i.e., flow of requests) between neighboring nodes. For simplicity, we assume that each component runs on a separate physical (or virtual) machine.[1] We assume that the DAG has one or more source nodes, that serve as entry points for application requests and one or more sinks, that serve as exits. The flow and processing of requests through such applications is captured by the DAG structure and is best explained with examples.

As examples of such distributed applications we consider two production financial applications. The first application is a stock trade processing application at a major financial firm; the application consists of 471 nodes and 2073 edges. New stock trade requests arrive at one of the source nodes and flow through the system and exit from the sink nodes as "results". Each intermediate node performs some intermediate processing on the trade request and triggers additional requests at downstream nodes. Nodes may aggregate incoming stock trades or break down a large stock order into smaller requests at downstream nodes. Figure 1 shows the structure of a market data dissemination application that disseminates stock prices and news updates for a company to trading terminals ("desktops") of stock traders. In this case, news items arrive from a number of sources and stock prices are obtained from a variety of exchanges, and this

[1] This assumption is easily relaxed and we employ it for simplicity of exposition.

information is processed, transformed, filtered and/or aggregated and disseminated to any desktop node that has subscribed to information for a particular company. This application has 8970 nodes and 22719 edges and must provide updates at low latency in order for stock traders to make trades based on the latest market news.

Thus, we assume that requests flow through the DAG, with intermediate processing at each node; a request may trigger multiple requests at one or more downstream child nodes, and each node may aggregate requests from upstream parents. As can be seen, such applications are significantly larger and more complex than traditional multi-tier web applications.

We assume that the DAG structure for each application is known a priori (there are automated techniques to derive the DAG structure by observing incoming and outgoing traffic at each node [8]). We assume that each node in the DAG is a black box—i.e., we can observe the incoming and outgoing request streams along its edges and the total node-level utilization but that we have no knowledge of the internals of the software component and how it processes each request. This is a reasonable assumption in practice since IT administrators typically do not have direct knowledge of the application logic inside a software component, requiring us to treat it as a black box. However, administrators have access to request logs that the application components may generate and can also track OS-level resource utilizations on each node.

We assume that there are R different types of requests in the entire distributed application. Each node can receive different types of requests belonging to the R types and can in turn trigger one or more requests of one of the R types at downstream child nodes. Given our black box assumption, the precise dependence of what type of outputs are generated by what set of inputs is unknown (and must be learned automatically by correlating request logs at a parent and a child). Similarly, the precise processing demands imposed by a set of requests are unknown and must also be learned.

Assuming such a data center application, our first problem is to model each application component (i.e., node of the DAG) by capturing the dependence between the incoming workload mix and the request latency, resource utilization, and the outgoing workload. Second, we need to use these *node-level* models to create *system-level* models that capture the behavior of a group of interacting nodes. Third, given such system-level models, we wish to enable rich workload-based what-if analysis of the distributed application. Such an analysis should allow administrators to pose what-if queries to determine the impact of a workload change at a particular node(s) on some other node(s) of the system. A typical what-if query is assumed to contain two parts: (i) the *"if"* part, which specifies the hypothetical workload change, and (ii) the *"what"* part, which specifies the nodes where the impact of this change should be computed. For instance, a volume-based what-if query could ask "what is the impact of doubling the volume of requests seen by source node i on the incoming workload and CPU utilization seen at some downstream node j?" Queries could also be concerned with the impact on latency: "what is the impact of doubling the volume of type B requests at node j on the latency of requests at node i?"

Thus, to design our what-if analysis system, we must address the following three problems: (i) how should we model the dependence between the incoming workload at a node and the request latency, node utilization and the outgoing workload to

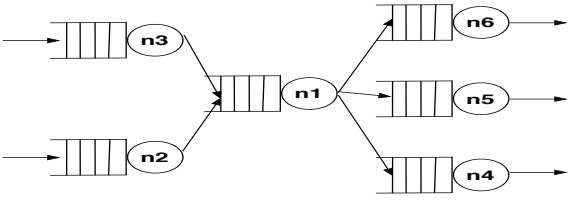


Fig. 2. Modeling a data center application using an open network of queues

downstream nodes? (ii) how should we combine node-level models to create system-level models that capture the aggregate behavior of a group of interacting nodes in the DAG? (iii) what algorithms should be used to efficiently execute a what-if query using these models? From an implementation standpoint, we are interested in a fourth question as well: (iv) How should our system scale to complex data center applications with tens or hundreds of components?

3 Modeling a Data Center Application

In this section, we first present a queuing model for a data center application that allows us to model the utilization and response time of these nodes. We then describe the construction of models to capture the input/output workload dependencies of these nodes. Finally, we explain how these node-level models are composed to construct system-wide models.

3.1 Queuing Theoretic Node-Level Models

Consider the DAG of a data center application with k nodes denoted by $n_1, \ldots, n_k$ and R different type of requests. We model the data center application using an open network of k queues, one for each node with R classes of requests. We model each node as a $M/G/1/PS$ queue i.e. the service times are assumed to have an arbitrary distribution and the service discipline at each node is assumed to be processor sharing (PS). Requests can arrive at a queue from other queues which are its parents in the DAG or in the case of source nodes of the DAG from external sources. For analytical tractability we assume that the distribution of inter-arrival times of requests coming from outside have a poisson distribution. We denote the arrival rates of requests of class r at the queue n_i from outside by $\lambda_{0,i}^r$. We assume that different classes of requests arriving at a queue have different mean service rates. We denote the mean service rate of requests of class r at node i by μ_i^r.

Thus the DAG of a data center application is modeled as an open network of queues as shown in Figure 2. We use the well known queueing theory result called the BCMP theorem [1] to analyze this network of queues. The BCMP theorem states that for such queueing networks the utilization, denoted by ρ_i, of a node n_i, is given by :

$$\rho_i = \sum_{r=1}^{R} \rho_i^r = \sum_{r=1}^{R} \frac{\lambda_i^r}{\mu_i^r} \tag{1}$$

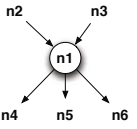


Fig. 3. Node-level model

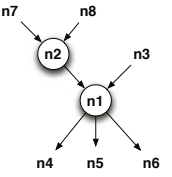


Fig. 4. Model Composition

where ρ_i^r denotes the resource utilization at node n_i due to class r requests, λ_i^r denotes the arrival rate of requests of type r at node n_i and μ_i^r denotes the service rate of requests of type r at node n_i. This equation models the resource utilization of the node as a function of the per-class arrival rate and per-class service rates. Similarly, the average number of requests of type r at node n_i under steady-state, denoted by $\overline{K}_i^r$, is given by:

$$\overline{K}_i^r = \frac{\rho_i^r}{1 - \rho_i} \tag{2}$$

We can now use *Little's Law* [3] to find out the $\overline{T}_i^r$, the average response time of requests of type r at node n_i using Equations 1 and 2:

$$\overline{T}_i^r = \frac{\overline{K}_i^r}{\lambda_i^r} = \frac{1}{\mu_i^r(1 - \rho_i)} \tag{3}$$

This equation models the response time at a node as a function of the total node resource utilization ρ_i and the per-class service rate μ_i^r.

Given a value for the per-class workload, λ_i^r, at a node we can use Equation 1 to find out the utilization ρ_i and then use the computed value ρ_i to find out the response time using Equation 3. The per-class service times μ_r^i is the only unknown in the equations. Since we assume that each node of the data center application is a black-box we need to estimate these unknowns from the available information gathered from monitoring of the node. We assume that request logs at a node contain an entry for each incoming request containing the timestamp and the request string or type of request and that the resource utilization of the node is being periodically monitored using a tool like $iostat$. Given such logs, multiple values of ρ_i and λ_i^r can be collected over time. Since Equation 1 captures the relationship between these $R + 1$ variables, the values of the unknown per-class service rates μ_i^r can be numerically estimated using a regression method such as least squares.

3.2 Workload Models

While queueing theory allows us to model the performance metrics of a node, we also need to capture the relationship between the incoming workload and the outgoing workload of a node.

To understand the node-level workload models that Predico needs to build, consider the node shown in Figure 3. This node n_1 has two parent nodes n_2 and n_3 and three

child nodes n_4, n_5 and n_6. Let $\lambda^r_{2,1}$ and $\lambda^r_{3,1}$ denote the arrival rate of requests of type r from node n_2 and n_3 respectively to node n_1. Similarly, let $\lambda^r_{1,4}, \lambda^r_{1,5}$ and $\lambda^r_{1,6}$ denote the arrival rate of requests of type r at node n_4, n_5 and n_6 respectively from node n_1. Predico needs to build models that capture the workload of each outgoing edge as functions of workload of the incoming edges. Thus, we seek a function for each of $\lambda^r_{1,4}, \lambda^r_{1,5}$ and $\lambda^r_{1,6}$ that expresses them as a function of $\overrightarrow{\lambda_{2,1}}$ and $\overrightarrow{\lambda_{3,1}}$ where $\overrightarrow{\lambda_{i,j}}$ is short-hand for observed rates of various request types on the edge going from node n_i to n_j i.e. $(\lambda^1_{i,j}, \lambda^2_{i,j}, \cdots \lambda^R_{i,j})$. Similarly, we seek functions for each of the other request types:

$$\lambda^w_{1,4} = f^w_{1,4}(\overrightarrow{\lambda_{2,1}}, \overrightarrow{\lambda_{3,1}}) \quad , 1 \leq w \leq R \tag{4}$$

We model workload-to-workload dependencies as piecewise linear functions. Although these dependencies are linear in steady state by the principle of job flow balance [3], [9], we choose piecewise linear modeling to capture the behavior of caches in servers. For instance, a node with a cache can initially be sending a large number of requests to downstream nodes when the cache is empty, but when the cache becomes full, it might serve requests from its cache instead of sending requests to its downstream nodes. This changing dependence of outgoing workload on incoming workload can be captured by two linear functions, one each for when a cache is cold and and when it is hot. To incorporate piecewise linear modeling, we replace the linear model shown in Equation 4 with a piecewise-linear model by dividing the $2R$-dimensional space spanned by $(\overrightarrow{\lambda_{2,1}}, \overrightarrow{\lambda_{3,1}})$ into n hypercube regions. A linear model is then used to capture the relationship in each of these regions independently. Thus, we can rewrite Equation 4 as a set of linear functions one for each region :

$$\lambda^w_{1,4} = \begin{cases} \sum_{r=1}^{r=R} \left(A_1^{w,r}\lambda^r_{2,1} + B_1^{w,r}\lambda^r_{3,1}\right) & \text{if } (\overrightarrow{\lambda_{2,1}}, \overrightarrow{\lambda_{3,1}}) \in Z_1 \\ \sum_{r=1}^{r=R} \left(A_2^{w,r}\lambda^r_{2,1} + B_2^{w,r}\lambda^r_{3,1}\right) & \text{if } (\overrightarrow{\lambda_{2,1}}, \overrightarrow{\lambda_{3,1}}) \in Z_2 \\ \cdots\cdots\cdots\cdots & \\ \sum_{r=1}^{r=R} \left(A_n^{w,r}\lambda^r_{2,1} + B_n^{w,r}\lambda^r_{3,1}\right) & \text{if } (\overrightarrow{\lambda_{2,1}}, \overrightarrow{\lambda_{3,1}}) \in Z_n \end{cases} \tag{5}$$

where Z_i is the i^{th} hypercube region.

Equation 5 relates the outgoing workload to incoming workload, but to use it for computing the outgoing workload $\lambda^w_{1,4}$ for a given value of incoming workload $\overrightarrow{\lambda_{2,1}}$ and $\overrightarrow{\lambda_{3,1}}$ we need to first find the number of regions, n, and the regions themselves, Z_i. We then need to find individual linear functions for each region by computing the weights of the corresponding linear function, $A_1^{w,r}$ and $B_1^{w,r}$. We use a regression analysis technique called multivariate adaptive regression splines (MARS) [6] that automatically fits piecewise linear functions on data. Predico uses the monitoring data that contains multiple measurements of the variables $\overrightarrow{\lambda_{2,1}}, \overrightarrow{\lambda_{3,1}}$ and $\lambda^w_{1,4}$ to give as training data to MARS which finds out the different regions and the linear function in each region.

3.3 Model Composition: From Node-Level to System-Level Models

Predico uses node-level models to construct system-wide models using model composition. Model composition essentially "chains" together multiple node-level models to

compute the workload, resource utilization and response time of a node as a function of one or more ancestor nodes. We illustrate the composition algorithm used by Predico using an example. Consider the sub-graph in Figure 4 that shows a parent node n_2, extending our earlier example in Figure 3. At the node-level, Predico can compute the outgoing workload going from node n_2 to node n_1, $\overrightarrow{\lambda_{2,1}}$, as a set of R piecewise linear functions, one for each request type :

$$\lambda_{2,1}^{w} = f_{2,1}^{w}(\overrightarrow{\lambda_{8,2}}, \overrightarrow{\lambda_{7,2}}) \quad , 1 \leq w \leq R \tag{6}$$

Equation 4 gives the outgoing workload going from node n_1 to n_4 :

$$\lambda_{1,4}^{w} = f_{1,4}^{w}(\overrightarrow{\lambda_{2,1}}, \overrightarrow{\lambda_{3,1}}) \quad , 1 \leq w \leq R \tag{7}$$

Substituting the value of $\overrightarrow{\lambda_{2,1}}$ from Equation 6 into Equation 7 we obtain a "composed model" :

$$\lambda_{1,4}^{w} = f_{1,4}^{w}(\overrightarrow{f_{2,1}}(\overrightarrow{\lambda_{8,2}}, \overrightarrow{\lambda_{7,2}}), \overrightarrow{\lambda_{3,1}}) \quad , 1 \leq w \leq R \tag{8}$$

where $\overrightarrow{f_{2,1}}(\overrightarrow{\lambda_{8,2}}, \overrightarrow{\lambda_{7,2}})$ is shorthand for $(f_{2,1}^{1}(\overrightarrow{\lambda_{8,2}}, \overrightarrow{\lambda_{7,2}}),\ f_{2,1}^{2}(\overrightarrow{\lambda_{8,2}}, \overrightarrow{\lambda_{7,2}}) \ , \ldots \ , f_{2,1}^{R}(\overrightarrow{\lambda_{8,2}}, \overrightarrow{\lambda_{7,2}}))$. Doing so enables the outgoing workload sent from node n_1 to n_4 to be expressed as a function of incoming workload of parent node n_2. This process can be repeated for the outgoing workload going to nodes n_5 and n_6 from node n_1 and can also be recursively extended to nodes that are further upstream from n_2.

Creation of the composed model shown in Equation 8 requires composing the piecewise linear function $f_{1,4}^{w}$ with each of the R piecewise linear functions $f_{2,1}^{w}, 1 \leq w \leq R$. Two piecewise linear functions can be easily composed by composing the individual linear functions in each corresponding region which leads to another piecewise linear function. Thus the composed model shown in Equation 8 is again a piecewise linear function which captures the relation between the outgoing workload of node n_1 and the incoming workload of a parent node n_2.

We can now do a similar composition to find the dependence of the resource utilization of node n_1, denoted by ρ_1, and the response time of requests of type r at node n_1, denoted by $\overline{T}_1^r$ on the incoming workload of parent node n_2 denoted by $\overrightarrow{\lambda_{8,2}}, \overrightarrow{\lambda_{7,2}}$. Substituting Equation 6 into the resource utilization equation given by Equation 1 we get :

$$\rho_1 = \sum_{r=1}^{R} \rho_1^r = \sum_{r=1}^{R} \frac{\lambda_1^r}{\mu_1^r} = \sum_{r=1}^{R} \frac{\lambda_{3,1}^r + \lambda_{2,1}^r}{\mu_1^r} \tag{9}$$

$$= \sum_{r=1}^{R} \frac{\lambda_{3,1}^r + f_{2,1}^r(\overrightarrow{\lambda_{8,2}}, \overrightarrow{\lambda_{7,2}})}{\mu_1^r} \tag{10}$$

which expresses the resource utilization of node n_1 as a function of the incoming workload of node n_2. Similarly, we can substitute from Equation 10 into the response time Equation 3 to express the response time of request type r at node n_1 as a function of the incoming workload of parent node n_2 :

$$\overline{T}_1^r = \frac{1}{\mu_1^r(1-\rho_1)} \tag{11}$$

4 Answering What-if Queries

In this section we describe the three step process used by Predico to answer a given what-if query. The execution of a what-if query is a three step process comprising of: 1) finding the *influence graph* of the given query, 2) creating the node-level models of the nodes in the *influence graph* using the modeling technique described above and 3) using the *change propagation* algorithm to execute the query. We describe the three steps in greater detail below.

4.1 On-the-Fly Model Construction Using the Influence Graph

Since the number of nodes and edges in the DAG may be large in complex applications, it is not economical to precompute all possible node-level models and periodically recompute models that have become invalid due to an actual workload or hardware change. Instead Predico employs a "just-in-time" policy to compute models on-the-fly when a query arrives; only those models that are necessary to answer the query are computed. Models from prior queries are cached and reused if they are still valid. Predico uses the notion of an *influence graph* to determine which models should be constructed to answer a query. Given a what-if query, the *influence graph is the set of all possible paths from the nodes in the "if" part of the query to the nodes in the "what" part*. Basically the influence of a workload change will propagate along all paths from the "if" nodes/edges to the "what" nodes; so the influence graph captures all of the nodes that must be considered to answer the query and other nodes in the DAG can be ignored.

Upon the arrival of a what-if query, Predico first computes the influence graph by generating the set of nodes that lie along all paths from the "if" nodes/edges to the "what" nodes. It then triggers on-demand construction of node-level workload models for all the nodes in the influence graph and node-level resource utilization and response time models for the "what" nodes alone. The use of the influence graph to prune the DAG and the reuse of previously computed models from the model cache enhances the scalability of the system and reduces computational overheads. The influence graph is also crucial for efficient query execution, as we will see in the next section.

4.2 Query Execution Using Change Propagation

After creating the node-level models for the nodes of the influence graph, Predico now needs to "execute" the query. Query execution involves propagating the specified workload change through the influence graph, one node at a time, to compute its final impact on the nodes/edges specified in the "what" part of the query. Once the workload change has been propagated to the nodes in the "what" part, the node-level models can be used to answer the query. Change propagation is equivalent to model composition—instead of directly computing a composed model for the "what" nodes/edges as a function of the

```
Input  : node-level models and influence graph for a what-if query
Output: value of workload/resource usage at "what" nodes/edges
for s In "if" nodes/edges do
    nodeQueue ← s
    while nodeQueue ≠ ∅ do
        currentNode ← Pop(nodeQueue)
        for e In GetIncomingEdges(currentNode) do
            if ValueChanged(e) then
                GetChangedValue(e)
            else
                GetUnchangedValue(e)
        for o In GetOutgoingEdges(currentNode) do
            if o is in the influence graph then
                use node-level model of currentNode to find workload value on o
                SetValue(o)
                ValueChanged(o) = TRUE
        for c In GetChildNodes(currentNode) do
            if c is in the influence graph then
                Push(nodeQueue,c)
```

Algorithm 1. Change Propagation via the Influence Graph

"if" nodes/edges, the propagation algorithm propagates the specified change through the influence graph all the way down to the nodes/edges in the "what" part to achieve the same result.

Predico's change propagation algorithm is described in Algorithm 1. It takes the node-level models and the influence graph, and traverses the influence graph in a breadth first manner. It starts with the nodes/edges in the "if" part and computes the values for the changed workload and then uses the model to compute its impact on the outgoing workload. This process is referred to as propagating the change from the incoming edges of a node to its outgoing edges. To illustrate, consider a query that is interested in estimating the impact of a doubling of the workload for a particular edge. If the original request rate was 10 req/s, then the new workload will be 20 req/s for that edge. This new value is used, along with the *unchanged request rates* for all other edges not impacted by the change, to compute the outgoing request rates for that node.

The algorithm proceeds in a breadth first fashion through the influence graph, starting with the "if" nodes/edges and computing the outgoing workload for each of the "if" nodes. The outgoing workload of a node becomes the incoming workload for downstream node(s), and the change propagation process repeats, one node at a time, in a breadth-first fashion, until the change has propagated to all of the "what" nodes/edges. At this point, the algorithm computes the value of interest at the node by using the node-level models and terminates.

5 Predico Implementation

This section describes WIFQL, a query language that can be used to pose what-if queries to Predico and the implementation details of Predico prototype.

```
query = what_part if_part ;
what_part = "compute" ( simple_compute_part | compound_compute_part );
compound_compute_part = ( simple_compute_part "AND"
                          ( simple_compute_part | compound_compute_part ));
simple_compute_part = ( "cpu utilization" | "spare capacity" | "latency" )
                      "at nodes" node_id {, node_id } ) |
                      "workload on" ( edge_id {, edge_id } );
edge_id = "(" node_id, node_id ")" ;
if_part = "if" ( simple_change_part | compound_change_part );
compound_change_part = ( simple_change_part "AND"
                         ( simple_change_part | compound_change_part) )
                         <EOL>;
simple_change_part = workload_change | hardware_change ;
workload_change = "workload" { "for request class"
                  request_class_id } ( ("at node" node_id ) |
                  ("on edge" edge_id ) ) set_operator value ;
hardware_change = ( "cpu_speed" | "memory" | "disk_speed" )
                  set_operator value ;
set_operator = "*=" | "/=";
```

Fig. 5. The grammar for Predico's What-If Query Language (WIFQL)

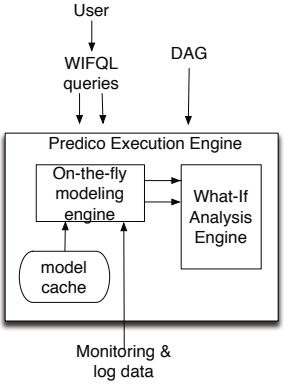


Fig. 6. Predico Architecture

5.1 Posing What-if Queries in Predico

Since the goal of Predico is to enable users to understand the impact of potential workload changes on the system behavior, our system supports a simple query language to enable a rich set of queries to be posed by IT administrators. Any query in our What-If Query Language (WIFQL) has two parts: a *what* part and an *if* part. The *if* part of the query describes the hypothetical change, while the *what* part asks the system to compute the impact of that change on different performance metrics at one or more nodes in the system. As an example of an WIFQL query, consider

```
compute workload on edges (n1, n4), (n1, n5), (n1, n6)
 cpu utilization at nodes  n1, n2
 latency at nodes n1, n2
 if workload on  (n2, n1) *= 2
workload on  (n3, n1) *= 0.5
```

This example query asks the system to compute the impact of a doubling of the workload along the edge going from node n_2 to n_1 and a halving of the workload along the edge going from node n_3 to n_1 on the CPU utilization and latency at nodes n_1 and n_2 and the workload on the edges going from node n_1 to nodes n_4, n_5 and n_6.

Figure 5 describes our query language grammar. As shown, the *if* part allows users to specify hypothetical changes to the workload or changes to the hardware (e.g., a faster CPU). The workload changes, which is the focus of this work, can be specified by identifying one or more edges or nodes in the DAG and indicating a change in volume or a change in the mix of requests; set operators such as multiply and divide can be used to specify relative changes to the current workload, rather than absolute values. The *what* part specifies the performance metrics of interest at particular nodes or edges; several

metrics are supported including resource utilizations, workloads, latencies or spare capacities. As indicated earlier, we assume that the DAG representing the application is known a priori and is used by queries to refer to particular nodes and edges of interest and specify workload changes on these nodes or edges.

5.2 Prototype Implementation

We have implemented a prototype of Predico using Python and the R statistical language to perform what-if analysis in large data center applications. Figure 6 depicts the high-level architecture of Predico.

The Predico frontend is implemented using a python implementation of the lex and yacc parsing tools. It accepts user-posed queries and parses them by using the grammar rules of WIFQL. User-posed queries are then executed by the Predico execution engine, which comprises of two key components; the on-the-fly modeling engine and the what-if analysis engine. The on-the-fly modeling engine first computes the influence graph using a graph API in python and then creates node-level models by using on-the-fly model construction. The modeling engine retrieves data about the workload on the incoming and outgoing edges of the node and the total resource utilization of the node and then invokes an R module for building the node-level models. The R module uses the MARS function present in the MDA package to build piecewise linear node-level workload models and the linear regression function to find the per-class service rates using least squares regression. Next, the what-if analysis engine uses these models to answer ("execute") the query via the change propagation algorithm to propagate the hypothetical workload change through the model and compute its impact on the nodes of interest to the user. The change propagation algorithm is again implemented by using the graph API written in python. The what-if analysis engine stores the node-level models computed by the modeling engine in a model cache that is implemented as three tables in the MySQL relational database engine; one each for storing the weight vectors used in node-level workload models, the regions of the piecewise-linear model and the per-class service rates of a node required in the node-level resource utilization and response-time models.

6 Experimental Evaluation

In this section, we evaluate the performance of Predico by performing experiments on two applications. We first evaluate the accuracy of the analytical node-level resource utilization and response-time models and then the piecewise-linear workload models. We then perform experiments to ascertain the accuracy of system-level models formed by composition. We then employ Predico to perform case studies where we pose what-if queries to Predico and compare the predictions with ground truth values observed in actual experimental data.

6.1 Experimental Setup

We evaluate Predico on two different applications. The first set of applications are from the financial domain and are being used by the data center of a financial institution. The second application is a benchmark e-commerce application.

1. We evaluate our system on traces collected from the two production financial applications described in Section 2. The traces collected from the stock trade processing application contain the total number of requests sent out by every component within every 30 second interval. The traces collected from the market data dissemination application contain data for the number of bytes sent out from every component on each of its outgoing edge, within every 30 second interval. Table 1 lists the characteristics of the traces.

Table 1. Characteristics of Production Traces

Application	#Nodes	#Edges	Duration	Metric	# of Records
Market Data Dissemintation	8970	22719	1 day	outgoing bytes	7763764
Stock Trade Processing	471	2073	4 days	outgoing requests	6060952

2. The second application is the TPC-W benchmark [15] which models an online bookstore application. We implement the TPC-W application as a 2-node Java servlet based application consisting of the front-end server (Tomcat) and a back-end database (MySQL). Notice that this application does not follow our DAG assumption since replies are sent back from the back-end database to the front-end server. Predico is also able to handle such applications that contain cycles between neighboring nodes by considering the two edges of the cycle separately. To implement this application we use a testbed comprising of two virtual machines for performing this experiment. Each virtual machine has a single 2.8 GHz Pentium 4 processor with 1GB memory. We use Tomcat version 5.5.26 and MySQL version 5.1.26 for setting up our TPC-W application. The TPC-W experimental setup allows us to monitor the end-to-end latency and resource utilization values apart from workload values.

6.2 Accuracy of Node-Level Resource Usage and Latency Models

We model the data center application as an open network of queues that lead to Equation 1 which captures the node-level resource utilization and Equation 3 which captures the node-level latency. We validate the accuracy of this queueing model using the TPCW application running on a two server testbed.

We use the *httperf* load generation tool to simulate requests arriving from customers with exponentially distributed inter-arrival times. The TPC-W web application exposes 14 different servlets which a customer visiting the website can invoke. We create a workload comprising of requests to two of these servlets, the *"new products "* and *"execute search"*. We independently vary the arrival rate of requests to both these servlets from 10 to 100 requests per second with increase of 10 requests per second, thus generating a total of 100 arrival rate combinations. For each arrival rate combination, we let the system run for 15 minutes and measure the CPU utilizations at the Tomcat and MySQL server and the end-to-end latency. We use half of the 100 values for estimating

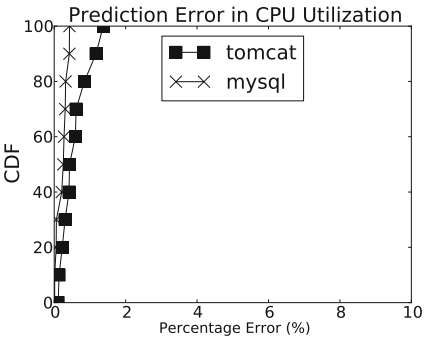


Fig. 7. Node-level Resource Usage model Accuracy

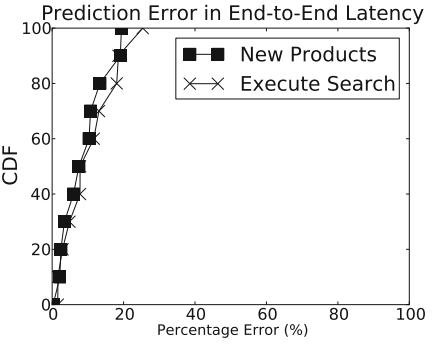


Fig. 8. Node-level Latency model Accuracy

the values of the per-class service rates on each of the 2 nodes and then use these values to predict the per-node resource utilizations and per-node per-class response time for the other half. The per-node response times are summed up to get the end-to-end latency. We compute the prediction errors by comparing the predictions of the node-level models with the values observed during the experiments. Figure 7 and Figure 8 shows the distribution of prediction errors in terms of percentage relative error in predicting the resource utilization and latency respectively.

By using an open network of queue modeling, we are able to predict node-level CPU utilization to within 2% of the actual value. The median prediction error for response time using our modeling approach is less than 10%.

6.3 Accuracy of Node-Level Workload Models

We evaluate the accuracy of using piecewise-linear functions created by using MARS to model the relationship of the outgoing workload of a node with the incoming workload of the node. We use the traces collected from the two applications to create these models and then ascertain the accuracy of these models.

For each of the two applications, we selected each component in turn and extracted the data for the workload on its incoming edges and outgoing edges. We then use MARS to estimate a function which expresses the workload on each outgoing edge of a component as a piecewise linear function of the workload on all the incoming edges on the component. We evaluate the accuracy of the piecewise linear model in predicting the workload on each outgoing edge of this component. Cross-validation was used to measure the prediction accuracy; we divide the trace data for the selected component into training windows of 1 hour each and compute a model using MARS for each window for each outgoing edge. We then use each model to predict the data points outside of the window it was trained on; the deviations between the predicted and actual values were measured. We use the root mean square (RMS) error as a metric of error; we divide the RMS error by the range of actual values to report the results in normalized RMS error (%). The average normalized RMS error for the models of all the outgoing edges of a component is taken as the error for that component. We depict the errors for all

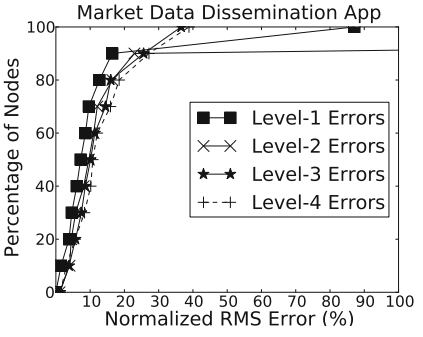


Fig. 9. Composed modeling for Market Data Dissemination App

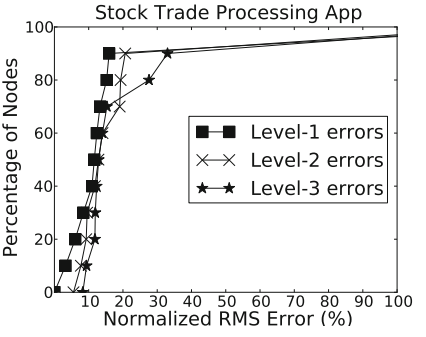


Fig. 10. Composed Modeling for Stock Trade Processing App

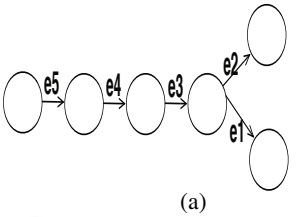
the components of the two applications using CDF curves that show the percentage of components that have errors below a certain value. Figure 9 shows the errors for the market data dissemination application while Figure 10 shows the errors for the stock trade processing application. The curve labeled "Level-1" errors shows the CDF for the errors. We describe the concept of levels and the description about the "Level-2" and "Level-3" curves later in this section. The CDF curves indicate that the workload-to-workload models of 70% of the components have errors less than 10% in the case of the market data dissemination application while models for 80% of the components have errors less than 15% in the case of the stock trade processing application.

Our experimental results show that piecewise linear modeling provides accurate models of node-level workload for production data center applications.

6.4 Accuracy of System-Level Models with Increasing Composition Depth

We evaluate the accuracy of system-level models created by composing multiple node-level models. Composition of multiple node-level models leads to an accumulation of the error terms. We conduct experiments to measure the increase in error with composing increasing number of node-level models. We again use the traces from the two financial applications to evaluate the accuracy of system-level models. We reuse the node-level models of each component built for validating the accuracy of node-level workload models in the previous section for this experiment.

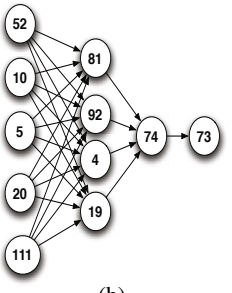
We select each component and compose its node-level workload model with that of its ancestor components to express the outgoing workload of this component as a function of the incoming workload of its ancestors. By using composition repeatedly we successively construct models expressing workload of a component as a function of its ancestors at different levels. Level 1 model is built between the outgoing workload of a component and its incoming workload. Level 2 model is built between the outgoing workload of a component and the incoming workload of its immediate parents. Similarly level i model is built between the component and its ancestors that are reachable in $(i - 1)$ edges. We compute the average normalized RMS error of each component by computing the average normalized RMS error for the models of all the outgoing edges



(a)

Prediction Source	Prediction Target			
	e1	e2	e3	e4
e3	2.4%	2.6%		
e4	4.7%	3.05%	4.39%	
e5	4.67%	2.95%	4.33%	0.79%

(c)



(b)

Prediction Source	Prediction Target					
	81	19	74	73	4	92
111, 20, 5, 101, 52	12.47 %	12.45%	10.2%	9.28%	12.5 %	12.54%
19, 4, 92, 81			10.63%	9.46 %		
74				3.34%		

(d)

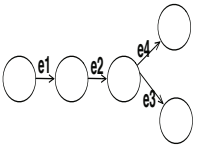
Fig. 11. Prediction Errors of composed modeling on different topologies

of the component using cross-validation and then averaging the errors. Figures 9 and 10 show the CDF of normalized RMS errors for each level for the two applications. The CDF curve drops with increasing levels implying that the errors increase as we predict the workload of a component using ancestors higher up the component in the graph. Inspite of the increasing errors with increasing levels, the errors remain tolerable; for the Market Data Dissemination application even at level 4 the prediction errors for 80% of the components are less than 20%, while for the Stock Trade Processing at the level of 3 for 75% of the components the errors are less than 20%.

Our results on using composition to create system-level models on the traces collected from the two production applications reveal that even with increasing composition depth, the system-level models are effective in predicting workload.

6.5 Accuracy of System-Level Models with Varying Topology

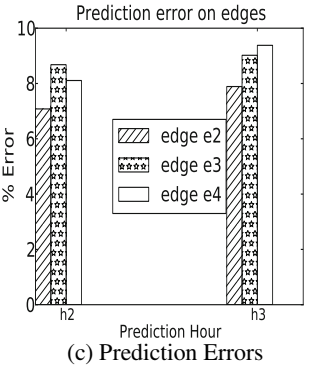
The node-level models can be composed in a number of ways to create a system-level model depending on the topology of the DAG. We perform experiments to ascertain the prediction accuracy of composed models under different topologies. For this experiment we select some subgraphs in the DAGs for the two applications. We select subgraphs that correspond to three topologies-*chain, split and join*. These topologies correspond to different ways in which the components can interact with one another in an application: (i) in the chain topology, each component receives requests from a single upstream component, (ii) in the split topology, a component can send requests to multiple downstream components and (iii) in the join topology, a component can receive requests from multiple upstream components. For each subgraph, we create node-level models for each component and then use composition to create models to predict the workload on each outgoing edge of the subgraph. We measure prediction



(a) Subgraph 1

Prediction Hour	Edge
	e1
$h1$	A
$h2$	$2A$
$h3$	$2.5A$

(b) Subgraph 1 Average Workload



(c) Prediction Errors

Fig. 12. What-If case study on Market Data Dissemination Application

errors in predicting workload of each outgoing edge as a function of incoming workload of its ancestors at increasing levels.

Figures 11(a) and 11(b) show the subgraphs that we choose for this experiment. Figures 11(a) is from the Market Data Dissemination application and Figure 11(b) is a subgraph from the Stock Trade Processing application. Figure 11(a) illustrates the chain and split topologies, while figure 11(b) is an example of a join and split topology. Tables 11(c) and 11(d) show how the errors of the composed models vary as we predict the workload on various edges/nodes of the graphs. For the subgraphs selected from the market data dissemination application the prediction errors on all edges are within 5% while for the subgraph selected from the stock trade processing application the prediction errors are within 13%.

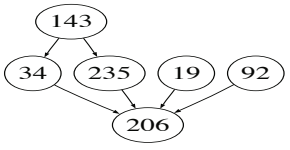
The errors reveal that Predico's composition based modeling technique performs well even in case of complex application topologies.

6.6 Workload-Only What-If Analysis Case Study

We create use-case scenarios to illustrate how Predico can be used in practice and evaluate its performance in answering what-if questions which commonly arise in large-scale applications. In this section, we pose workload-related what-if questions; we choose subgraphs from the market data dissemination application and the stock trade processing application and use Predico to predict the impact of workload changes on source nodes at the workload on the other edges of the subgraphs.

We choose one subgraph each from the market data dissemination application and the stock trade processing application. The first subgraph has 1 source node while the other subgraph has 3 source nodes.

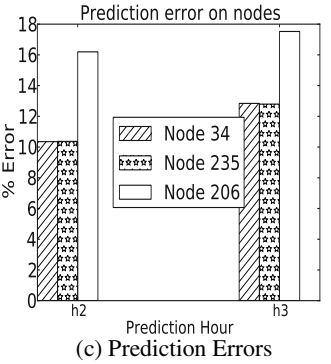
The topology of the first subgraph is shown in Figure 12(a). On this subgraph we pose the query: *"what happens to the workload on downstream edges of subgraph 1 if the outgoing workload of the single source node increases by 2 and 2.5 times the current value"*. We examine the application traces and find periods of 1 hour duration each, h_1, h_2 and h_3, such that the outgoing workload from the source node increases by 2 and 2.5 times the workload in h_1 in the hours h_2 and h_3 respectively. Predico uses the trace



(a) Subgraph 2

Prediction Hour	Node	
	143	19,92
h1	A	B
h2	1.12 A	1.5 B
h3	1.8 A	1.65 B

(b) Subgraph 2 Average Workload



(c) Prediction Errors

Fig. 13. What-If case study on Stock Trade Processing Application

from hour h_1 and then predicts the workload values in hours h_2 and h_3. We compare the ground truth value of the workload seen in the two hours and compare Predico's predictions to compute the errors. Figure 12(c) plots the errors on all the downstream edges in terms of the normalized RMS error for each of the two changes mentioned in the what-if question.

The topology of the second subgraph is shown in Figure 13(a). On this subgraph we pose the query: *"what happens to the workload on downstream nodes a) if the workload on the source nodes 143, 19 and 92 becomes 1.12, 1.5 and 1.5 times respectively the current value b) if the workload on the source nodes 143, 19 and 92 becomes 1.8, 1.65 and 1.65 times respectively the current value times"*. We compare Predico's prediction with ground truth values observed in the traces to compute the errors. Figure 13(c) plots the errors for the two queries for the downstream nodes in terms of normalized RMS error.

The trace collected from the stock trade processing application only contain the requests going out of each node and we assume that these requests are equally distributed among all its outgoing edges. Similarly, in the case of the market data dissemination application, the traces contain the bytes sent out on each edge and we assume that the number of bytes are an approximation of the number of requests. *We note that even under these simplifying assumptions, Predico is able to make predictions with errors between 8% and 18%.*

7 Related Work

A number of recent efforts have focused on building systems for performing what-if analysis on various distributed systems. The design and implementation of a self-predicting cluster-based storage system is presented in [13]. The approach, however, involves intrusive instrumentation of the system that is not feasible in production environments. WISE [12] is a system for answering what-if deployment and configuration questions for content distribution networks (CDN). WISE, however, only answers questions related to network latency and does not consider the server processing within data centers.

Apart from systems that are directly aimed at performing what-if analysis, a number of modeling techniques have been proposed that predict the performance of the system and can be employed for answering what-if questions about the system. Most of these techniques are aimed at multi-tiered systems. A number of these techniques use queuing models to predict the response time and resource utilization of such applications [16],[2],[5]. Similar to our approach, least squares is used to parameterize the queuing models in [17]. Similar to our automatic model derivation, the authors of [7] also automatically derive node-level models to capture relationships between workload; their technique is based on linear models while we have used a queuing-network modeling based approach. In [10], nonstationarity in workloads is utilized to derive models for predicting the resource utilization and response time of an application as a function of workload volume and workload mix. The modeling approach proposed in [11] creates "profiles" for the different components of a distributed application to model the resource demands placed by the components under different workloads on the underlying hardware. IRONModel [14] proposes a modeling architecture for creating robust models. The models are used for answering what-if questions about the impact of reconfigurations on the response time and throughput of a large storage system. In contrast to these systems, Predico is aimed at large-scale systems and enables easily modeling arbitrary distributed applications by joining together individual node-level models. Modellus [4] also uses composition of models to model data center applications. Modellus, however, models workload-to-workload interactions only while Predico looks at response time models as well. Also, Modellus is only a modeling framework, while Predico combines modeling with a full-scale what-if analysis system.

8 Conclusions

Data center operators often need to ascertain the impact of unseen workload changes on large distributed applications. Predicting how a certain change in workload will influence complex data center applications is a challenging problem that needs automation. In this paper we presented Predico, a system which enables the user to perform "what-if" analysis on large distributed applications. Predico is non-intrusive and only uses commonly available monitoring data to construct models and uses a new change propagation technique to estimate the impact of specified workload changes.

We modeled a large-scale data center application as an open network of queues to derive resource utilization, latency and workload models. We used traces from two large production applications from data centers of a major financial institution and data from synthetic enterprise applications to evaluate the efficacy of Predico's what-if modeling framework. Our experimental evaluation validated the accuracy of the node-level resource utilization, response time and workload models and then showed how Predico enables what-if analysis in two different applications.

Acknowledgements. This reseach was supported in part by NSF grants CNS-0855128, CNS-0916972, CNS-0720616 and OCI-1032765.

References

1. Baskett, F., Mani Chandy, K., Muntz, R.R., Palacios, F.G.: Open, closed, and mixed networks of queues with different classes of customers. J. ACM 22(2), 248–260 (1975)
2. Bennani, M.N., Menascé, D.A.: Resource allocation for autonomic data centers using analytic performance models. In: ICAC, Washington, DC, USA, pp. 229–240 (2005)
3. Denning, P.J., Buzen, J.P.: The operational analysis of queueing network models. ACM Comput. Surv. 10, 225–261 (1978)
4. Desnoyers, P., Wood, T., Shenoy, P., Patil, S., Vin, H.: Modellus: Automated Modeling of Complex Internet Data Center Applications. Technical report, UMass CS (2009)
5. Diao, Y., Hellerstein, J.L., Parekh, S.S., Shaikh, H., Surendra, M., Tantawi, A.N.: Modeling differentiated services of multi-tier web applications. In: MASCOTS, pp. 314–326 (2006)
6. Friedman, J.H.: Multivariate adaptive regression splines. The Annals of Statistics 19(1) (1991)
7. Jiang, G., Chen, H., Yoshihira, K.: Discovering Likely Invariants of Distributed Transaction Systems for Autonomic System Management. In: ICAC, Dublin, Ireland, pp. 199–208 (June 2006)
8. Kind, A., Hurley, P., Massar, J.: A Light-Weight and Scalable Network Profiling System. ERCIM News 60 (2005)
9. Menascé, D.A., Almeida, V.A.F.: Capacity planning for Web performance: metrics, models and methods. Prentice-Hall, Inc., Upper Saddle River (1998)
10. Christopher, S., Terence, K., Alex, Z.: Exploiting nonstationarity for performance prediction. In: EuroSys, pp. 31–44 (2007)
11. Stewart, C., Shen, K.: Performance Modeling and System Management for Multi-component Online Services. In: Proc. USENIX Symp. on Networked Systems Design and Implementation (NSDI) (May 2005)
12. Tariq, M., Zeitoun, A., Valancius, V., Feamster, N., Ammar, M.: Answering what-if deployment and configuration questions with wise. SIGCOMM Comput. Commun. Rev. 38(4), 99–110 (2008)
13. Thereska, E., Abd-El-Malek, M., Wylie, J.J., Narayanan, D., Ganger, G.R.: Informed data distribution selection in a self-predicting storage system. In: ICAC, pp. 187–198. IEEE Computer Society, Washington, DC (2006)
14. Thereska, E., Ganger, G.R.: Ironmodel: robust performance models in the wild. SIGMETRICS Perform. Eval. Rev. 36(1), 253–264 (2008)
15. TPC. The tpcw benchmark, http://www.tpc.org/tpcw/
16. Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., Tantawi, A.: An Analytical Model for Multi-tier Internet Services and Its Applications. In: Proc. of the ACM SIGMETRICS Conf., Banff, Canada (June 2005)
17. Zhang, Q., Cherkasova, L., Smirni, E.: A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In: ICAC, Washington, DC, USA (2007)

GreenWare: Greening Cloud-Scale Data Centers to Maximize the Use of Renewable Energy

Yanwei Zhang[1], Yefu Wang[1], and Xiaorui Wang[1,2]

[1] Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville, TN 37996
[2] Department of Electrical and Computer Engineering
The Ohio State University, Columbus, OH 43210
{yzhang82,ywang38}@eecs.utk.edu, xwang@ece.osu.edu

Abstract. To reduce the negative environmental implications (e.g., CO_2 emission and global warming) caused by the rapidly increasing energy consumption, many Internet service operators have started taking various initiatives to operate their cloud-scale data centers with renewable energy. Unfortunately, due to the intermittent nature of renewable energy sources such as wind turbines and solar panels, currently renewable energy is often more expensive than brown energy that is produced with conventional fossil-based fuel. As a result, utilizing renewable energy may impose a considerable pressure on the sometimes stringent operation budgets of Internet service operators. Therefore, two key questions faced by many cloud-service operators are 1) how to dynamically distribute service requests among data centers in different geographical locations, based on the local weather conditions, to maximize the use of renewable energy, and 2) how to do that within their allowed operation budgets.

In this paper, we propose GreenWare, a novel middleware system that conducts dynamic request dispatching to maximize the percentage of renewable energy used to power a network of distributed data centers, subject to the desired cost budget of the Internet service operator. Our solution first explicitly models the intermittent generation of renewable energy, e.g., wind power and solar power, with respect to varying weather conditions in the geographical location of each data center. We then formulate the core objective of GreenWare as a constrained optimization problem and propose an efficient request dispatching algorithm based on linear-fractional programming (LFP). We evaluate GreenWare with real-world weather, electricity price, and workload traces. Our experimental results show that GreenWare can significantly increase the use of renewable energy in cloud-scale data centers without violating the desired cost budget, despite the intermittent supplies of renewable energy in different locations and time-varying electricity prices and workloads.

1 Introduction

Recent years have seen the rapid growth of large and geographically distributed data centers deployed by Internet service operators to support various services

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 143–164, 2011.

such as cloud computing. As an effort to deal with the increasingly severe global energy crisis, reducing the high energy consumption of those cloud-scale data centers has become a serious challenge. For example, some cloud-service data centers are termed as *mega data centers*, because they host hundreds of thousands of servers and can draw tens to hundreds of megawatts of power at peak [22]. It has also been reported that in a conservative estimation, Google hosts more than 500,000 servers in its data centers distributed in different locations and consumes at least 6.3×10^5 MWh in total annually [39]. Therefore, minimizing the energy consumption of cloud-scale data centers has recently received a lot of research attention (e.g., [20,17,29,13,47,18]).

In addition to high electricity bills, the enormous energy consumption of cloud-scale data centers can also lead to negative environmental implications (*e.g.*, CO_2 emission and global warming), due to their large carbon footprints. The reason is that most of the produced electricity around the world comes from carbon-intensive approaches, *e.g.*, coal burning [29]. Such energy produced with conventional fossil-based fuel is commonly referred to as brown energy. Therefore, to mitigate the negative environmental implications caused by the rapidly increasing energy consumption, many Internet service operators have started taking various initiatives to operate their cloud-scale data centers with renewable (or *green*) energy. In contrast to brown energy, green (or clean) energy is normally generated from renewable energy sources, such as wind turbines and solar panels, and is thus more environmentally friendly. For example, major Internet service operators, *e.g.*, Google, Microsoft, and Yahoo!, have all started to increasingly power some of their data centers using renewable energy, and so reduce their dependence on brown energy [38,4,43]. Therefore, since data centers in different geographical locations may have different availabilities of renewable energy depending on the local weather conditions, it is important for cloud-service operators to dynamically distribute service requests among different data centers to maximize the use of renewable energy.

Unfortunately, due to the intermittent nature of renewable energy sources such as wind and sunlight, currently renewable energy can be often more expensive to produce than brown energy [2,11]. While some data centers are trying to build their own wind farms or solar photovoltaic (PV) power plants, due to concerns such as expensive facility investments and management, many Internet service operators choose to work with professional renewable energy producers and utilize the green energy integrated into the power grid. For example, Google has recently purchased 20 years' worth of wind energy from an Iowa wind farm, which will be sufficient to power several of its data centers in Oklahoma [12]. Google also invested $100 million in the Shepherds Flat Wind Farm in Oregon to generate 845 megawatts of green power, which will be sold directly to Southern California Edison's power grid. As a result of its higher production costs, renewable energy coming from the grid can be more expensive than brown energy. For example, the industrial electricity price for solar energy can be 16.14 cents per KWh in a sunny climate and 35.51 cents per KWh in a cloudy climate [8]. In contrast, the wholesale brown energy price can be around 6 cents per KWh [39]. The Los Angeles

Department of Water and Power also estimates that the extra cost for green energy is at least 3 cents per KWh [5]. Therefore, utilizing renewable energy may impose a considerable pressure on the sometimes stringent operation budgets of Internet service operators, as the electricity cost of operating data centers has become a significant portion, *e.g.*, 20% or more of the monthly costs of those enterprises [22]. Hence, a key dilemma faced by many service operators is how to exploit renewable energy to the maximum degree that is allowed by their monthly operating budgets.

In this paper, we propose *GreenWare*, a novel middleware system that conducts dynamic request dispatching to maximize the percentage of renewable energy used to power a network of distributed data centers, subject to the desired cost budgets of Internet service operators. We first model the intermittent generation of renewable energy, *i.e.*, wind power and solar power, with respect to the varying weather conditions in the geographical location of each data center. For example, the available wind power generated from wind turbines is modeled based on the ambient wind speed [35,9], while the available solar power from solar plants is estimated by modeling the maximum power point on irradiance (*i.e.*, solar energy per unit area of the solar panel's face) and temperature [31,41]. Based on the models, we formulate the core objective of GreenWare as a constrained optimization problem, in which the constraints capture the Quality of Service (QoS, *e.g.*, response time) requirements from customers, the intermittent availabilities of renewable energy in different locations, the peak power limit of each data center, and the monthly cost budget of the Internet service operator. We then transfer the optimization problem into a linear-fractional programming (LFP) formulation for an efficient request dispatching solution with a polynomial time average complexity.

Specifically, this paper makes the following major contributions:

- We propose a novel GreenWare middleware system in operating geographically distributed cloud-scale data centers. GreenWare dynamically dispatches incoming service requests among different data centers, based on the time-varying electricity prices and availabilities of renewable energy in their geographical locations, to maximize the use of renewable energy, while enforcing the monthly budget determined by the Internet service operator.
- We explicitly model renewable energy generation, *i.e.*, wind turbines and solar panels, with respect to the varying weather conditions in the geographical location of each data center. As a result, our solution can effectively handle the intermittent supplies of renewable energy.
- We formulate the core objective of GreenWare as a constrained optimization problem and propose an efficient request dispatching solution based on LFP.
- We evaluate GreenWare with real-world weather, electricity price, and workload traces. Our experimental results show that GreenWare can significantly reduce the dependence of cloud-scale data centers on fossil-fuel-based energy without violating the desired cost budget, despite the intermittent supplies of renewable energy and time-varying electricity prices and workloads.

The rest of the paper is organized as follows. Section 2 introduces the overall architecture of the proposed GreenWare middleware system. Section 3 presents the modeling and formulations of GreenWare. Section 4 discusses the simulation strategy. Section 5 evaluates GreenWare with real-world traces. Section 6 reviews the related work and Section 7 concludes the paper.

2 GreenWare Architecture

In this section, we provide a high-level description of the proposed GreenWare system. GreenWare dynamically conducts request dispatching among data centers in order to maximize the percentage of renewable energy used to power a network of distributed data centers, based on the time-varying electricity prices and availabilities of renewable energy in their geographical locations. In the meantime, GreenWare guarantees the desired QoS for customers and effectively maintains the electricity bill within a cost budget determined by the Internet service operators.

In this work, we assume that a network of distributed data centers share a common cost budget, which can be determined by the Internet service operator periodically in each budgeting period (*e.g.*, a month). A local optimizer is assumed to be present in each single data center in the network to dynamically adjust the number of active servers to minimize the power consumption of the data center, while maintaining a desired level of QoS based on a QoS model detailed in Section 3.2. We also assume that the short-term weather conditions (*e.g.*, in one hour) and the configurations of wind turbines and solar panels of each data center are available. As shown in Figure 1, *GreenWare* is a centralized system that manages a data center network for maximizing the use of renewable energy within the cost budget. While such a centralized architecture is commonly used in the management of data center networks [29,40,39], GreenWare can be extended to work in a hierarchical way, which is our future work. Similar to [13,34,48], we use one month as the budgeting period and one hour as the period for GreenWare to be invoked and conduct the request dispatching operation.

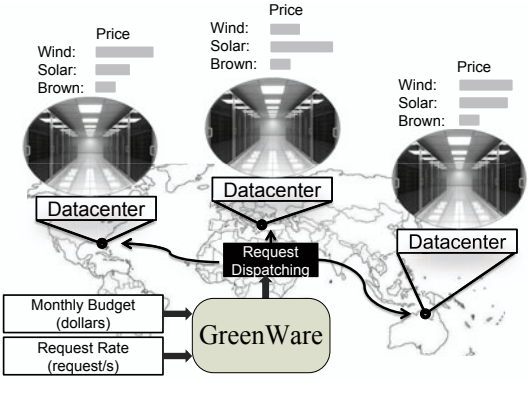


Fig. 1. Proposed GreenWare system for distributed cloud-scale data center networks

In every invocation period, GreenWare performs three steps: First, GreenWare computes the hourly budget based on the monthly cost budget from the service operator and the electricity cost already consumed in the previous invocation periods, as well as the observations of the workload's historical behaviors in the same hours in the past (*e.g.*, last two weeks) as discussed in Section 4.3. Second, based on the time-varying electricity prices and availability of renewable energy at each data center, with respect to the varying weather conditions in their geographical location (*e.g.*, irradiance, temperature, and wind speed), GreenWare runs the optimization algorithm in Section 3 to compute the desired request dispatching (*e.g.*, the fraction of workload allocated to each data center) such that (1) the overall percentage of renewable energy used to power a network of distributed data centers is maximized within budget constraints; (2) the total electricity cost is below the budget of the current hour; and (3) the application-level QoS (*e.g.*, desired response time) for customers is guaranteed. Third, GreenWare redirects the incoming requests among data centers based on the determined request dispatching in Step (2), using the dynamic request routing mechanism already deployed in cloud-scale data center networks. Note that dynamic request routing has already been implemented by many Internet service operators to map requests to servers, for the purposes of customer QoS guarantees and fault-tolerance [39].

3 Design Methodology of GreenWare

In this section, we first present the problem formulation of the optimization objective of GreenWare. We then introduce the adopted performance and server power models, as well as the wind power model and solar power model. Finally, we discuss our request dispatching solution. Note that we focus mainly on wind power and solar power in this work because there exists meteorological data [6] for us to simulate their intermittent availabilities in distributed data centers. GreenWare can be applied to other types of renewable energy, such as hydro-electric and geothermal, if their corresponding meteorological data is also available.

3.1 Problem Formulation

We first introduce the following notation. N data centers are operated in a cloud-scale data-center network. The i^{th} data center consumes pW_i kilowatts of wind energy, pS_i kilowatts of solar energy and pB_i kilowatts of brown energy, respectively. The total power consumption p_i (*i.e.*, $p_i = pW_i + pS_i + pB_i$) of the i^{th} data center should not exceed the peak power limit Ps_i of the data center. The intermittent availabilities of the renewable energy in the local power market of the i^{th} data center are denoted as PW_i and PS_i. In particular, PW_i and PS_i are the estimated wind power output from the wind farm and the maximum solar power output from the solar plant, respectively. The corresponding wind farm and solar plant are assumed to be the renewable energy sources for the local

power market of the i^{th} data center. PrW_i, PrW_i and PrB_i are the current electricity prices of the three types of energy from the power market of the i^{th} data center, respectively. The whole system has an incoming workload of λ requests per hour. Our algorithm allocates the i^{th} data center with a workload of λ_i requests per hour to maximize the percentage of renewable energy used, depending on the wind and solar power models based on local weather conditions (presented in Sections 3.3 and 3.4), within the allocated cost budget Cs. The average response time of the i^{th} data center is R_i and the corresponding response time set point is Rs_i.

Given a workload of λ requests per hour, the optimization goal is to dynamically choose a request dispatching strategy such that the i^{th} data center is assigned λ_i requests to maximally use renewable energy to power the data center network within the cost budget. Specifically, in order to maximize the overall renewable energy usage of all the N data centers, x_i percentage of wind power and y_i percentage of solar power out of the total power consumption p_i by the i^{th} data center will have to be determined. Then, z_i percentage of the total power consumption is supplemented in the form of brown energy. It is clear that $z_i = 1 - x_i - y_i$. In summary, the optimization problem can be expressed as follows.

Problem 1:

$$Maximize: \frac{\sum_{i=1}^{N}(pW_i + pS_i)}{\sum_{i=1}^{N}(pW_i + pS_i + pB_i)} \tag{1}$$

subject to

$$\sum_{i=1}^{N} \lambda_i = \lambda \tag{2}$$

$$\lambda_i \geq 0 \tag{3}$$

$$R_i \leq Rs_i \tag{4}$$

$$0 \leq pW_i \leq PW_i \tag{5}$$

$$0 \leq pS_i \leq PS_i \tag{6}$$

$$0 \leq pW_i + pS_i + pB_i \leq Ps_i \tag{7}$$

$$\sum_{i=1}^{N}(PrW_i \cdot pW_i + PrS_i \cdot pS_i + PrB_i \cdot pB_i) \leq Cs \tag{8}$$

Specifically, pW_i, pS_i, pB_i, PW_i, and PS_i (in KW) will be numerically the same as energy (in KWh) since the invocation period used in this work is assumed to be one hour. In order to solve the optimization **Problem 1**, it is important to model the variables pW_i, pS_i and pB_i as functions of λ_i, x_i and y_i. It is clear that

$$pW_i = x_i \cdot p_i; pS_i = y_i \cdot p_i; pB_i = z_i \cdot p_i \tag{9}$$

where $x_i + y_i + z_i = 1$.

Thus, in the following we first model the power consumption p_i and the average response time R_i with the request distribution rate λ_i for the i^{th} data center. We then model the availabilities of wind power and solar power, *i.e.*, PW_i and PS_i, respectively, based on the weather condition of the i^{th} data center, *e.g.*, irradiance, temperature, and wind velocity. We discuss an efficient solution design for **Problem 1** in Section 3.5.

3.2 Response Time and Power Models

Queueing theory is widely used to model the performance of a web server [13,14]. In this paper, we use the M/M/n queueing model in queueing theory [40] to model the response time for a data center. The average response time of the requests to a web server consists of two portions: (1) the average waiting time that the requests spend in a queue waiting to be serviced and (2) the service time, *i.e.*, $\frac{1}{\mu}$, given the service rate μ of the data center. Specifically, the average waiting time for a data center with n active servers can be expressed as $\frac{1}{n \cdot \mu - \lambda} \cdot P_Q$, where P_Q represents the probability that the incoming requests need to wait in a queue to be serviced. Furthermore, we assume that all the active servers will likely keep busy, *i.e.*, running at close to 100% utilization, because a local optimizer running in each data center minimizes the number of active servers. Hence, without loss of generality, P_Q is assumed to be 1, since all the active servers are assumed to be running at close to 100% utilization. The same assumption is used in existing solutions on electricity cost minimization for data centers [40]. Therefore, we have

$$R_i = \frac{1}{\mu_i} + \frac{1}{n_i \cdot \mu_i - \lambda_i} \tag{10}$$

where n_i is the number of active servers and μ_i is the average service rate of a single server, *i.e.*, the number of requests the server is able to process in a unit time, in the i^{th} data center .

As discussed in Section 2, we assume that a local optimizer runs in every data center and dynamically adjusts the number of active servers to provide a desired level of QoS (*e.g.*, response time) with the least number of servers. As a result, given a request rate of λ_i and a desired response time Rs_i of the i^{th} data center, the number of desired active servers n_i can be derived from equation (10). Thus, we have $p_i = n_i \cdot sp_i$, where sp_i is the average power consumption of a single server in the i^{th} data center. Although the power consumption of a server is usually a function of the utilization of the server, we assume that sp_i is constant because when the local optimizer minimizes the number of active servers, all the

servers remaining active will likely run close to a 100% utilization. Thus, the utilization will be approximately the same. It is then clear that a linear server power model based on the incoming work rate λ_i for the i^{th} data center can be derived, *i.e.*, $p_i = f(\lambda_i)$, where $f(\lambda_i)$ is a linear function.

3.3 Wind Power Model

The number of wind turbine installations is rapidly growing worldwide. It is expected that the US can get 20% of its electricity from wind energy by the year 2030 [25,37]. It has been shown that wind power generated by wind turbines in a wind farm can be modeled as a function of the actual wind speed [35,9]. For example, the wind power output p_{wind} by a single wind turbine, with respect to a wind speed of v, can be approximated as follows

$$p_{wind} = \begin{cases} 0 & v < v_{in}, v > v_{out} \\ p_r \cdot \frac{v - v_{in}}{v_r - v_{in}} & v_{in} < v < v_r \\ p_r & v_r < v < v_{out} \end{cases}$$

where v_r, p_r are the rated speed and power of the wind turbine and v_{in}, v_{out} are cut-in and cut-out wind speeds. Specifically, the cut-in speed is the wind speed at which the turbine first starts to rotate and generate power, *e.g.*, a typical value between 3 and 4 meters per second; while the cut-out speed is employed by the braking system to bring the rotor to a standstill to eliminate the risk of damaging the turbine rotor due to the continuously rising wind speed, *e.g.*, a cut-out speed of usually around 25 meters per second.

In the case of a large-scale wind power generation farm, *e.g.*, one consisting of a large number m_w of wind-turbines, the overall wind power output is estimated as the sum of the power output values sampled at different turbines for simplicity [21]

$$PW = \sum_{k=1}^{m_w} p_{wind}^k$$

where p_{wind}^k is the power output from the k^{th} wind turbine with respect to the wind speed v, with t05e assumption that the wind turbines have the same wind speed in the same wind farm.

3.4 Solar Power Model

The worldwide photovoltaic (PV) power capacity installation grows in a nearly exponential way, despite their relatively high cost [41]. In this work, we model the solar power generated by solar plants with respect to the varying weather conditions, such as irradiance and temperature, based on a single diode equation [41,36]. In particular, the single diode equation has been widely used to simulate the available electrical power generated from a single PV panel. Specifically, the resulting current-voltage characteristic of a PV panel is

$$i = I_{ph} - I_o \cdot (e^{\frac{v + i \cdot R_s}{n_s \cdot V_{th}}} - 1) - \frac{v + i \cdot R_s}{R_{sh}} \tag{11}$$

where I_{ph} is the photo-generated current while I_o is the dark saturation current with respect to the ambient weather pattern. Moreover, the single-diode model takes into account both the series and parallel (shunt) resistance of the PV panel, referred to as R_s and R_{sh}, respectively. V_{th} is the junction thermal voltage, *i.e.*, $V_{th} = k \cdot T/q$, where k is Boltzmann's constant, q is the charge of the electron and T is the ambient temperature. n_s is the number of the solar cells in the PV panel connected in series, *e.g.*, $n_s = 72$ in BP-MSX 120 panels [1].

To show the solar power output from PV panels with respect to the varying weather conditions (*e.g.*, irradiance and temperature), equation (11) can then be transformed as equation (12) by including these two key factors, *i.e.*, irradiance and temperature [41]. In particular, it has been demonstrated that the dark saturation current of I_o just varies with the ambient temperature T, independent on the irradiance condition G [41,16]. Furthermore, for a high-quality solar cell, it typically has a low series resistance R_s but a high parallel resistance R_{sh}. As a result, the solar model in this work only takes into considerations the series resistance (*i.e.*, $R_{sh} = \infty$), which is consistent with the prior study [31]. We thus have the fact that I_{ph} can be approximated by I_{sc} for simplicity, where I_{sc} is the short-circuit current. In particular, I_{sc} is directly proportional to the irradiance as well as the ambient temperature. Thus, we have

$$i(G,T) = I_{sc}(G,T) - I_o(T) \cdot e^{\frac{v(G,T)+i(G,T)\cdot R_s}{n_s \cdot V_{th}}} \tag{12}$$

where $I_{sc}(G,T) = \frac{G}{G_0} \cdot I_{sc} \cdot (1 + \frac{k_i}{100} \cdot (T - T_0))$ and $I_o(T) = I_{sc} \cdot (1 + \frac{k_i}{100} \cdot (T - T_0)) \cdot e^{-\frac{V_{oc} + k_v \cdot (T - T_0)}{n_s \cdot V_{th}}}$. G_0 and T_0 are the respective irradiance level and temperature in Standard Test Conditions (STC), *i.e.*, $G_0 = 1000W/m^2$ and $T_0 = 25^oC$. I_{sc}, V_{oc}, k_v and k_i are the given parameters of short-circuit current, open-circuit voltage, temperature coefficients of the short-circuit and the open-circuit in STC from the datasheet of PV panels, respectively.

In particular, the solar power produced by a PV panel with respect to the varying weather conditions, based on the current-voltage characteristic shown as equation (11) is the product of the output voltage and current. Namely, $p_{solar} = v(G,T) \cdot i(G,T)$. It has been demonstrated that the power output p_{solar} generated by a PV panel shows a unique maximum value under uniform irradiation and temperature [31,41]. In order to achieve the maximum efficiency of solar plants, some researchers have already put efforts in extracting the maximum power point from solar plants [19,27]. We thus estimate the solar power output by a PV panel as the maximal power value which can be extracted from the PV panel (referred to as *mpp*). Specifically, *mpp* is achieved with respect to an optimal load r_{mp} and the corresponding current i_{mp} [19], where $r_{mp} = R_s + \frac{n_s \cdot V_{th}}{I_{sc}(G,T) + I_o(T) - i_{mp}}$. Thus, $mpp = i_{mp}^2 \cdot r_{mp}$. The Lambert W-function method is then used to calculate the maximum power point *mpp* of the PV panel with respect to the varying weather conditions. We assume that there are m_s PV panels installed in a large-scale solar plant. Thus, the overall solar power output by the solar plant is estimated as

$$PS = \sum_{k=1}^{m_s} mpp^k$$

where mpp^k is the maximum power point from the k^{th} PV panel with respect to the irradiance G and temperature T.

3.5 Problem Solution

Based on the analysis above, the optimization **Problem 1** is a non-linear programming problem with both a non-linear objective function and non-linear constraints, with respect to decision variables of λ_i, x_i and y_i. However, for a service operator, it is important to design an efficient solution in order to dynamically make decisions to green the data centers with acceptable runtime overheads. We thus transfer the non-linear optimization **Problem 1** into a well-studied linear-fractional programming formulation as in the form of **Problem 2**, which can be further transferred into a standard linear programming problem. Specifically, note that for the equations (9) with respect to pW_i, pS_i and pB_i as discussed in Section 3.1, we can alternatively assume that among the λ_i requests serviced by the i^{th} data center, λ_i^W, λ_i^S and λ_i^B requests are serviced with wind energy, solar energy and brown energy, respectively. Thus, we can limit the decision variables for the optimization **Problem 1** in (1 - 8) to only workload-related variables of λ_i^W, λ_i^S and λ_i^B, instead of both workload-related variables (*i.e.*, λ_i) and percentage variables (*i.e.*, x_i and y_i).

Since $\lambda_i = \lambda_i^W + \lambda_i^S + \lambda_i^B$, **Problem 1** in (1 - 8) can be further transferred as follows.

Problem 2:

$$Maximize: \frac{\sum_{i=1}^{N} f(\lambda_i^W + \lambda_i^S)}{\sum_{i=1}^{N} f(\lambda_i^W + \lambda_i^S + \lambda_i^B)} \tag{13}$$

subject to

$$\sum_{i=1}^{N} (\lambda_i^W + \lambda_i^S + \lambda_i^B) = \lambda \tag{14}$$

$$\lambda_i^W \geq 0 \tag{15}$$

$$\lambda_i^S \geq 0 \tag{16}$$

$$\lambda_i^B \geq 0 \tag{17}$$

$$R_i \leq Rs_i \tag{18}$$

$$0 \leq f(\lambda_i^W) \leq PW_i \tag{19}$$

$$0 \leq f(\lambda_i^S) \leq PS_i \tag{20}$$

$$0 \leq f(\lambda_i^W + \lambda_i^S + \lambda_i^B) \leq Ps_i \tag{21}$$

$$\sum_{i=1}^{N} (PrW_i \cdot f(\lambda_i^W) + PrS_i \cdot f(\lambda_i^S) + PrB_i \cdot f(\lambda_i^B)) \leq Cs \tag{22}$$

Specifically, $f(\lambda_i^W)$, $f(\lambda_i^S)$ and $f(\lambda_i^B)$ represent the amount of wind energy, solar energy and brown energy consumed in the i^{th} data center, respectively. It is clear that $f(\lambda_i^W)$, $f(\lambda_i^S)$ and $f(\lambda_i^B)$ are linear functions as discussed in Section 3.2.

Problem 2 is thus a specific case of linear-fractional programming problem with a fractional objective function and linear constraints. In order to solve the LFP-based optimization **Problem 2**, we leverage a standard technique discussed in [24] to transfer the problem in (13 - 22) to a linear programming problem. The detailed transformation is not shown due to space limitations, but the steps can be found in [24]. In our system, we implement the proposed GreenWare middleware system based on the *linprog* solver in Matlab. In particular, *linprog* uses an simplex method, which has been proven to have a low complexity in practice [7].

4 Simulation Setup

We aim to use realistic parameters in our experimental setup. We design a simulator and use real-world weather data, Web request traces, as well as electricity price data from utility companies to evaluate the proposed GreenWare system. As discussed, GreenWare dynamically conducts request dispatching to maximize the percentage of renewable energy used to power a network of distributed data centers within the cost budget determined by the Internet service operator. These evaluations primarily target web server-based applications, which provide the request-response type of web services. Specifically, the setup simulates an Internet-scale data center network such as Google's data centers within the US.

4.1 Datacenter Parameters

In our evaluation, we simulate a large system composed of four geographically distributed data centers for an Internet service operator (*e.g.*, Google). Accordingly, four different locations are assumed in the simulator, *i.e.*, *San Luis Valley* in *Colorado*, *Los Angeles* in *California*, *Oak Ridge* in *Tennessee* and *Lanai* in *Hawaii*, which are the locations whose meteorological data are available in [6].

The power consumption profile of each server in the same location is assumed to be approximately the same, which is usually true when homogeneous servers

and configurations are used in each data center [40,33]. Specifically, similar to a related study [32], the server configuration in each location is respectively assumed to be as follows: Data Center 1 (2.0 GHz AMD Athlon processor), Data Center 2 (1.2 GHz Intel Pentium 4630 processor), Data Center 3 (2.9 GHz Intel Pentium D950 processor), and Data Center 4 (2.7 GHz AMD Athlon processor). Their power consumption is assumed to be 88.88, 34.10, 149.19, and 141.28 Watts and their processing capacity coefficients are estimated as 500, 300, 725, and 675 requests per second, respectively.

4.2 Renewable Energy Availability

To emulate the intermittent availabilities of renewable energy in the locations of different data centers, *i.e.*, wind power and solar power, we use meteorological data from the Measurement and Instrumentation Data Center (MIDC) [6] of the National Renewable Energy Laboratory. A variety of meteorological data, including irradiances, temperature, and wind speed, is covered in those records from the MIDC. Moreover, prior studies have shown that the data from the MIDC is sufficiently accurate [31]. In particular, we use meteorological data from the four stations, *e.g.*, *Sun Spot One*, *Loyola Marymount University Rotating Shadowband Radiometer*, *Oak Ridge National Laboratory* and *La Ola Lanai*, since they have consistent time periods with available meteorological data, beginning from June 1st, 2010 to June 30th, 2010. We further assume that there are 200 turbines installed in each wind farm and 10,000 solar panels installed in each solar plant to provide renewable energy to the local power utilities of the 4 data centers. In particular, BP-MSX 120 panels produced by British Petroleum are assumed to be used in the solar plants [1].

Specifically, based on the power models discussed in Sections 3.3 and 3.4, as well as the varying weather conditions obtained from MIDC, the available renewable energy of all the 4 data centers throughout the entire simulated month is demonstrated in Figures 2 and 3. In particular, Figure 2 depicts the overall available wind energy of all the 4 data centers, while Figure 3 shows the overall available solar energy. As shown in these two figures, the available renewable energy shows a diurnal pattern. This is due to the fact that the local weather conditions have a nearly diurnal pattern.

4.3 Real-World Workload Traces

To build our workloads in the simulator, we use a trace of Internet traffic from Wikipedia.org [45]. In particular, we use this tracefile with 2-month long data, which contains 10% of user requests that arrived at Wikipedia between October 1st, 2007 and November 30th, 2007. Figure 4 shows the hourly behavior of user requests in October and November, 2007. As illustrated in the figure, the users' behavior shows a very clear weekly pattern in visiting the Wikipedia website. Specifically, we take the 1-month long Wikipedia trace of November as the incoming workload in the simulator while using the October trace data to work as the historical observations of the workload to predict hourly cost budgets.

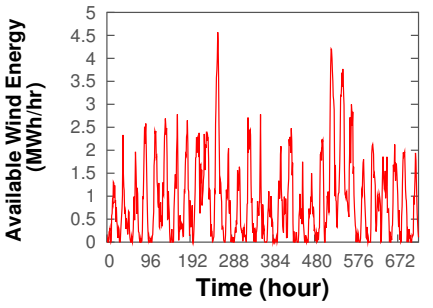


Fig. 2. The trace of available wind energy throughout the entire simulated month

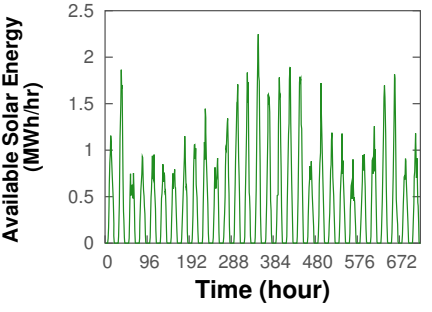


Fig. 3. The trace of available solar energy throughout the entire simulated month

To do so, we maintain a history of the request arrival rate seen during each hour of the week over the past several weeks. We then calculate every averaged hourly workload weight of the whole week over the past several weeks as the hourly budget weight in the coming week. Based on experiments, we find that for this Wikipedia trace, a 2-week long history trace data can provide a reasonable prediction on hourly cost budgets. Note that more sophisticated prediction methods, such as [46], can also be integrated into our system.

To make our evaluation more general, we also stress test GreenWare with another workload trace from the 1998 World Cup game, which includes the request data of 33 servers from 4 geographical locations. In particular, it records the incoming requests to all the servers with a granularity of 1 second from April 30th to July 26th, 1998.

4.4 Electricity Price Traces

To simulate the electricity price for the brown energy, we use the price trace from New York Independent System Operator (NYISO) [10], since they have complete and accurate price data records. Specifically, we use the Day-Ahead price data from November 1st, 2007 to November 30th, 2007, which is consistent with the dates of the Wikipedia traces. We apply the price data from the four

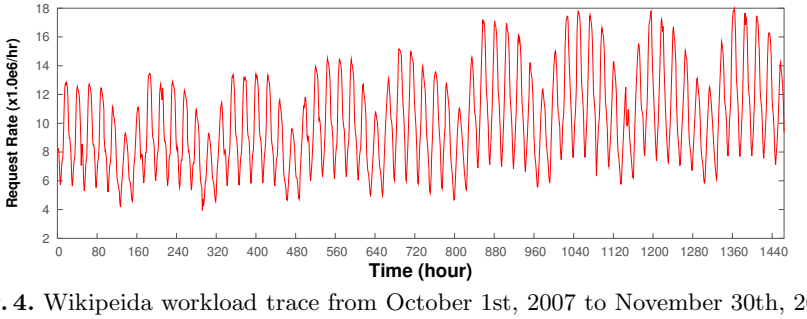


Fig. 4. Wikipeida workload trace from October 1st, 2007 to November 30th, 2007

zones, including Capital, Central, Dunwoodie and Genesee to the 4 data centers in our simulation.

On the other hand, regarding the electricity price of renewable energy, it is usually true that renewable energy has a higher electricity price compared to brown energy [2,5], due to the intermittent nature of renewable energy sources such as wind and sunlight, as well as expensive facility investments and management. For example, renewable energy costs an additional 1.5 cents per KWh compared to the regular energy in the power market of Virginia [2]. Furthermore, solar energy is typically much more expensive than wind energy, due to the relatively high capital expenses [3,11]. Thus, to be more practical, in our simulation we assume that the wind electricity price is 1.5 cents higher per KWh than brown energy [2]; while solar energy is 18.0 cents higher per KWh [3].

5 Evaluation Results

In this section, we first introduce two baselines. We then compare the proposed GreenWare middleware system against the baselines.

5.1 Baselines

In our work, we use two baselines in our experiments, a cost minimization only policy and a green energy usage maximization only policy, referred to as *Min-Cost* and *Max-Green*, respectively. (1) **Min-Cost.** Similar to GreenWare, Min-Cost also tries to minimize the electricity cost by distributing requests among geographically distributed data centers to leverage the varying electricity prices in different locations. However, different from GreenWare, Min-Cost is unaware of renewable energy and thus prefers brown energy in cost minimization. Min-Cost is similar to the state-of-the-art work [40] in minimizing the electricity bill in operating data center networks. (2) **Max-Green.** Similar to GreenWare, Max-Green tries to maximize the use of renewable energy by distributing more requests to data centers where more renewable energy is available. However, Max-Green does so regardless of the cost budget and thus may lead to a high operation cost for the Internet service operators and sometimes even budget violations. This scheme is similar to the state-of-the-art work [42] in powering data centers with renewable energy.

5.2 Impacts of the Monthly Cost Budget

In this experiment, we evaluate the proposed GreenWare middleware with respect to different monthly cost budgets.

Figures 5 and 6 depict how GreenWare works with the Wikipedia workload under a monthly cost budget of $340K. In particular, these two figures show that with a sufficient monthly cost budget (*e.g.*, as shown in Figure 5, the allocated hourly budget is sufficient throughout the entire month), brown energy is used only in the invocation periods with insufficiently available renewable energy.

That is, as indicated in Figure 6, only when the available renewable energy supply is less than the actual renewable energy demand (*i.e.*, a difference lower than 0), the corresponding renewable energy usage does not reach 100%, *e.g.*, the hours of 2, 5, 6, 7 and etc. Note that there are some invocation periods which have a zero usage of renewable energy, *e.g.*, the hours of 1, 3, 4 and etc. This is because that there is no available renewable energy at all due to the weather conditions in those invocation periods. In addition, Figure 5 demonstrates that the hourly allocated cost budget within one week shows a growing trend. This is due to the fact that we carry over the unused allocated cost budget from previous invocation periods to the remaining invocation periods in the same week.

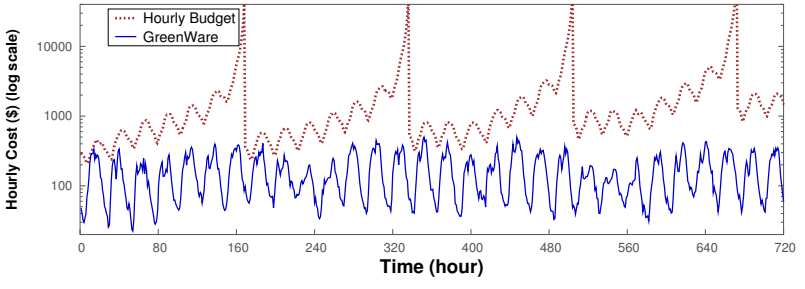


Fig. 5. Hourly electricity cost by GreenWare with a sufficient monthly cost budget of $340K, with respect to Nov. 2007 Wikipedia trace

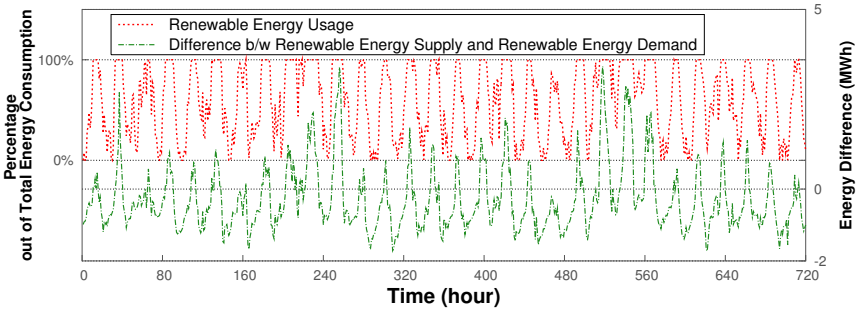


Fig. 6. Hourly renewable energy usage by GreenWare with a sufficient monthly cost budget of $340K, with respect to Nov. 2007 Wikipedia trace

We then study GreenWare under a series of different monthly cost budgets. As shown in Figure 7, with the increase of the monthly cost budget, the monthly average percentage of renewable energy usage keeps rising and then stays stable. This is due to the fact that fewer invocation periods are allocated with an insufficient cost budget in the case with a higher monthly cost budget. Therefore, more renewable energy can be used to power the data center networks. For example, with a monthly cost budget of $100K, there are 202 invocation periods

which have sufficient renewable energy supply but with an insufficient allocated cost budget; while as low as only 42 invocation periods are allocated with an insufficient cost budget in the case with a $160K monthly cost budget. As a result, a higher monthly average percentage of 58.17% of renewable energy usage is achieved with the monthly cost budget of $160K, compared to a percentage of 45.95% with the monthly budget $100K. Thus, when all the invocation periods have a sufficient budget due to a sufficient monthly cost budget, *e.g.*, $320K and $340K, the monthly average renewable energy usage stays stable. This set of experiments demonstrates that GreenWare can significantly increase the use of renewable energy in powering the data center network, subject to the desired cost budget.

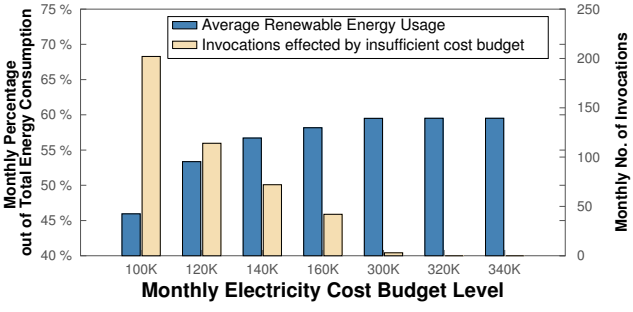


Fig. 7. Average percentage of renewable energy usage by GreenWare with a series of different monthly cost budgets

5.3 Comparison with Baselines

In this experiment, we compare GreenWare with the two baselines: Min-Cost and Max-Green.

Figure 8 depicts the cost and brown energy consumption of GreenWare, Max-Green and Min-Cost, with respect to a given monthly budget, *e.g.*, $100K, for the Wikipeida workload. The results are normalized against Min-Cost, which actually indicates the case of only using brown energy in powering data center networks. Figure 8 shows that although Max-Green (*i.e.*, maximizing the use of green energy regardless of cost budget) can decrease brown energy consumption by 58% compared to Min-Cost by utilizing as much renewable energy as possible. However, due to its unawareness of cost budget, Max-Green results in a 109% cost increase and exceeds the monthly cost budget by 29%. On the other hand, GreenWare can achieve an as-much-as-42% decrease in brown energy consumption at only a 52% cost increase, compared to Min-Cost. More importantly, GreenWare successfully controls the electricity bill to stay within the cost budget for the Internet service operator.

To demonstrate the effectiveness of GreenWare with different workloads, we also stress test GreenWare using the 1998 World Cup trace. Specifically, we use the request trace in June as the incoming workload in the simulation, and the

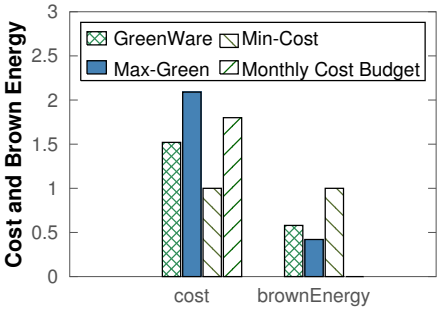


Fig. 8. Comparison between GreenWare and baselines with respect to Nov. 2007 Wikipeida trace

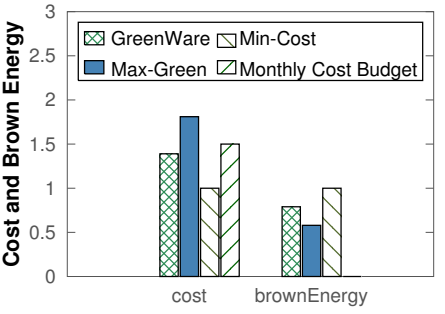


Fig. 9. Comparison between GreenWare and baselines with respect to Jun. 1998 World Cup trace

May trace as historical data to predict the hourly cost budget. To simulate the workload of cloud-service data centers, we proportionally increase the request numbers. Figure 9 shows the experiment results on the comparison between GreenWare and the two baselines. As demonstrated in the figure, Max-Green achieves a 42% decrease in brown energy consumption compared to Min-Cost. However, the electricity bill exceeds the given monthly cost budget (*e.g.*, \$100K) by 31%. On the other hand, GreenWare obtains an as-much-as-21% decrease in brown energy consumption while successfully controlling the electricity bill to stay within the monthly cost budget.

5.4 Impacts of Pricing Policies of Renewable Energy

In this experiment, we show that the proposed GreenWare middleware always prefers the type of renewable energy that has a lower electricity price. Thus, an efficient cost minimization is guaranteed. Since in our work we just consider two types of the most popular renewable energy, *i.e.*, wind energy and solar energy, we assume two different pricing policies: (1) wind energy has a lower electricity price, as discussed in Section 4.4; and (2) solar energy has a lower price than wind energy. Note that the current practice is that wind energy is typically less expensive than solar energy. However, in order to stress test GreenWare, we assume a lower price for solar energy in (2).

Figures 10 and 11 demonstrate how the usage of different types of renewable energy varies with different pricing policies as discussed above. Intuitively, the more expensive renewable energy is taken into use only when the less expensive type of renewable energy is used up. As shown in Figure 10, with the first pricing policy (*i.e.*, wind energy price is lower), solar energy is used to power data centers only after all the supplied wind energy has been used up, as indicated in the second data center (DC#2). Similarly, with the second pricing policy, wind energy is used to power data centers only after all the available less expensive solar energy is consumed, as in all the data centers in Figure 11. Note that

in Figure 10, Data Centers 1, 3 and 4 begin to use the more expensive solar energy though there is still some wind energy left. This is because that there are some invocation periods when the available wind energy is too much to serve the incoming workload. As a result, some wind energy is left unused and the unused wind energy cannot be used in the following invocation periods due to the intermittent feature of the renewable energy.

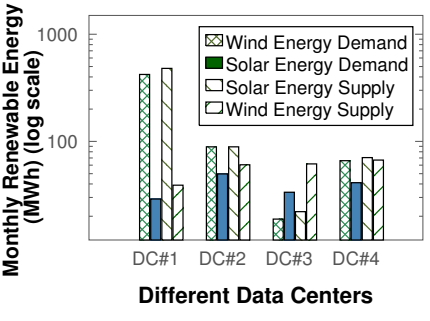


Fig. 10. Monthly renewable energy usage by GreenWare when wind energy price is lower than solar energy price

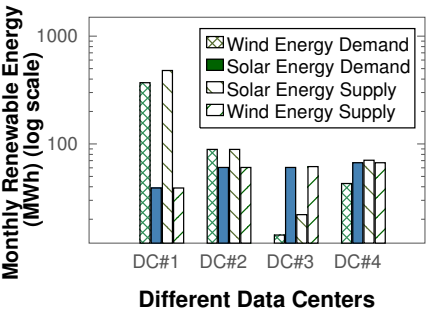


Fig. 11. Monthly renewable energy usage by GreenWare when solar energy price is lower than wind energy price

6 Related Work

Greening data centers is becoming an increasingly important topic in operating cloud-scale data center networks for Internet service operators, due to (1) data centers having become major energy consumers [44] and (2) the global energy crisis and environmental concerns (*e.g.*, global warming) [31]. To our best knowledge, our study is the first one that proposes to maximally use the renewable energy supplied by the local power utilities for Internet service operators, while being aware of the time-varying electricity price and enforcing a desired cost budget. Compared with the state of the art, the considerations of various realistic constraints make GreenWare more practical. We now discuss the related work.

Energy conservation in data centers. Many recent research projects have tried to minimize the energy consumption of data centers. For example, Chen et al. [18] and Chase et al. [17] reduce the energy consumption of connection servers hosting long-lived TCP-connection services and web servers providing request-response type of services, respectively. Heo et al. [23] have developed an adaptation graph analysis mechanism to solve the conflicts between interacting adaptive components, *e.g.*, *On/Off* and *dynamic voltage scaling* policies in server farms, to minimize energy consumption. Elnozahy et al. [20] investigate various combinations of dynamic voltage scaling and node on/off policies to reduce the energy consumption in server farms. Other strategies on reducing energy consumption of servers are also proposed (*e.g.*, [26,47]).

Our work differs from these efforts in several ways: (1) none of them try to use renewable energy to power data center networks; and (2) none of them put efforts on managing the electricity cost for the Internet service operators.

Managing electricity cost in data centers. A few recent projects have proposed to minimize the electricity bills of data center networks. For example, Qureshi et al. [39] try to lower the electricity bill by utilizing the varying electricity prices in different locations of distributed data centers. Rao et al. [40] consider a multi-electricity-market environment to reduce the electricity bill. In a recent study, Zhang et al. [48] propose an electricity bill capping algorithm to minimize the electricity cost within the cost budget for data center networks. Lin et al. [33] have tried to minimize the energy cost together with delay cost by rightly sizing data centers. Our work differs significantly from these efforts in that none of them try to maximize the use of renewable energy in powering data center networks for the Internet service operators.

Utilizing renewable energy in data centers. This is a relatively new topic with only few initial studies. Le et al. [29,28] propose to cap the consumption of brown energy while maintaining service level agreements (SLAs). Liu et al. [34] investigate how renewable energy can be used to lower the electricity price of brown energy in a specific power market, *i.e.*, where the brown energy is dynamically priced in proportion to the total brown energy consumption. Brown et al. [15] propose a simulation infrastructure to model a data center using renewable energy sources. In contrast to those studies, GreenWare aims to solve a related but different problem, *i.e.*, maximizing the use of renewable energy subject to the cost budget of the Internet service operators. Steward et al. [42] also try to maximize the use of renewable energy in data centers. However, their study assumes that Internet service operators have their own wind farms or solar plants. In contrast, GreenWare considers a different case where the service operators buy renewable energy from the power grid, which is a more common case for many data centers because of concerns such as expensive facility investments and management. In addition, their study does not consider the extra cost of renewable energy and may lead to budget violations as shown in the comparisons between GreenWare and Max-Green in Section 5.3. Li et al. [30] propose a load power tuning scheme for managing intermittent renewable power in a single data center without considering the costs. In contrast, we focus on distributing requests among data centers in different locations.

7 Conclusion

Two key questions faced by many cloud-service operators are 1) how to dynamically distribute service requests among data centers in different geographical locations, based on the local weather conditions, to maximize the use of renewable energy, and 2) how to do so within their allowed operation budgets. In this paper, we have presented GreenWare, a novel middleware system that conducts dynamic request dispatching to maximize the percentage of renewable energy

used to power a network of distributed data centers, subject to the desired cost budget of the Internet service operators. Our solution first explicitly models the intermittent generation of renewable energy, e.g., wind power and solar power, with respect to varying weather conditions in the geographical location of each data center. We then formulate the core objective of GreenWare as a constrained optimization problem and propose an efficient request dispatching algorithm based on linear-fractional programming (LFP). We evaluate GreenWare with real-world weather, electricity price, and workload traces. Our experimental results show that GreenWare can significantly increase the use of renewable energy in cloud-scale data centers without violating the desired cost budget, despite the intermittent supplies of renewable energy in different locations and time-varying electricity prices and workloads.

Acknowledgments. This work was supported, in part, by NSF under CNS-0720663 and CAREER Award CNS-0845390.

References

1. BP MSX 120 solar module, http://pdf.directindustry.com/pdf/bp-solar/bp-msx-120-solar-module/15873-68158.html
2. Dominion Virginia Power, http://www.dom.com/
3. Energy modality comparison based on projected cents per kilowatt-hour, http://peswiki.com/
4. Green House Data: Greening the data center, http://www.greenhousedata.com/
5. Los Angeles Department of Water & Power, http://www.ladwp.com/
6. Measurement and Instrumentation data center, http://www.nrel.gov/midc/
7. The Running Time of the Simplex Method, http://www.mpi-inf.mpg.de
8. Solar Electricity Prices, http://solarbuzz.com/
9. WindPower Program, http://www.wind-power-program.com/index.htm
10. NYISO (1999), http://www.nyiso.com/
11. Solar Power at Data Center Scale (2009), http://www.datacenterknowledge.com/
12. Google Buys 20 Years' Worth of Wind Energy To Power Data centers (2010), http://www.huffingtonpost.com/
13. Ahmad, F., Vijaykumar, T.N.: Joint optimization of idle and cooling power in data centers while maintaining response time. In: ASPLOS (2010)
14. Bolch, G., Greiner, S., de Meer, H., Trivedi, K.S.: Queueing Networks and Markov Chains. Wiley Interscience (2005)
15. Brown, M., Renau, J.: Rerack: Power simulation for data centers with renewable energy generation. In: GreenMetrics (2011)
16. Castaner, L., Silvestre, S.: Modelling Photovoltaic Systems Using PSpice. John Wiley & Sons (2002)
17. Chase, J.S., Anderson, D.C., Thakar, P.N., Vahdat, A.M., Doyle, R.P.: Managing energy and server resources in hosting centers. In: SOSP (2001)
18. Chen, G., He, W., Liu, J., Nath, S., Rigas, L., Xiao, L., Zhao, F.: Energy-aware server provisioning and load dispatching for connection-intensive internet services. In: NSDI (2008)

19. Ding, J., Radhakrishnan, R.: A new method to determine the optimum load of a real solar cell using the lambert w-function. Solar Energy Materials and Solar Cell (2008)
20. Elnozahy, E.N.M., Kistler, J.J., Rajamony, R.: Energy-Efficient Server Clusters. In: Falsafi, B., VijayKumar, T.N. (eds.) PACS 2002. LNCS, vol. 2325, pp. 179–196. Springer, Heidelberg (2003)
21. Kariniotakis, G.N., Stavrakakis, G.S., Nogaret, E.F.: Wind power forecasting using advanced neural networks models. IEEE Transactions on Energy Conversion, 762–767 (1996)
22. Greenberg, A., Hamilton, J., Maltz, D.A., Patel, P.: The cost of a cloud: research problems in data center networks. ACM SIGCOMM Computer Communication Review (2008)
23. Heo, J., Henriksson, D., Liu, X., Abdelzaher, T.: Integrating adaptive components: An emerging challenge in performance-adaptive systems and a server farm case-study. In: RTSS (2007)
24. Hillier, F.S., Lieberman, G.J.: Introduction to Operations Research. McGraw-Hill (2005)
25. Hohl, A.: Wind Power for Data Centers (2009), http://www.renewableenergyworld.com/rea/blog/post/2009/08/wind-power-for-data-centers
26. Horvath, T., Abdelzaher, T., Skadron, K., Liu, X.: Dynamic voltage scaling in multitier web servers with end-to-end delay control. IEEE Transactions on Computers, 444–458 (2007)
27. Kothari, L.S., Mathur, P.C., Kapoor, A., Saxena, P., Sharma, R.P.: Determination of optimum load for a solar cell. Journal of Applied Physics, 5982–5984 (2009)
28. Le, K., Bianchini, R., Martonosi, M., Nguyen, T.D.: Cost- and energy-aware load distribution across data centers. In: HOTPOWER (2009)
29. Le, K., Bilgir, O., Bianchini, R., Martonosi, M., Nguyen, T.D.: Managing the cost, energy consumption, and carbon footprint of internet services. In: SIGMETRICS (2010)
30. Li, C., Qouneh, A., Li, T.: Characterizing and analyzing renewable energy driven data centers. In: SIGMETRICS (2011)
31. Li, C., Zhang, W., Cho, C.B., Li, T.: Solarcore: Solar energy driven multi-core architecture power management. In: HPCA (2011)
32. Li, J., Li, Z., Ren, K., Liu, X., Su, H.: Towards optimal electric demand management for internet data centers. In: Techreport (2010)
33. Lin, M., Wierman, A., Andrew, L.L.H., Thereska, E.: Dynamic right-sizing for power-proportional data centers. In: INFOCOM (2011)
34. Liu, Z., Lin, M., Wierman, A., Low, S.H., Andrew, L.L.H.: Greening geograhpical load balancing. In: SIGMETRICS (2011)
35. Patel, M.R.: Power systems: Design, Analysis, and Operation. CRC Press (2006)
36. Paukshto, M.V., Lovetskiy, K.: Invariance of single diode equation and its application. In: PVSC (2008)
37. Petru, T., Thiringer, T.: Modeling of wind turbines for power system studies. IEEE Transactions on Power Systems, 1132–1139 (2002)
38. Pistoia, G.: Battery Operated Devices and Systems: From Portable Electronics to Industrial Products. Elsevier (2011)
39. Qureshi, A., Weber, R., Balakrishnan, H., Guttag, J., Maggs, B.: Cutting the electric bill for internet-scale systems. In: SIGCOMM (2009)

40. Rao, L., Liu, X., Xie, L., Liu, W.: Minimizing electricity cost: optimization of distributed internet data centers in a multi-electricity-market environment. In: INFOCOM (2010)
41. Sera, D., Teodorescu, R., Rodriguez, P.: PV panel model based on datasheet values. In: ISIE (2007)
42. Stewart, C., Shen, K.: Some joules are more precious than others: Managing renewable energy in the datacenter. In: HOTPOWER (2009)
43. Thibodeau, P.: Wind power data center project planned in urban area (2008), http://www.computerworld.com/
44. United states environmental protection agency. Report to congress on server and data center energy efficiency (2007)
45. Urdaneta, G., Pierre, G., van Steen, M.: Wikipedia workload analysis for decentralized hosting. Elsevier Computer Networks 53(11), 1830–1845 (2009), http://www.globule.org/publi/WWADH_comnet2009.html
46. Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P.: Dynamic provisioning of multitier internet applications. In: ICAC (2005)
47. Verma, A., Dasgupta, G., Nayak, T.K., De, P., Kothari, R.: Server workload analysis for power minimization using consolidation. In: USENIX ATC (2009)
48. Zhang, Y., Wang, Y., Wang, X.: Capping the electricity cost of cloud-scale data centers with impacts on power markets. In: HPDC (2011)

Resource Provisioning Framework for MapReduce Jobs with Performance Goals *

Abhishek Verma[1], Ludmila Cherkasova[2], and Roy H. Campbell[1]

[1] University of Illinois at Urbana-Champaign
{verma7,rhc}@illinois.edu
[2] HP Labs, Palo Alto
{lucy.cherkasova}@hp.com

Abstract. Many companies are increasingly using MapReduce for efficient large scale data processing such as personalized advertising, spam detection, and different data mining tasks. Cloud computing offers an attractive option for businesses to rent a suitable size Hadoop cluster, consume resources as a service, and pay only for resources that were utilized. One of the open questions in such environments is the amount of resources that a user should lease from the service provider. Often, a user targets specific performance goals and the application needs to complete data processing by a certain time deadline. However, currently, the task of estimating required resources to meet application performance goals is solely the users' responsibility. In this work, we introduce a novel framework and technique to address this problem and to offer a new resource sizing and provisioning service in MapReduce environments. For a MapReduce job that needs to be completed within a certain time, the job profile is built from the job past executions or by executing the application on a smaller data set using an automated profiling tool. Then, by applying scaling rules combined with a fast and efficient capacity planning model, we generate a set of resource provisioning options. Moreover, we design a model for estimating the impact of node failures on a job completion time to evaluate worst case scenarios. We validate the accuracy of our models using a set of realistic applications. The predicted completion times of generated resource provisioning options are within 10% of the measured times in our 66-node Hadoop cluster.

1 Introduction

Private and public clouds offer a new delivery model with virtually unlimited computing and storage resources. Many companies are following the new trend of using MapReduce [1] and its open-source implementation Hadoop for large-scale, data intensive processing and for mining petabytes of unstructured information. However, setting up a dedicated Hadoop cluster requires a significant capital expenditure that can be difficult to justify. Cloud computing offers a compelling alternative and allows users to rent resources in a "pay-as-you-go" fashion. A list of supported services by Amazon (Amazon Web Services) includes MapReduce environments for rent. It is an attractive and cost-efficient option for many users because acquiring and maintaining complex, large-scale infrastructures is a difficult and expensive decision. Hence, a typical practice among

* This work was largely completed during A. Verma's internship at HP Labs. R. Campbell and A. Verma are supported in part by NSF CCF grants #0964471, IIS #0841765 and Air Force Research grant FA8750-11-2-0084.

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 165–186, 2011.

MapReduce users is to develop their applications in-house using a small development testbed and test it over a small input dataset. They can lease a MapReduce cluster from the service provider and subsequently execute their MapReduce applications on large input datasets of interest. Often, the application is a part of a more elaborate business pipeline, and the MapReduce job has to produce results by a certain time deadline, i.e., it has to achieve certain performance goals and service level objectives (SLOs). Thus, a typical performance question in MapReduce environments is "*how to estimate the required resources (number of map and reduce slots) for a job so that it achieves certain performance goals and completes data processing by a given time?*" Currently, there is no available methodology to easily answer this question, and businesses are left on their own to struggle with the resource sizing problem: they need to perform adequate application testing, performance evaluation, capacity planning estimation, and then request appropriate amount of resources from the service provider.

In this work, we propose a novel framework to solve this problem and offer a new resource sizing and provisioning service in MapReduce environments. First, we introduce an automated profiling tool that extracts a compact job profile from the past application execution(s) in the production Hadoop cluster. Alternatively, profiling can done by executing a given application with a smaller input dataset than the original one. The power of the designed technique is that it offers a compact job profile that is comprised of performance *invariants* which are independent of the amount of resources assigned to the job (i.e., the size of the Hadoop cluster) and the size of the input dataset. The job profile accurately reflects the application performance characteristics during all phases of a given job: map, shuffle/sort, and reduce phases. Our automated profiling technique does not require any modifications or instrumentation of neither the application nor of the underlying Hadoop execution engine. All this information can be obtained from the counters at the job master during the job's execution or alternatively parsed from the job execution logs written at the job tracker.

For many applications, increasing input dataset while keeping the same number of reduce tasks leads to an increased amount of data shuffled and processed per reduce task. Using linear regression, we derive *scaling factors* for shuffle and reduce phases to estimate their service times as a function of input data.

We design a *MapReduce performance model* that predicts the job completion time based on the job profile, input dataset size, and allocated resources. We enhance the designed model to evaluate the performance impact of failures on job completion time. This model helps in evaluating worst case scenarios and deciding on the necessity of additional resources or program changes as a means of coping with potential failure scenarios. Finally, we propose a fast and efficient, fully automated capacity planning procedure for estimating the required resources to meet a given application SLO. The output of the model is a set of plausible solutions (if such solutions exist for a given SLO) with a choice of different numbers of map and reduce slots that need to be allocated for achieving performance goals of this application.

We validate the accuracy of our approach and performance models using a set of realistic applications. First, we build the application profiles and derive scaling factors using small input datasets for processing. Then we perform capacity planning and generate plausible resource provisioning options for achieving a given application SLO for processing a given (large) dataset. The predicted completion times of these generated options are within 10% of the measured times in the 66-node Hadoop cluster.

This paper is organized as follows. Section 2 provides a background on MapReduce. Section 3 introduces our approach towards profiling MapReduce jobs. Section 4 presents a variety of MapReduce performance models and the SLO-based resource provisioning. The efficiency of our approach and the accuracy of designed models is evaluated in Section 5. Section 6 describes the related work. Section 7 summarizes the paper and outlines future directions.

2 MapReduce Background

This section provides an overview of the MapReduce [1] abstraction, execution, scheduling, and failure modes. In the MapReduce model, computation is expressed as two functions: map and reduce. The map function takes an input pair and produces a list of intermediate key/value pairs. The intermediate values associated with the same key k_2 are grouped together and then passed to the reduce function. The reduce function takes intermediate key k_2 with a list of values and processes them to form a new list of values.

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$
$$reduce(k_2, list(v_2)) \rightarrow list(v_3)$$

MapReduce jobs are distributed and executed across multiple machines: the map stage is partitioned into *map tasks* and the reduce stage is partitioned into *reduce tasks*.

Each map task processes a logical split of input data that generally resides on a distributed file system. Files are typically divided into uniform sized blocks (default size is 64 MB, which is a file system parameter) and distributed across the cluster nodes. The map task reads the data, applies the user-defined map function on each record, and buffers the resulting output. This data is sorted and partitioned for different reduce tasks, and written to the local disk of the machine executing the map task. The reduce stage consists of two phases: *shuffle* and *reduce* phase. In the *shuffle phase*, the reduce tasks fetch the intermediate data files from the already completed map tasks, thus following the "pull" model. The intermediate files from all the map tasks are sorted. An external merge sort is used in case the intermediate data does not fit in memory as follows: the intermediate data is shuffled, merged in memory, and written to disk. After all the intermediate data is shuffled, a final pass is made to merge all these sorted files. Thus, the shuffling and sorting of intermediate is interleaved: we denote this by *shuffle/sort* or simply *shuffle* phase. Finally, in the *reduce phase*, the sorted intermediate data is passed to the user-defined reduce function. The output from the reduce function is generally written back to the distributed file system.

Job scheduling in Hadoop is performed by a master node, which manages a number of worker nodes in the cluster. Each worker has a fixed number of *map slots* and *reduce slots*, which can run tasks. The number of map and reduce slots is statically configured. The slaves periodically send heartbeats to the master to report the number of free slots and the progress of tasks that they are currently running. Based on the availability of free slots and the scheduling policy, the master assigns map and reduce tasks to slots in the cluster.

In the real world, user code is buggy, processes crash, and machines fail. MapReduce is designed to scale to a large number of machines and to yield a graceful performance degradation in case of failures. There are three types of failures that can occur.

First, a map or reduce task can fail because of buggy code or runtime exceptions. The worker node running the failed task detects task failures and notifies the master. The master reschedules the execution of the failed task, preferably on a different machine. Secondly, a worker can fail, e.g., because of OS crash, faulty hard disk, or network interface failure. The master notices a worker that has not sent any heartbeats for a specified time interval and removes it from its worker pool for scheduling new tasks. Any tasks in progress on the failed worker are rescheduled for execution. The master also reschedules all the completed map tasks on the failed worker that belong to running jobs, since the intermediate data of these maps may not be accessible to reduce tasks of these jobs. Finally, the failure of the master is the most serious failure mode. Currently, Hadoop has no mechanism for dealing with the failure of the job master. This failure is rare and can be avoided by running multiple masters and using a Paxos consensus protocol to decide the primary master.

3 Profiling MapReduce Jobs

In this section, we discuss different executions of the same MapReduce job in the Hadoop cluster as a function of the job's map and reduce tasks and the allocated map and reduce slots for executing it. Our goal is to extract a single *job profile* that uniquely captures critical performance characteristics of the job execution in different stages. We introduce a fully automated profiling technique that extracts a compact job profile from the past application execution(s) in the production Hadoop cluster.

3.1 Job Execution as a Function of Allocated Resources

Let us consider two popular MapReduce applications (described below) and demonstrate the differences between their job executions and job completion times as a function of the amount of resources allocated to these jobs.

- The first application is the *Sort benchmark* [2], which involves the use of identity map/reduce function: The output of the map and reduce task is the same as its input. Thus, the entire input of map tasks is shuffled to reduce tasks and then written as output.
- The second application is *WikiTrends* application that processes Wikipedia article traffic logs that were collected (and compressed) every hour. *WikiTrends* counts the number of times each article has been visited in the given input dataset, i.e. access frequency or popularity count of each Wikipedia article over time.

First, we run the Sort benchmark with 8GB input on 64 machines each configured with a single map and a single reduce slot, i.e., with 64 map and 64 reduce slots overall. Figure 1 shows the progress of the map and reduce tasks over time (on the x-axis) vs the 64 map slots and 64 reduce slots (on the y-axis). Since we use blocksize of 128MB, we have 8GB/128MB = 64 input splits. As each split is processed by a different map task, the job consists of 64 map tasks. This job execution results in a single map and reduce *wave*. We split each reduce task into its constituent shuffle/sort and reduce phases. As seen in the figure, since the shuffle phase starts immediately after the first map task is completed, the shuffle phase overlaps with the map stage.

Next, we run the Sort benchmark with the same 8GB input dataset on the same testbed, except this time, we provide it with fewer resources: 16 map slots and 22 reduce

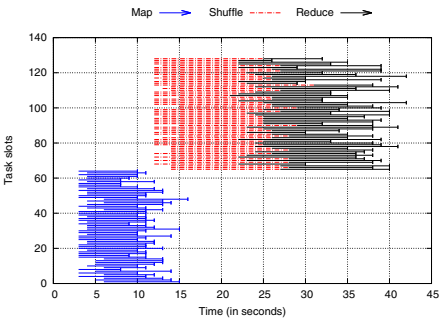


Fig. 1. Sorting with 64 map and 64 reduce slots

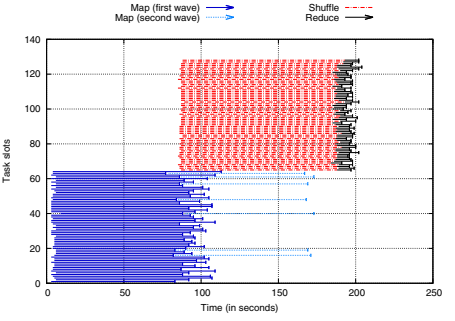


Fig. 2. WikiTrends with 64 map and 64 reduce slots

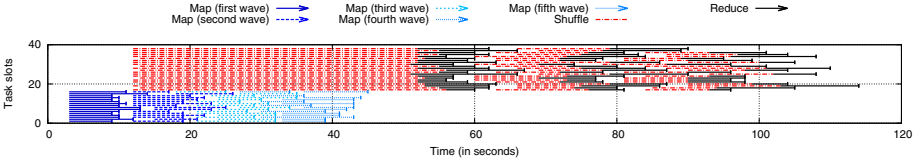


Fig. 3. Sorting with 16 map and 22 reduce slots

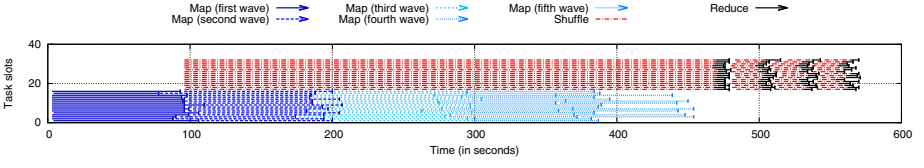


Fig. 4. WikiTrends with 16 map and 16 reduce slots

slots. As shown in Figure 3, since the number of map tasks is greater than the number of provided map slots, the map stage proceeds in multiple rounds of slot assignment, viz. 4 waves ($\lceil 64/16 \rceil$)[1]. These waves are not synchronized with each other, and the scheduler assigns the task to the slot with the earliest finishing time. Similarly, the reduce stage proceeds in 3 waves ($\lceil 64/22 \rceil$).

While the executions of four map waves resemble each other, note the difference between the first reduce wave and the following two reduce waves in this job execution. As we mentioned earlier, the shuffle phase of the first reduce wave starts immediately after the first map task completes. Moreover, this first shuffle phase continues until all the map tasks are complete, and their intermediate data is copied to the active reduce tasks. Thus the first shuffle phase overlaps with the entire map stage. After the shuffle/sort phase is done, the reduce computation can be performed and the generated data

[1] Note, that for multiple map and reduce waves, there is an extra overhead for starting map and reduce tasks. In our ongoing work, we design a set of micro-benchmarks to automatically assess these overheads in different MapReduce environments for incorporating them in the performance model.

written back to HDFS. After that, the released reduce slots become available to the next reduce tasks. As shown in Figure 3, there is a drastic difference between the execution of the first reduce wave, but the executions of the remaining reduce waves bear a strong resemblance to each other.

Figures 2 and 4 present our second example with WikiTrends application. In this example, we process a subset of logs collected during a few days in April, 2011. There are 71 files in the set that correspond to 71 map tasks and 64 reduce tasks in this application. First, we execute WikiTrends with 64 map and 64 reduce slots. The job execution consists of two map waves ($\lceil 71/64 \rceil$) and a single reduce wave as shown in Figure 2. The second map wave processes only 7 map tasks. However, the shuffle phase of the reduce stage can be completed only when all the map tasks are done, and overlaps with both preceding map waves. Figure 4 shows the WikiTrends execution with 16 map and 16 reduce slots. The job execution has 5 map and 4 reduce waves. Again, we can see a striking difference between the first reduce wave and the remaining 3 reduce waves (which resemble each other).

As observed from Figures 1- 4, it is difficult to predict the completion time of the same job when different amount of resources are given to the job. Traditionally, a simple rule of thumb states [3], that if T is a completion time of a MapReduce job with X map and Y reduce slots then by using a smaller Hadoop cluster with $X/2$ map and $Y/2$ reduce slots the same job will be processed twice as slow, i.e., in $2 \cdot T$. While it is clear, that the job execution time is a function of allocated resources, the scaling rules are more complex, and the simple example with WikiTrends shows this. The completion time of WikiTrends in 64x64 configuration is approx. 200 sec. However, the completion time of WikiTrends in 16x16 configuration (4 times smaller cluster) is approx. 570 sec, which is far less than 4 times (naively expected) completion time increase. Evidently, more elaborate modeling and job profiling techniques are needed to capture the unique characteristics of MapReduce applications and to predict their completion time.

3.2 Job Performance Invariants as a Job Profile

Our goal is to create a compact job profile comprising of performance *invariants* that are independent of the amount of resources assigned to the job over time and that reflects all phases of a given job: map, shuffle/sort, and reduce phases. Metrics and timing of different phases, that we use below, can be obtained from the counters at the job master during the job's execution or parsed from the logs.

Map Stage: The map stage consists of a number of map tasks. To compactly characterize the distribution of the map task durations and other invariant properties, we extract the following metrics:

$$(M_{min}, M_{avg}, M_{max}, AvgSize_M^{input}, Selectivity_M), \text{where}$$

- M_{min} – the minimum map task duration. Since the shuffle phase starts when the first map task completes, we use M_{min} as an estimate for the beginning of the shuffle phase.
- M_{avg} – the average duration of map tasks to summarize the duration of a map wave.

- M_{max} – the maximum duration of the map tasks[2]. Since the shuffle phase can finish only when the entire map stage completes, i.e. all the map tasks complete, M_{max} is an estimate for a worst map wave completion time.
- $AvgSize_M^{input}$ - the average amount of input data for the map task. We use it to estimate the number of map tasks to be spawned for processing a new dataset.
- $Selectivity_M$ – the ratio of the map output size to the map input size. It is used to estimate the amount of intermediate data produced by the map stage as input to the reduce stage.

Reduce Stage: As described earlier, the reduce stage consists of the shuffle/sort and reduce phases. The shuffle phase begins only after the first map task has completed. The shuffle phase (of any reduce wave) completes when the entire map stage is complete and all the intermediate data generated by the map tasks has been shuffled to the reduce tasks and has been sorted. After shuffle/sort completes, the reduce phase is performed. Thus the profiles of shuffle and reduce phases are represented by the *average* and *maximum* of their tasks durations. In addition, for the reduce phase, we compute the *reduce selectivity*, denoted as $Selectivity_R$, which is defined as the ratio of the reduce output size to its input.

The shuffle phase of the first reduce wave may be significantly different from the shuffle phase that belongs to the next reduce waves (illustrated in Figure 3, 4). This happens because the shuffle phase of the first reduce wave overlaps with the entire map stage and depends on the number of map waves and their durations. Therefore, we collect two sets of measurements: (Sh_{avg}^1, Sh_{max}^1) for shuffle phase of the first reduce wave (called, *first shuffle*) and $(Sh_{avg}^{typ}, Sh_{max}^{typ})$ for shuffle phase of the other waves (called, *typical shuffle*). Since we are looking for performance invariants that are independent of the amount of allocated resources to the job, we characterize shuffle phase of the first reduce wave in a special way and include only the non-overlapping portions of the first shuffle in (Sh_{avg}^1 and Sh_{max}^1). Thus the job profile in the shuffle phase is characterized by two pairs of measurements: $(Sh_{avg}^1, Sh_{max}^1, Sh_{avg}^{typ}, Sh_{max}^{typ})$.

The reduce phase begins only after the shuffle phase is complete. The profile of the reduce phase is represented by the *average* and *maximum* of the reduce tasks durations and the *reduce selectivity*, denoted as $Selectivity_R$, which is defined as the ratio of the reduce output size to its input: $(R_{avg}, R_{max}, Selectivity_R)$.

The extracted job profile is independent of the scheduler that is used for executing the jobs: the job profile metrics represent the task durations and the amount of processed data. The collected job profile is "scheduler-agnostic".

4 MapReduce Performance Model

In this section, we design a MapReduce performance model that is based on *i)* the job profile and *ii)* the performance bounds of completion time of different job phases. This model can be used for predicting the job completion time as a function of the input dataset size and allocated resources.

[2] To avoid outliers and improve the robustness of the measured maximum durations, one can use the mean of a few top values instead of the maximum alone.

4.1 General Theoretical Bounds

First, we establish the performance bounds for a makespan (completion time) of a given set of n tasks that is processed by k servers (or by k slots in MapReduce environments).

Let $T_1, T_2, \ldots, T_n$ be the duration of n tasks of a given job. Let k be the number of slots that can each execute one task at a time. The assignment of tasks to slots is done using a simple, online, *greedy* algorithm, i.e., assign each task to the slot with the earliest finishing time.

Let $avg = (\sum_{i=1}^{n} T_i)/n$ and $max = \max_i \{T_i\}$ be the *average* and *maximum* durations of the n tasks respectively.

Makespan Theorem: The makespan of the greedy task assignment is at least $n \cdot avg\ /k$ and at most $(n-1) \cdot avg/k + max$.

The lower bound is trivial, as the best case is when all n tasks are equally distributed among the k slots (or the overall amount of work $n \cdot avg$ is processed as fast as possible by k slots). Thus, the overall makespan is at least $n \cdot avg/k$.

For the upper bound, let us consider the worst case scenario, i.e., the longest task $\hat{T} \in \{T_1, T_2, \ldots, T_n\}$ with duration max is the last processed task. In this case, the time elapsed before the final task $\hat{T}$ is scheduled is at most the following: $(\sum_{i=1}^{n-1} T_i)/k \leq (n-1) \cdot avg/k$. Thus, the makespan of the overall assignment is at most $(n-1) \cdot avg/k + max$.[3] ■

These bounds are particularly useful when $max \ll n \cdot avg/k$, i.e., when the duration of the longest task is small as compared to the total makespan. The difference between lower and upper bounds represents the range of possible job completion times due to non-determinism and scheduling.

4.2 Bounds-Based Completion Time Estimates of a MapReduce Job

Let us consider job J with a given profile extracted from the past job executions. Let J be executed with a new dataset that is partitioned into N_M^J map tasks and N_R^J reduce tasks. Let S_M^J and S_R^J be the number of map and reduce slots allocated to job J respectively.

Let M_{avg} and M_{max} be the average and maximum durations of map tasks (defined by the job J profile). Then, by Makespan Theorem, the lower and upper bounds on the duration of the entire map stage (denoted as T_M^{low} and T_M^{up} respectively) are estimated as follows:

$$T_M^{low} = N_M^J \cdot M_{avg}/S_M^J \tag{1}$$

$$T_M^{up} = (N_M^J - 1) \cdot M_{avg}/S_M^J + M_{max} \tag{2}$$

The reduce stage consists of shuffle (which includes the interleaved sort phase) and reduce phases. Similarly, Makespan Theorem can be directly applied to compute the lower and upper bounds of completion times for reduce phase (T_R^{low}, T_R^{up}) since we have measurements for average and maximum task durations in the reduce phase, the numbers of reduce tasks N_R^J and allocated reduce slots S_R^J.[4]

[3] Similar ideas were explored in the classic papers on scheduling, e.g., to characterize makespan bounds in [4].

[4] For simplicity of explanation, we omit the normalization step of measured durations in job profile with respect to $AvgSize_M^{input}$ and $Selectivity_M$. We will discuss it next in Section 4.3.

The subtlety lies in estimating the duration of the shuffle phase. We distinguish the non-overlapping portion of the *first shuffle* and the task durations in the *typical shuffle* (see Section 3 for definitions). The portion of the typical shuffle phase in the remaining reduce waves is computed as follows:

$$T_{Sh}^{low} = \left(\frac{N_R^J}{S_R^J} - 1\right) \cdot Sh_{avg}^{typ} \tag{3}$$

$$T_{Sh}^{up} = \left(\frac{N_R^J - 1}{S_R^J} - 1\right) \cdot Sh_{avg}^{typ} + Sh_{max}^{typ} \tag{4}$$

Finally, we can put together the formulae for the lower and upper bounds of the overall completion time of job J:

$$T_J^{low} = T_M^{low} + Sh_{avg}^1 + T_{Sh}^{low} + T_R^{low} \tag{5}$$

$$T_J^{up} = T_M^{up} + Sh_{max}^1 + T_{Sh}^{up} + T_R^{up} \tag{6}$$

Note that we can re-write Eq. 5 for T_J^{low} by replacing its parts with more detailed Eq. 1 and Eq. 3 and similar equations for sort and reduce phases as it is shown below:

$$T_J^{low} = \frac{N_M^J \cdot M_{avg}}{S_M^J} + \frac{N_R^J \cdot (Sh_{avg}^{typ} + R_{avg})}{S_R^J} + Sh_{avg}^1 - Sh_{avg}^{typ} \tag{7}$$

This presentation allows us to express the estimates for completion time in a simplified form shown below:

$$T_J^{low} = A_J^{low} \cdot \frac{N_M^J}{S_M^J} + B_J^{low} \cdot \frac{N_R^J}{S_R^J} + C_J^{low}, \tag{8}$$

where $A_J^{low} = M_{avg}$, $B_J^{low} = (Sh_{avg}^{typ} + R_{avg})$, and $C_J^{low} = Sh_{avg}^1 - Sh_{avg}^{typ}$. Eq. 8 provides an explicit expression of a job completion time as a function of map and reduce slots allocated to job J for processing its map and reduce tasks, i.e., as a function of (N_M^J, N_R^J) and (S_M^J, S_R^J). The equation for T_J^{up} can be written similarly.

4.3 Scaling Factors

In the previous section, we showed how to extract the job profile and use it for predicting job completion time when different amounts of resources are used. When the job is executed on a larger dataset the number of map tasks and reduce tasks may be scaled proportionally if the application structure allows it. In some cases, the number of reduce tasks is statically defined, e.g., 24 hours a day, or the number of categories (topics) in Wikipedia, etc. When the job is executed on a larger dataset while the number of reduce tasks is kept constant, the durations of the reduce tasks naturally increase as the size of the intermediate data processed by each reduce task increases. The duration of the map tasks is not impacted because this larger dataset is split into a larger number of map tasks but each map task processes a similar portion of data. The natural attempt might be to derive a single scaling factor for reduce task duration as a function of the amount of processed data, and then use it for the shuffle and reduce phase duration scaling as well. However, this might lead to inaccurate results. The reason is that the shuffle phase

performs data transfer and its duration is mainly defined by the network performance. The reduce phase duration is defined by the application specific computation of the user supplied reduce function and significantly depends on the disk write performance. Thus, the duration scaling in these phases might be different. Consequently, we derive two scaling factors for shuffle and reduce phases separately, each one as a function of the processed dataset size.

Therefore in the staging environment, we perform a set of k experiments ($i = 1, 2, ..., k$) with a given MapReduce job for processing different size input datasets (while keeping the number of reduce tasks constant), and collect the job profile measurements. We derive scaling factors with linear regression in the following way. Let D_i be the amount of intermediate data for processing per reduce task. Note that we can always estimate this amount of intermediate data from the average amount of input data per map task, the number of job map and reduce tasks, and the selectivity metric in the job profile. Let $Sh^{typ}_{i,avg}$ and $R_{i,avg}$ be the job profile measurements for shuffle and reduce phases respectively. Then, using linear regression, we solve the following sets of equations:

$$C^{Sh}_0 + C^{Sh}_1 \cdot D_i = Sh^{typ}_{i,avg}, (i = 1, 2, \cdots, k) \tag{9}$$

$$C^{R}_0 + C^{R}_1 \cdot D_i = R_{i,avg}, (i = 1, 2, \cdots, k) \tag{10}$$

Derived scaling factors (C^{Sh}_0, C^{Sh}_1) for shuffle phase and (C^{R}_0, C^{R}_1) for reduce phase are incorporated in the job profile. When job J processes an input dataset that leads to a different amount of intermediate data D_{new} per reduce task, its profile is updated as $Sh^{typ}_{avg} = C^{Sh}_0 + C^{Sh}_1 \cdot D_{new}$ and $R_{avg} = C^{R}_0 + C^{R}_1 \cdot D_{new}$. Similar scaling factors can be derived for maximum durations Sh^{typ}_{max} and R_{max} as well as for the first shuffle phase measurements. The proposed scaling factors support a general MapReduce performance model where the ratio of map to reduce tasks in the job can vary (or be changed) for different job executions.

For jobs that are routinely executed in the production cluster on new datasets, scaling factors may be derived in an *on-line* fashion by using the same method described above.

4.4 Impact of Failures on the Completion Time Bounds

The performance implications of failures depend on the type of failures (discussed in Section 2). For example, *disk failures* are typical, but their performance implications depend on the amount of data that needs to be reconstructed: for each data block with a number of copies less than the default replication level, Hadoop will reconstruct the additional copies. In this work, we consider a *worker failure* – another typical type of failure which has direct performance implications for a MapReduce job. If the failure happens while the job was running, the failed worker might have completed or in-progress map tasks. All of these map tasks need to be recomputed, since the intermediate data generated by these tasks might be unavailable to current or future reduce tasks. The same applies to the reduce tasks which were in progress on the failed worker: they need to be restarted on a different node. There are reported cases of up to 50% job completion time increase when a single worker failure happens during the reduce stage [5]. The application designer might be interested in evaluating worst case scenarios and deciding on the necessity of additional resources or program changes (e.g., accepting the partial results) as a means of coping with potential failure scenarios.

In order to understand the performance impact of a worker failure on job completion time, we need to consider not only *when* the failure happened, but also whether additional resources in the system can be allocated to the job to compensate for the failed worker. For example, if a worker failure happens in the very beginning of the map stage and the resources of the failed worker are immediately replenished with additional ones, then the lower and upper bounds of job completion time remain practically the same. However, if the failed worker resources are not replenished then the performance bounds are higher. On the other hand, if a worker failure happens during the job's last wave of reduce tasks then all the completed map tasks that reside on the failed node as well as the reduce tasks that were in-progress on this node have to be re-executed, and even if the resources of the failed node are immediately replenished there are serious performance implications of this failure so late during the job execution. The latency for recomputing the map and reduce tasks of the failed node can not be hidden: this computation time is explicitly on the critical path of the job execution and is equivalent of adding entire map and reduce stage latency: $M_{max} + Sh^{typ}_{max} + R_{max}$.

Given the time of failure t_f, we need to quantify the failure impact on the job completion time bounds. Let us consider job J with a given profile, which is partitioned into N^J_M map tasks and N^J_R reduce tasks. Let the worker failure happen at some point of time t_f. There are two possibilities for the job J execution status at the time of failure, it is either in the map or the reduce stage. Using the job profile metrics, we can analyze whether the failure happened during the map or reduce stage. We can do it by estimating the completion time of the map stage (using low or upper bounds of a completion time, or its average). For example, if $t_f \leq T^{low}_M$ then the failure happened during the map stage, otherwise the map stage has been completed and the failure occurred during the job reduce stage. For both scenarios, we need to approximate the number of map and reduce tasks yet to be completed.

- *Case (1):* Let us assume that the job execution is in the map stage at time t_f (i.e., $t_f \leq T^{low}_M$). In order to determine the number of map tasks yet to be processed, we approximate the number of completed ($N^J_{M_{done}}$) and failed ($N^J_{M_{fail}}$) tasks as follows:

$$N^J_{M_{done}} \cdot M_{avg}/S^J_M = t_f \implies N^J_{M_{done}} = \lfloor t_f \cdot S^J_M/M_{avg} \rfloor$$

  If there are W worker nodes in the Hadoop cluster for job J processing and one of them fails, then the number of failed map tasks is:

$$N^J_{M_{fail}} = \lfloor N^J_{M_{done}}/W \rfloor$$

  Thus, the number of map and reduce tasks yet to be processed at time t_f (denoted as N^J_{M,t_f} and N^J_{R,t_f}) are determined as follows:

$$N^J_{M,t_f} = N^J_M - N^J_{M_{done}} + N^J_{M_{fail}} \text{ and } N^J_{R,t_f} = N^J_R$$

- *Case (2):* Let us now assume that the map stage is complete, and the job execution is in the reduce stage at time t_f, $t_f \geq T^{low}_M$ and all the map tasks N^J_M are completed. The number of completed reduce tasks $N^J_{R_{done}}$ at time t_f can be evaluated using Eq. 8:

$$B^{low}_J \cdot \frac{N^J_{R_{done}}}{S^J_R} = t_f - C^{low}_J - A^{low}_J \cdot \frac{N^J_M}{S^J_M}$$

Then the number of failed map and reduce tasks can be approximated as:

$$N^J_{M_{fail}} = \lfloor N^J_M / W \rfloor \text{ and } N^J_{R_{fail}} = \lfloor N^J_{R_{done}} / W \rfloor$$

The remaining map and reduce tasks of job J yet to be processed at time t_f are determined as follows:

$$N^J_{M,t_f} = N^J_{M_{fail}} \text{ and } N^J_{R,t_f} = N^J_R - N^J_{R_{done}} + N^J_{R_{fail}}$$

Let S^J_{M,t_f} and S^J_{R,t_f} be the number of map and reduce slots allocated to job J after the node failure. If the failed resources are not replenished, then the number of map and reduce slots is correspondingly decreased. The number of map and reduce tasks yet to be processed are N^J_{M,t_f} and N^J_{R,t_f} as shown above. Then the performance bounds on the processing time of these tasks can be computed using Eq. 5 and Eq. 6 introduced in Section 4.2. The worker failure is detected only after time δ depending on the value of the *heart beat interval*. Hence, the time bounds are also increased by δ.

This model is easily extensible to estimate the impact of multiple failures.

4.5 SLO-Based Resource Provisioning

When users plan the execution of their MapReduce applications, they often have some *service level objectives* (SLOs) that the job should complete within time T. In order to support the job SLOs, we need to be able to answer a complementary performance question: given a MapReduce job J with input dataset D, how many map and reduce slots need to be allocated to this job so that it finishes within T?

We observe a *monotonicity property* for MapReduce environments. Let job J complete within time T when we allocate some number of map and reduce slots respectively. Clearly, by allocating a higher number of map and reduce slots to a job (i.e., $\hat{M}$), one can only decrease the job completion time. In the light of this monotonicity property, we reformulate the problem as follows. Given a MapReduce job J with input dataset D identify *minimal combinations* (S^J_M, S^J_R) of map and reduce slots that can be allocated to job J so that it finishes within time T? We consider three design choices for answering this question:

1) T is targeted as a *lower bound* of the job completion time. Typically, this leads to the least amount of resources allocated to the job for finishing within deadline T. The lower bound corresponds to an ideal computation under allocated resources and is rarely achievable in real environments.

2) T is targeted as an *upper bound* of the job completion time. Typically, this leads to a more aggressive resource allocations and might lead to a job completion time that is much smaller than T because worst case scenarios are also rare in production settings.

3) Given time T is targeted as the *average* between lower and upper bounds on job completion time. This more balanced resource allocation might provide a solution that enables the job to complete within time T.

Algorithm 1 finds the minimal combinations of map/reduce slots (S_M^J, S_R^J) for one of design choices above, e.g., when T is targeted as a *lower bound* of the job completion time. The algorithm sweeps through the entire range of map slot allocations and finds the corresponding values of reduce slots that are needed to complete the job within time T using a variation of Eq. 8 introduced in Section 4.2. The other cases when T is targeted as the upper bound and the average bound are handled similarly.

Algorithm 1. Resource Allocation Algorithm

Input:
Job profile of J
$(N_M^J, N_R^J) \leftarrow$ Number of map and reduce tasks of J
$(S_M, S_R) \leftarrow$ Total number of map and reduce slots in the cluster
$T \leftarrow$ Deadline by which job must be completed
Output: $P \leftarrow$ Set of plausible resource allocations (S_M^J, S_R^J)

for $S_M^J \leftarrow$ MIN(N_M^J, S_M) **to** 1 **do**
  Solve the equation $\frac{A_J^{low} \cdot N_M^J}{S_M^J} + \frac{B_J^{low} \cdot N_R^J}{S_R^J} = T - C_J^{low}$ for S_R^J
  if $0 < S_R^J \leq S_R$ **then**
    $P \leftarrow P \cup (S_M^J, S_R^J)$
  else
    // Job cannot be completed within deadline T
    // with the allocated map slots
    Break out of the loop
  end if
end for

Note, that the complexity of the proposed Algorithm 1 is $O(min(N_M^J, S_m))$ and thus linear in the number of map slots.

5 Evaluation

We perform our experiments on 66 HP DL145 GL3 machines. Each machine has four AMD 2.39MHz cores, 8 GB RAM and two 160GB hard disks (7200rpm SATA). The machines are set up in two racks. The 1Gb network interfaces of the machines in the same rack are connected to a Gigabit Procurve 2650 switch. The racks are interconnected using a ProCurve 2900 switch. We use Hadoop 0.20.2 with two machines as job master and the DFS master. The remaining 64 machines are used as worker nodes, each configured with a single map and reduce slot. The blocksize of the file system is set to 64MB and the replication level is set to 3. We disabled speculation in all our experiments as it did not lead to any significant improvements.

In order to validate our model, we use four representative MapReduce applications:

1. **Twitter:** This application uses the 25GB twitter dataset created by Kwak et. al. [6] containing an edge-list of twitter userids. Each edge (i, j) means that user i follows user j. The Twitter application counts the number of asymmetric links in the dataset, that is, $(i, j) \in E$, but $(j, i) \notin E$.

2. **Sort:** The Sort application sorts 64GB of random data generated using random text writer in GridMix2[5]. It uses identity map and reduce tasks, since the framework performs the actual sorting.
3. **WikiTrends:** We use the data from Trending Topics (TT)[6]: Wikipedia article traffic logs that were collected (and compressed) every hour in the months of April to August 2010. Our MapReduce application counts the number of times each article has been visited according to the given input dataset, which is very similar to the job that is run periodically by TT.
4. **WordCount:** It counts the word frequencies in 27 GB of Wikipedia article text corpus. The map task tokenizes each line into words, while the reduce task counts the occurrence of each word.

5.1 Performance Invariants

In our first set of experiments, we aim to validate whether the metrics, that we chose for the inclusion in the job profile, indeed represent performance invariants across different executions of the job on the same input dataset. To this end, we execute our MapReduce jobs on the same datasets and the same Hadoop cluster but with a variable number of map and reduce slots: *i)* 64 map and 32 reduce slots, *ii)* 16 map and 16 reduce slots. The collected job profile metrics are summarized in Table 1. The average input size for WikiTrends is 59.15MB and 64MB for the other applications. We observe that the average duration metrics are within 10% of each other. The maximum durations show slightly higher variance. Each experiment is performed 10 times, and again, collected metrics exhibit less than 10% variation. From these measurements, we conclude that job profile indeed accurately captures application behavior characteristics and reflect the job performance invariants.

Table 1. Job profiles of the four MapReduce applications

Job	Map slots	Reduce slots	Map Task duration (s)			Map Selectivity	1st Shuffle (s)		Typ. Shuffle (s)		Reduce (s)		Reduce Selectivity
			Min	Avg	Max		Avg	Max	Avg	Max	Avg	Max	
Twitter	64	32	26	30	42	3.24	8	11	37	40	22	44	3.2×10^{-8}
	16	16	26	29	43	3.24	7	10	37	41	21	41	3.2×10^{-8}
Sort	64	32	2	5	16	1.00	7	13	30	50	53	75	1.00
	16	16	2	4	14	1.00	8	11	30	51	54	73	1.00
WordCount	64	32	5	34	40	1.31	8	11	24	30	11	14	0.46
	16	16	5	34	41	1.31	7	10	23	28	10	14	0.46
WikiTrends	64	32	66	99	120	9.98	13	27	115	142	26	34	0.37
	16	16	65	98	121	9.98	14	27	113	144	26	32	0.37

5.2 Scaling Factors

We execute WikiTrends and WordCount applications on gradually increasing datasets with a fixed number of reduce tasks for each application. Our intent is to measure the trend of the shuffle and reduce phase durations (average and maximum) and validate the linear regression approach proposed in Section 4.3. The following table below gives the details of the experiments and the resulting co-efficients of linear regression, i.e., scaling factors of shuffle and reduce phase durations derived for these applications.

[5] http://hadoop.apache.org/mapreduce/docs/current/gridmix.html
[6] http://trendingtopics.org

Parameters	WikiTrends	WordCount
Size of input dataset	4.3GB to 70GB	4.3GB to 43GB
Number of map tasks	70 to 1120	70 to 700
Number of reduce tasks	64	64
Number of map, reduce slots	64, 32	64, 32
$C^{Sh}_{0,avg}, C^{Sh}_{1,avg}$	16.08, 2.44	6.92, 0.66
$C^{Sh}_{0,max}, C^{Sh}_{1,max}$	10.75, 2.29	11.28, 0.71
$C^{R}_{0,avg}, C^{R}_{1,avg}$	11.45, 0.56	4.09, 0.22
$C^{R}_{0,max}, C^{R}_{1,max}$	7.96, 0.43	7.26, 0.24

Figure 5 shows that the trends are indeed linear for WikiTrends and WordCount. While we show multiple points in Fig. 5, typically, at least two points are needed for deriving the scaling factors. The accuracy will improve with accumulated job execution points. Note that the lines do not pass through the origin and hence the durations are not directly proportional to the dataset size. We observe similar results for Twitter and Sort applications but do not include them in the paper due to lack of space.

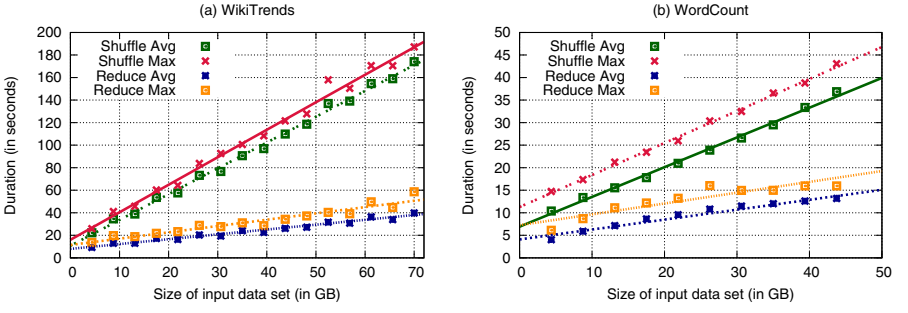


Fig. 5. Linear scaling of shuffle and reduce durations for WikiTrends and WordCount

5.3 Performance Bounds of Job Completion Times

In section 4, we designed performance bounds that can be used for estimating the completion time of MapReduce application with a given job profile. The expectations are that the job profile can be built using a set of job executions for processing small size input datasets, and then this job profile can be used for predicting the completion time of the same application processing a larger input dataset. Therefore, in these experiments, first, the job profiles are built using the three trials on small datasets (e.g., 4.3, 8.7 and 13.1 GB for WordCount) with different numbers of map and reduce slots. After that, by applying linear regression to the extracted job profiles from these runs, we determine the scaling factors for shuffle and reduce phases of our MapReduce jobs. The derived scaling factors are used to represent the job performance characteristics and to extrapolate the duration of the shuffle and reduce phases when the same applications are used for processing larger input datasets with parameters shown in the following table:

Parameters	Twitter	Sort	WikiTrends	WordCount
# of map tasks	370	1024	168	425
# of reduce tasks	64	64	64	64
# of map slots	64	64	64	64
# of reduce slots	16	32	8	8

Finally, by using the updated job profiles and applying the formulae described in Section 4, we predict the job completion times.

The results of these experiments are shown in Figure 6. We observe that our model accurately bounds the measured job completion time, and if we use for prediction the average of lower and upper bounds (denoted T_J^{avg}) then the relative error between T_J^{avg} and the measured job completion time is less than **10%** in all cases. The predicted upper bound on the job completion time T_J^{up} can be used for ensuring SLOs.

Fig. 6. Comparison of predicted and measured job completion times

The solid fill color within the bars in Figure 6 represent the reduce stage duration, while the pattern portion reflects the duration of the map stage. For Sort and WordCount, bounds derived from the profile provide a good estimate for map and reduce stage durations. For WikiTrends, we observe a higher error in the estimation of the durations, mostly, due to the difference in processing of the unequal compressed files as inputs. For Twitter, we observe that even though the predicted minimum and maximum are farther apart because of the difference in average and maximum durations of it phases, the average of the two bounds would provide a good estimate of the measured duration.

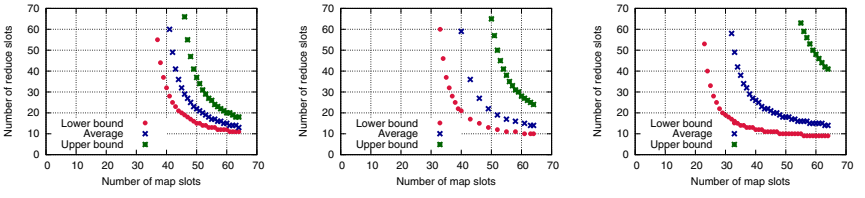
The power of the proposed approach is that it offers a compact job profile that can be derived from past executions (while processing smaller dataset) and then used for completion time prediction of the job on a large input dataset while also using different amount of resources assigned to the job.

5.4 SLO-Based Resource Provisioning

In this section, we perform experiments to validate the accuracy of the SLO-based resource provisioning model introduced in Section 4.5. It operates over the following inputs *i)* a job profile built in the staging environment using smaller datasets, *ii)* the targeted amount of input data for processing, *iii)* the required job completion time. We aim to evaluate the accuracy of resource allocations recommended by the model for completing the job within a given deadline.

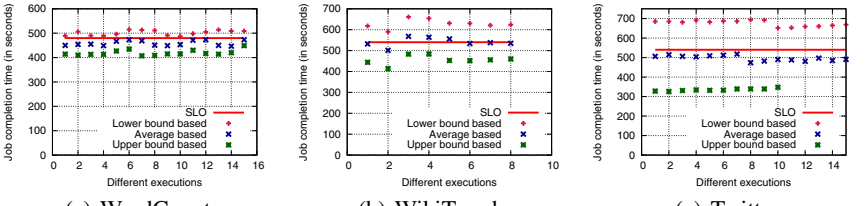
Figure 7 shows a variety of plausible solutions (the outcome of the SLO-based model) for WordCount, WikiTrends and Twitter with a given deadline D= 8, 9, and

8 minutes respectively. The X and Y axes of the graph show the number of map and reduce slots respectively that need to be allocated in order to meet the job's deadline. Figure 7 presents three curves that correspond to three possible design choices for computing the required map/reduce slots as discussed in Section 4.5: when the given time T is targeted as the lower bound, upper bound, or the average of the lower and upper bounds. As expected, the recommendation based on the upper bound (worst case scenario) suggests more aggressive resource allocations with a higher number of map and reduce slots as compared to the resource allocation based on the lower bound. The difference in resource allocation is influenced by the difference between the lower and upper bounds. For example, WordCount has very tight bounds which lead to more similar resource allocations based on them. For Twitter, the difference between the lower and upper bounds of completion time estimates is wider, which leads to a larger difference in the resource allocation options.



(a) WordCount (D = 8 mins) (b) WikiTrends (D = 9 mins) (c) Twitter (D = 9 mins)

Fig. 7. Different allocation curves based on bounds for different deadlines



(a) WordCount (b) WikiTrends (c) Twitter

Fig. 8. Do we meet deadlines using the bounds?

Next, we perform a set of experiments with the applications on our 66-node Hadoop cluster. We sample each curve in Figure 7, and execute the applications with recommended allocations of map and reduce slots in our Hadoop cluster to measure the actual job completion times. Figure 8 summarizes the results of these experiments. If we base our resource computation on the lower bound of completion time, it corresponds to the "ideal" scenario. The model based on lower bounds suggests insufficient resource allocations: almost all the job executions with these allocations have missed their deadline. The closest results are obtained if we use the model that is based on the average of lower and upper bounds of completion time. However, in many cases, the measured completion time can exceed a given deadline (by 2-7%). If we base our computation on the upper bounds of completion time, the model over provisions resources. While all the job executions meet their deadline, the measured job completion times are lower

than the target SLO, sometimes by as much as 40%. The resource allocation choice will depend on the user goals and his requirements on how close to a given SLO the job completion time should be. The user considerations might also take into account the service provider charging schema to evaluate the resource allocation alternatives on the curves shown in Figure 7.

5.5 Prediction of Job Completion Time with Failures

In this section, we validate the model for predicting the job completion time with failures introduced in Section 4.4. For this experiment, we set the heartbeat interval to $3s$. If a heartbeat is not received in the last $20s$, the worker node is assumed to have failed. We use the WikiTrends application which consists of 720 map and 120 reduce tasks. The application is allocated 60 map and 60 reduce slots. The WikiTrends execution with given resources takes $t = 1405s$ to complete under normal circumstances. Figure 9 shows a set of two horizontal lines that correspond to lower and upper bounds of the job completion time under normal case.

Then, using the model with failures introduced in Section 4.4, we compute the lower and upper bounds for job completion time when a failure happens at time t_f (time is represented by X-axes). The model considers two different scenarios: when resources of the failed node are 1) replenished and 2) not replenished. These scenarios are reflected in Figure 9 (a) and (b) respectively. Figure 9 shows the predicted lower and upper bounds (using the model with failures) along with the measured job completion time when the worker process is killed at different points in the course of the job execution.

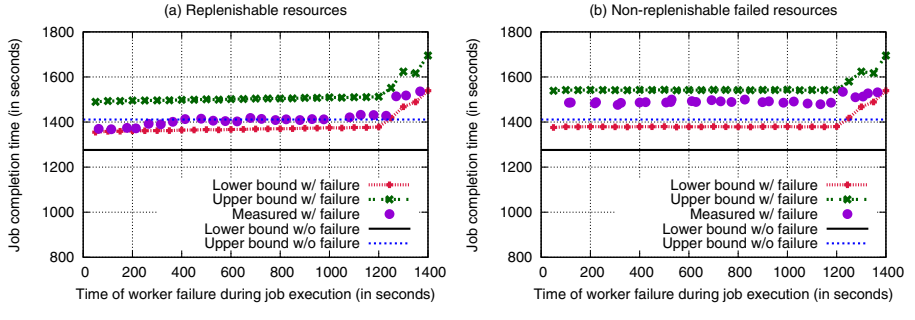


Fig. 9. Model with failures: two cases with replenishable resources and non-replenishable failed resources

The shape of the lines (lower and upper bounds) for job completion time with failures is quite interesting. While the completion time with failures increases compared to the regular case, but this increase is practically constant until approximately $t = 1200s$. The map stage completes at $t = 1220(\pm 10)s$. So, the node failure during the map stage has a relatively mild impact on the overall completion time, especially when the failed resources are replenished. However, if the failure happens in the reduce stage (especially towards the end of the job processing) then it has a more significant impact on the

job completion time even if the failed node resources are replenished. Note that the measured job completion time with failures stays within predicted bounds, and hence the designed model can help the user to estimate the worst case scenario.

6 Related Work

Originally, MapReduce (and its open source implementation Hadoop) was designed for periodically running large batch workloads. With a primary goal of minimizing the job makespan the simple *FIFO* scheduler was very efficient and there was no need for special resource provisioning since the entire cluster resources could be used by the submitted job. As the number of users sharing the same MapReduce cluster increased, a new *Capacity* scheduler [7] was introduced to support more efficient cluster sharing. Capacity scheduler partitions the resources into pools and provides separate queues and priorities for each pool. By enabling partitioning of the cluster resources, the users and system administrators do need to answer an additional question: how much resources do the time-sensitive jobs require and how to translate these requirements in the capacity scheduler settings? This question is still open: there are many research efforts discussed below that aim to design a MapReduce performance model for resource provisioning and predicting the job completion time.

In order to maintain fairness between different users, the *Hadoop Fair Scheduler* (HFS) [8] allocates equal shares to each of the users running the MapReduce jobs. It also tries to maximize data locality by delaying the scheduling of the task, if no local data is available. Similar fairness and data locality goals are pursued in *Quincy* scheduler [9] proposed for the Dryad environment [10]. However, both HFS and Quincy do not provide any special support for achieving the application performance goals and the service level objectives (SLOs). Dynamic proportional share scheduling [11] allows users to bid for map and reduce slots by adjusting their spending over time. While this approach allows dynamically controlled resource allocation, it is driven by economic mechanisms rather than a performance model and/or application profiling.

FLEX [12] extends HFS by proposing a special slot allocation schema that aims to optimize explicitly some given scheduling metric. FLEX relies on the speedup function of the job (for map and reduce stages) that produces the job execution time as a function of the allocated slots. This function aims to represent the application model, but it is not clear how to derive this function for different applications and for different sizes of input datasets. FLEX does not provide a technique for job profiling and detailed MapReduce performance model, but instead uses a set of simplifying assumptions about the job execution, tasks durations and job progress over time.

Polo et al. [13] introduce an online job completion time estimator which can be used for adjusting the resource allocations of different jobs. However, their estimator tracks the progress of the map stage alone and has no information or control over the reduce stage. Ganapathi et al. [14] use Kernel Canonical Correlation Analysis to predict the performance of MapReduce workloads. However, they concentrate on Hive queries and do not attempt to model the actual execution of the MapReduce job. The authors discover the feature vectors through statistical correlation.

Morton et al. [15] propose *ParaTimer* for estimating the progress of parallel queries expressed as Pig scripts [16] that can translate into directed acyclic graphs (DAGs) of MapReduce jobs. In their earlier work [17], they designed *Parallax* – a progress

estimator that aims to predict the completion time of a limited class of Pig queries that translate into a sequence of MapReduce jobs. In both papers, instead of a detailed profiling technique that is designed in our work, the authors rely on earlier debug runs of the same query for estimating throughput of map and reduce stages on the input data samples provided by the user. The approach is based on precomputing the expected schedule of all the tasks, and therefore identifying all the pipelines (sequences of MapReduce jobs) in the query. The approach relies on a simplistic assumption that map (reduce) tasks of the same job have the same duration.

Phan et al. [18] aim to build an optimal schedule for a set of MapReduce jobs with given deadlines. The authors investigate different factors that impact job performance and its completion time such as ratio of slots to core, the number of concurrent jobs, data placement, etc. MapReduce jobs with a single map and reduce waves are considered, and the scheduling problem is formulated as a constraint satisfaction problem (CSP). There are some other simplifications in MapReduce job processing where the data transfer (shuffle and sort) is considered as a separate (intermediate) phase between map and reduce tasks while in reality the shuffle phase overlaps significantly with map stage. All these assumptions and the CSP complexity issues, make it difficult to extend the proposed approach for a general case.

Originally, Hadoop was designed for homogeneous environments. There has been recent interest [19] in heterogeneous MapReduce environments. Our approach and the proposed scaling technique will efficiently work in heterogeneous MapReduce environments. In a heterogeneous cluster, the slower nodes would be reflected in the longer tasks durations, and they all would contribute to the average and maximum task durations in the job profile. While we do not explicitly consider different types of nodes, their performance is reflected in the job profile and used in the future prediction.

In our earlier work, we proposed a framework, called ARIA [20], for a Hadoop deadline-based scheduler which extracts and utilizes the job profiles from the past executions. The shortcoming of the earlier work is that it does not have scaling factors to adjust the extracted profile and lacks the ability for job profiling on smaller datasets. Our earlier workshop paper [21] proposed the idea of scaling factors. This work builds on it providing a detailed evaluation and also models the impact of failures.

Much of the recent work also focuses on anomaly detection, stragglers and outliers control in MapReduce environments [19, 22–24] as well as on optimization and tuning cluster parameters and testbed configuration [25, 26]. While this work is orthogonal to our research, the results are important for performance modeling in MapReduce environments. Providing more reliable, well performing, balanced environment enables reproducible results, consistent job executions and supports more accurate performance modeling and predictions.

7 Conclusion

We have designed a novel framework that aims to enrich private and public clouds offering with an automated SLO-driven resource sizing and provisioning service in MapReduce environments. While there are several companies that offer Hadoop clusters for rent, they do not provide additional performance services to answer a set of typical questions: How much resources the user application needs in order to achieve certain performance goals and complete data processing by a certain deadline. What is the

impact of failures on job completion time? To answer these questions, we introduced a novel automated profiling technique for MapReduce applications by building a compact but representative job profile in a staging environment. The approach allows executing a given application on the set of small input datasets. Then by applying a special scaling technique and designed performance models, one can estimate the resources required for processing a targeted large dataset while meeting given SLOs. We also designed a performance model for estimating the impact of failures on MapReduce applications.

We validated the accuracy of our approach and designed performance models using a set of realistic applications in the 66-node Hadoop cluster. The accuracy of the results depends on resource contention, especially, the network contention in the production Hadoop cluster. In our testbed, the network was not a bottleneck, and it led to the accurate prediction for job completion time. Typically, service providers tend to over provision network resources to avoid undesirable side effects of network contention. At the same time, it is an interesting modeling question whether such a network contention factor can be introduced, measured, and incorporated in the proposed performance models. Another interesting future work is the resource provisioning of more complex applications (e.g., Pig queries) that are defined as a composition of MapReduce jobs and meeting SLO requirements for a given set of MapReduce jobs.

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
2. O'Malley, O., Murthy, A.: Winning a 60 second dash with a yellow elephant (2009)
3. White, T.: Hadoop: The Definitive Guide, Page 6. Yahoo Press
4. Graham, R.: Bounds for certain multiprocessing anomalies. Bell System Tech. Journal 45, 1563–1581 (1966)
5. Ko, S., Hoque, I., Cho, B., Indranil Gupta, I.: On Availability of Intermediate Data in Cloud Computations. In: Proceedings of HotOS 2009 (2009)
6. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a social network or a news media?. In: Proc. of Intl. Conference on World Wide Web, pp. 591–600. ACM (2010)
7. Apache, Capacity Scheduler Guide (2010), http://hadoop.apache.org/common/docs/r0.20.1/capacity_scheduler.html
8. Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., Stoica, I.: Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In: Proc. of EuroSys, pp. 265–278. ACM (2010)
9. Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., Goldberg, A.: Quincy: fair scheduling for distributed computing clusters. In: SOSP. ACM (2009)
10. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. ACM SIGOPS OS Review 41(3) (2007)
11. Sandholm, T., Lai, K.: Dynamic Proportional Share Scheduling in Hadoop. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) JSSPP 2010. LNCS, vol. 6253, pp. 110–131. Springer, Heidelberg (2010)
12. Wolf, J., Rajan, D., Hildrum, K., Khandekar, R., Kumar, V., Parekh, S., Wu, K.-L., Balmin, A.: FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads. In: Gupta, I., Mascolo, C. (eds.) Middleware 2010. LNCS, vol. 6452, pp. 1–20. Springer, Heidelberg (2010)
13. Polo, J., Carrera, D., Becerra, Y., Torres, J., Ayguadé, E., Steinder, M., Whalley, I.: Performance-driven task co-scheduling for mapreduce environments. In: 12th IEEE/IFIP Network Operations and Management Symposium (2010)

14. Ganapathi, A., Chen, Y., Fox, A., Katz, R., Patterson, D.: Statistics-driven workload modeling for the cloud. In: Proceedings of SMDB (2010)
15. Morton, K., Balazinska, M., Grossman, D.: ParaTimer: a progress indicator for MapReduce DAGs. In: Proceedings of SIGMOD, pp. 507–518. ACM (2010)
16. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Proceedings of SIGMOD, pp. 1099–1110. ACM (2008)
17. Morton, K., Friesen, A., Balazinska, M., Grossman, D.: Estimating the progress of MapReduce pipelines. In: Proceedings of ICDE, pp. 681–684. IEEE (2010)
18. Phan, L., Zhang, Z., Loo, B., Lee, I.: Real-time MapReduce Scheduling. In: Technical Report No. MS-CIS-10-32, University of Pennsylvania (2010)
19. Zaharia, M., Konwinski, A., Joseph, A., Katz, R., Stoica, I.: Improving mapreduce performance in heterogeneous environments. In: OSDI (2008)
20. Verma, A., Cherkasova, L., Campbell, R.: ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. In: Proc. of ICAC (2011)
21. Verma, A., Cherkasova, L., Campbell, R.: SLO-Driven Right-Sizing and Resource Provisioning of MapReduce Jobs. In: Proc. of Workshop on Large Scale Distributed Systems and Middleware, LADIS (2011)
22. Tan, J., Pan, X., Kavulya, S., Marinelli, E., Gandhi, R., Narasimhan, P.: Kahuna: Problem Diagnosis for MapReduce-based Cloud Computing Environments. In: NOMS (2010)
23. Tan, J., Kavulya, S., Gandhi, R., Narasimhan, P.: Visual, Log-Based Causal Tracing for Performance Debugging of MapReduce Systems. In: ICDCS, pp. 795–806. IEEE (2010)
24. Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., Harris, E.: Reining in the Outliers in Map-Reduce Clusters using Mantri. In: OSDI (2010)
25. Intel, Optimizing Hadoop* Deployments, (2010), http://communities.intel.com/docs/DOC-4218
26. Kambatla, K., Pathak, A., Pucha, H.: Towards optimizing hadoop provisioning in the cloud. In: Proc. of the First Workshop on Hot Topics in Cloud Computing (2009)

Resource-Aware Adaptive Scheduling for MapReduce Clusters

Jordà Polo[1], Claris Castillo[2], David Carrera[1], Yolanda Becerra[1], Ian Whalley[2], Malgorzata Steinder[2], Jordi Torres[1], and Eduard Ayguadé[1]

[1] Barcelona Supercomputing Center (BSC) and
Technical University of Catalonia (UPC)
{jordap,dcarrera,yolandab,torres,eduard}@ac.upc.edu
[2] IBM T.J. Watson Research Center
{claris,inw,steinder}@us.ibm.com

Abstract. We present a resource-aware scheduling technique for MapReduce multi-job workloads that aims at improving resource utilization across machines while observing completion time goals. Existing MapReduce schedulers define a static number of slots to represent the capacity of a cluster, creating a fixed number of execution slots per machine. This abstraction works for homogeneous workloads, but fails to capture the different resource requirements of individual jobs in multi-user environments. Our technique leverages job profiling information to dynamically adjust the number of slots on each machine, as well as workload placement across them, to maximize the resource utilization of the cluster. In addition, our technique is guided by user-provided completion time goals for each job. Source code of our prototype is available at [1].

Keywords: MapReduce, scheduling, resource-awareness, performance management.

1 Introduction

In recent years, the industry and research community have witnessed an extraordinary growth in research and development of data-analytic technologies. Pivotal to this phenomenon is the adoption of the MapReduce programming paradigm [2] and its open-source implementation Hadoop [3].

Pioneer implementations of MapReduce [3] have been designed to provide overall system goals (e.g., job throughput). Thus, support for user-specified goals and resource utilization management have been left as secondary considerations at best. We believe that both capabilities are crucial for the further development and adoption of large-scale data processing. On one hand, more users wish for ad-hoc processing in order to perform short-term tasks [4]. Furthermore, in a Cloud environment users pay for resources used. Therefore, providing consistency between price and the quality of service obtained is key to the business model of the Cloud. Resource management, on the other hand, is also important as Cloud providers are motivated by profit and hence require both high levels of automation and resource utilization while avoiding bottlenecks.

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 187–207, 2011.

The main challenge in enabling resource management in Hadoop clusters stems from the resource model adopted in MapReduce. Hadoop expresses capacity as a function of the number of tasks that can run concurrently in the system. To enable this model the concept of typed-'slot' was introduced as the schedulable unit in the system. 'Slots' are bound to a particular type of task, either reduce or map, and one task of the appropriate type is executed in each slot. The main drawback of this approach is that slots are fungible across jobs: a task (of the appropriate type) can execute in any slot, regardless of the job of which that task forms a part. This loose coupling between scheduling and resource management limits the opportunity to efficiently control the utilization of resources in the system. Providing support for user-specified 'goals' in MapReduce clusters is also challenging, due to high variability induced by the presence of outlier tasks (tasks that take much longer than other tasks) [5–8]. Solutions to mitigate the detrimental impact of such outliers typically rely on scheduling techniques such as speculative scheduling [9], and killing and restarting of tasks [5]. These approaches, however, may result in wasted resources and reduced throughput. More importantly, all existing techniques are based on the typed-slot model and therefore suffer from the aforementioned limitations.

In this work we present RAS [1], a Resource-aware Adaptive Scheduler for MapReduce capable of improving resource utilization and which is guided by completion time goals. In addition, RAS addresses the system administration issue of configuring the number of slots for each machine, which—as we will demonstrate—has no single, homogeneous, and static solution for a multi-job MapReduce cluster.

While existing work focuses on the current typed-slot model—wherein the number of tasks per worker is fixed throughout the lifetime of the cluster, and slots can host tasks from any job—our approach offers a novel resource-aware scheduling technique which advances the state of the art in several ways:

- Extends the abstraction of 'task slot' to 'job slot'. A 'job slot' is job specific, and has an associated resource demand profile for map and reduce tasks.
- Leverages resource profiling information to obtain better utilization of resources and improve application performance.
- Adapts to changes in resource demand by dynamically allocating resources to jobs.
- Seeks to meet soft-deadlines via a utility-based approach.
- Differentiates between map and reduce tasks when making resource-aware scheduling decisions.

The structure of the paper is as follows. We give an overview of Hadoop in Section 2. The scheduler's design and implementation is described in detail in Section 3. An evaluation of our prototype in a real cluster is is presented in Section 4. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 MapReduce and Hadoop

The execution of a MapReduce job is divided into a Map phase and a Reduce phase. In the Map phase, the map tasks of the job are run. Each map task comprises the execution of the job's *map()* function as well as some supporting actions (for example, data sorting). The data output by each map task is written into a circular memory buffer—when this buffer reaches a threshold, its content is sorted by key and flushed to a temporary file. These files are then served via HTTP to machines running reduce tasks. Reduce tasks are divided into three sub-phases: shuffle, sort and reduce. The shuffle sub-phase is responsible for copying the map output from the machines hosting maps to the reducer's machine. The sort sub-phase sorts the intermediate data by key. Finally, the reduce sub-phase, which runs the job's *reduce()* function, starts after the keys destined for the particular reducer have been copied and sorted, and the final result is then written to the distributed file system.

Hadoop [3] is an open source implementation of MapReduce provided by the Apache Software Foundation. The Hadoop architecture follows the master/slave paradigm. It consists of a master machine responsible for coordinating the distribution of work and execution of jobs, and a set of worker machine responsible for performing work assigned by the master. The master and slaves roles are performed by the 'JobTracker' and 'TaskTracker' processes, respectively. The singleton JobTracker partitions the input data into 'input splits' using a splitting method defined by the programmer, populates a local task-queue based on the number of obtained input splits, and distributes work to the TaskTrackers that in turn process individual splits. Work units are represented by 'tasks' in this framework. There is one map task for every input split generated by the JobTracker. The number of reduce tasks is defined by the user. Each TaskTracker controls the execution of the tasks assigned to its hosting machine.

In Hadoop resources are abstracted into typed slots. A slot is bound to a particular type of task (map or reduce), but is fungible across jobs. The slot is the schedulable unit, and as such is the finest granularity at which resources are managed in the system. The number of slots per TaskTracker determines the maximum number of concurrent tasks that are allowed to run in the worker machine. Since the number of slots per machine is fixed for the lifetime of the machine, existing schedulers implement a task-assignment solver. The default scheduler in Hadoop, for example, implements the FIFO (First-In First-Out) scheduling strategy. More recently, the Fair Scheduler [9] assigns jobs to 'pools', and then guarantees a certain minimum number of slots to each pool.

3 Resource-Aware Adaptive Scheduler

The driving principles of RAS are resource awareness and continuous job performance management. The former is used to decide task placement on TaskTrackers over time, and is the main object of study of this paper. The latter is used to estimate the number of tasks to be run in parallel for each job in order to meet some performance objectives, expressed in RAS in the form of completion time goals, and was extensively evaluated and validated in [6].

In order to enable this resource awareness, we introduce the concept of 'job slot'. A job slot is an execution slot that is bound to a particular job, and a particular task type (reduce or map) within that job. This is in contrast to the traditional approach, wherein a slot is bound only to a task type regardless of the job. In the rest of the paper we will use the terms 'job slot' and 'slot' interchangeably. This extension allows for a finer-grained resource model for MapReduce jobs. Additionally, RAS determines the number of job slots, and their placement in the cluster, dynamically at run-time. This contrasts sharply with the traditional approach of requiring the system administrator to statically and homogeneously configure the slot count and type on a cluster. This eases the configuration burden and improves the behavior of the MapReduce cluster.

Completion time goals are provided by users at job submission time. These goals are treated as soft deadlines in RAS as opposed to the strict deadlines familiar in real-time environments: they simply guide workload management.

3.1 Problem Statement

We are given a set of MapReduce jobs $\mathcal{J} = \{1, \ldots, J\}$, and a set of TaskTrackers $\mathcal{TT} = \{1, \ldots, TT\}$. We use j and tt to index into the sets of jobs and TaskTrackers, respectively. With each TaskTracker tt we associate a series of resources, $\mathcal{R} = \{1, \ldots, R\}$. Each resource of TaskTracker tt has an associated capacity $\Omega_{tt,1}, \ldots, \Omega_{tt,r}$. In our work we consider disk bandwidth, memory, and CPU capacities for each TaskTracker. Note that extending the algorithm to accommodate for other resources, e.g., storage capacity, is straightforward.

A MapReduce job (j) is composed of a set of tasks, already known at submission time, that can be divided into map tasks and reduce tasks. Each TaskTracker tt provides to the cluster a set of job-slots in which tasks can run. Each job-slot is specific for a particular job, and the scheduler will be responsible for deciding the number of job-slots to create on each TaskTracker for each job in the system.

Each job j can be associated with a completion time goal, T^j_{goal}, the time at which the job should be completed. When no completion time goal is provided, the assumption is that the job needs to be completed at the earliest possible time. Additionally, with each job we associate a resource consumption profile. The resource usage profile for a job j consists of a set of average resource demands $\mathcal{D}_j = \{\Gamma_{j,1}, \ldots, \Gamma_{j,r}\}$. Each resource demand consists of a tuple of values. That is, there is one value associated for each task type and phase (map, reduce in shuffle phase, and reduce in reduce phase, including the final sort).

We use symbol P to denote a placement matrix of tasks on TaskTrackers, where cell $P_{j,tt}$ represents the number of tasks of job j placed on TaskTracker tt. For simplicity, we analogously define P^M and P^R, as the placement matrix of Map and Reduce tasks. Notice that $P = P^M + P^R$. Recall that each task running in a TaskTracker requires a corresponding slot to be created before the task execution begins, so hereafter we assume that placing a task in a TaskTracker implies the creation of an execution slot in that TaskTracker.

Based on the parameters described above, the goal of the scheduler presented in this paper is to determine the best possible placement of tasks across the TaskTrackers as to maximize resource utilization in the cluster while observing

the completion time goal for each job. To achieve this objective, the system will dynamically manage the number of job-slots each TaskTracker will provision for each job, and will control the execution of their tasks in each job-slot.

3.2 Architecture

Figure 1 illustrates the architecture and operation of RAS. The system consists of five components: Placement Algorithm, Job Utility Calculator, Task Scheduler, Job Status Updater and Job Completion Time Estimator.

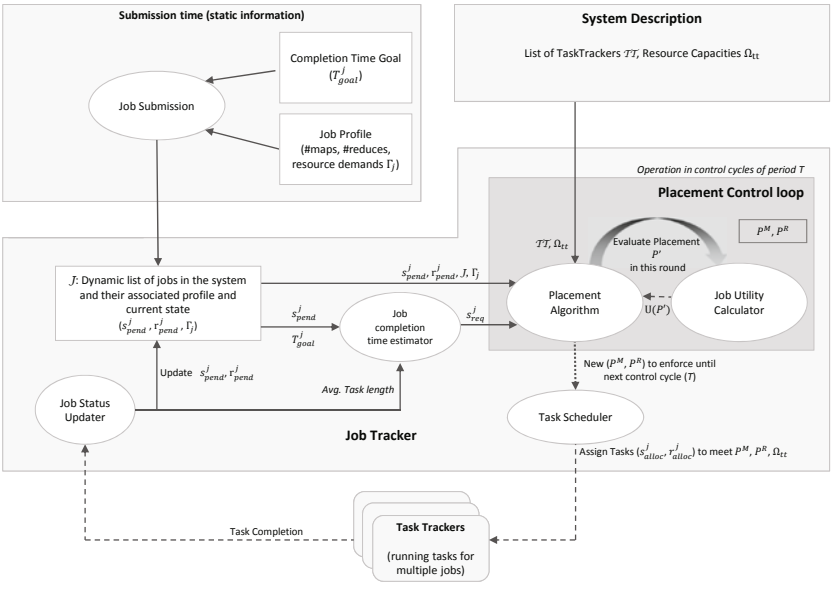


Fig. 1. System architecture

Most of the logic behind RAS resides in the JobTracker. We consider a scenario in which jobs are dynamically submitted by users. Each submission includes both the job's completion time goal (if one is provided) and its resource consumption profile. This information is provided via the job configuration XML file. The JobTracker maintains a list of active jobs and a list of TaskTrackers. For each active job it stores a descriptor that contains the information provided when the job was submitted, in addition to state information such as number of pending tasks. For each TaskTracker (TT) it stores that TaskTracker's resource capacity (Ω_{tt}).

For any job j in the system, let s^j_{pend} and r^j_{pend} be the number of map and reduce tasks pending execution, respectively. Upon completion of a task, the TaskTracker notifies the **Job Status Updater**, which triggers an update of s^j_{pend} and r^j_{pend} in the job descriptor. The Job Status Updater also keeps track of the average task length observed for every job in the system, which is later used to estimate the completion time for each job.

The **Job Completion Time Estimator** estimates the number of *map* tasks that should be allocated concurrently (s^j_{req}) to meet the completion time goal of each job. To perform this calculation it relies on the completion time goal T^j_{goal}, the number of pending *map* tasks (s^j_{pend}), and the observed average task length. Notice that the scenario we focus on is very dynamic, with jobs entering and leaving the system unpredictably, so the goal of this component is to provide estimates of s^j_{req} that guide resource allocation. This component leverages the techniques already described in [6] and therefore we will not provide further details in this paper.

The core of RAS is the Placement Control loop, which is composed of the **Placement Algorithm** and the **Job Utility Calculator**. They operate in control cycles of period T, which is of the order of tens of seconds. The output of their operation is a new placement matrix P that will be active until the next control cycle is reached (current time + T). A short control cycle is necessary to allow the system to react quickly to new job submissions and changes in the task length observed for running jobs. In each cycle, the Placement Algorithm component examines the placement of tasks on TaskTrackers and their resource allocations, evaluates different candidate placement matrices and proposes the final output placement to be enforced until next control cycle. The Job Utility Calculator calculates a utility value for an input placement matrix which is then used by the Placement Algorithm to choose the best placement choice available.

The **Task Scheduler** is responsible for enforcing the placement decisions, and for moving the system smoothly between a placement decision made in the last cycle to a new decision produced in the most recent cycle. The Task Scheduler schedules tasks according to the placement decision made by the Placement Controller. Whenever a task completes, it is the responsibility of the Task Scheduler to select a new task to execute in the freed slot, by providing a task of the appropriate type from the appropriate job to the given TaskTracker.

In the following sections we will concentrate on the problem solved by the Placement Algorithm component in a single control cycle.

3.3 Performance Model

To measure the performance of a job given a placement matrix, we define a utility function that combines the number of map and reduce slots allocated to the job with its completion time goal and job characteristics. Below we provide a description of this function.

Given placement matrices P^M and P^R, we can define the number of map and reduce slots allocated to a job j as $s^j_{alloc} = \sum_{tt \in \mathcal{TT}} P^M_{j,tt}$ and $r^j_{alloc} = \sum_{tt \in \mathcal{TT}} P^R_{j,tt}$ correspondingly.

Based on these parameters and the previous definitions of s^j_{pend} and r^j_{pend}, we define the utility of a job j given a placement P as:

$$u_j(P) = u^M_j(P^M) + u^R_j(P^R), \quad \text{where } P = P^M + P^R \tag{1}$$

where u_j^M is a utility function that denotes increasing satisfaction of a job given a placement of map tasks, and u_j^R is a utility function that shows satisfaction of a job given a placement of reduce tasks. The definition of both functions is:

$$u_j^M(P^M) = \begin{cases} \frac{s^j_{alloc} - s^j_{req}}{s^j_{pend} - s^j_{req}} & s^j_{alloc} \geq s^j_{req} \\ \frac{log(s^j_{alloc})}{log(s^j_{req})} - 1 & s^j_{alloc} < s^j_{req} \end{cases} \tag{2}$$

$$u_j^R(P^R) = \frac{log(r^j_{alloc})}{log(r^j_{pend})} - 1 \tag{3}$$

Notice that in practice a job will never get more tasks allocated to it than it has remaining: to reflect this in theory we cap the utility at $u_j(P) = 1$ for those cases.

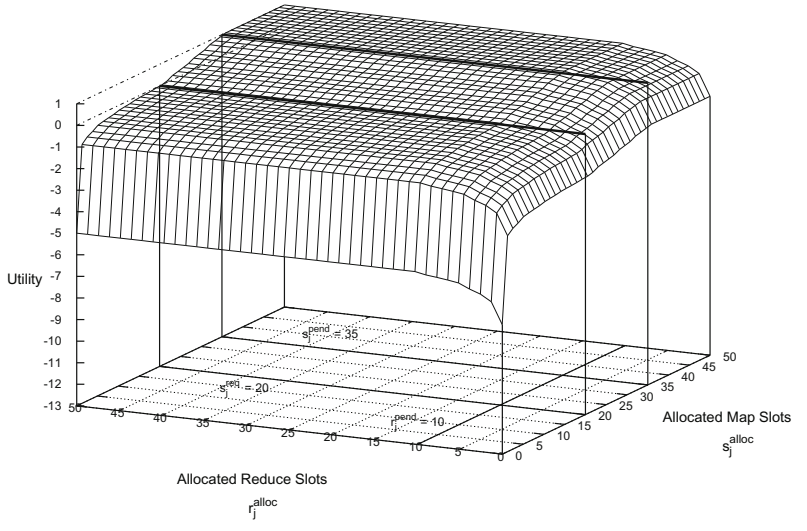


Fig. 2. Shape of the Utility Function when $s^j_{req} = 20$, $s^j_{pend} = 35$, and $r^j_{pend} = 10$

The definition of u differentiates between two cases: (1) the satisfaction of the job grows logarithmically from $-\infty$ to 0 if the job has fewer map slots allocated to it than it requires to meet its completion time goal; and (2) the function grows linearly between 0 and 1, when $s^j_{alloc} = s^j_{pend}$ and thus all pending map tasks for this job are allocated a slot in the current control cycle. Notice that u_j^M is a monotonically increasing utility function, with values in the range $(-\infty, 1]$. The intuition behind this function is that a job is unsatisfied ($u_j^M < 0$) when the number of slots allocated to map tasks is less than the minimum number required to meet the completion time goal of the job. Furthermore, the logarithmic shape of the function stresses the fact that it is critical for a job to make progress and therefore at least one slot must be allocated. A job is no longer unsatisfied

($u_j^M = 0$) when the allocation equals the requirement ($s^j_{alloc} = s^j_{req}$), and its satisfaction is positive ($u_j^M > 0$) and grows linearly when it gets more slots allocated than required. The maximum satisfaction occurs when all the pending tasks are allocated within the current control cycle ($s^j_{alloc} = s^j_{pend}$). The intuition behind u_j^R is that reduce tasks should start at the earliest possible time, so the shuffle sub-phase of the job (reducers pulling data produced by map tasks) can be fully pipelined with execution of map tasks. The logarithmic shape of this function indicates that any placement that does not run all reducers for a running job is unsatisfactory. The range of this function is $[-1, 0]$ and, therefore, it is used to subtract satisfaction of a job that, independently of the placement of map tasks, has unsatisfied demand for reduce tasks. If all the reduce tasks for a job are allocated, this function gets value 0 and thus, $u_j(P) = u_j^M(P^M)$.

Figure 2 shows the generic shape of the utility function for a job that requires at least 20 map tasks to be allocated concurrently ($s^j_{req} = 20$) to meet its completion time goal, has 35 map tasks ($s^j_{pend} = 35$) pending to be executed, and has been configured to run 10 reduce tasks ($r^j_{pend} = 10$), none of which have been started yet. On the X axis, a variable number of allocated slots for reduce tasks (r^j_{alloc}) is shown. On the Y axis, a variable number of allocated slots for map tasks (s^j_{alloc}) is shown. Finally, the Z axis shows the resulting utility value.

3.4 Placement Algorithm and Optimization Objective

Given an application placement matrix P, a utility value can be calculated for each job in the system. The performance of the system can then be measured as an ordered vector of job utility values, U. The objective of RAS is to find a new placement P of jobs on TaskTrackers that maximizes the global objective of the system, $U(P)$, which is expressed as follows:

$$\max \quad \min_j u_j(P) \tag{4}$$

$$\min \quad \Omega_{tt,r} - \sum_{tt}(\sum_j P_{j,tt}) * \Gamma_{j,r} \tag{5}$$

such that

$$\forall_{tt}\forall_r \quad (\sum_j P_{j,tt}) * \Gamma_{j,r} \leq \Omega_{tt,r} \tag{6}$$

This optimization problem is a variant of the Class Constrained Multiple-Knapsack Problem. Since this problem is NP-hard, the scheduler adopts a heuristic inspired by [10], and which is outlined in Algorithm 1. The proposed algorithm consists of two major steps: placing reduce tasks and placing map tasks.

Algorithm 1. Placement Algorithm run at each Control Cycle

Inputs $P^M(job,tt)$: Placement Matrix of Map tasks, $P^R(job,tt)$: Placement Matrix of Reduce tasks, J: List of Jobs in the System, D: Resource demand profile for each job, TT: List of TaskTrackers in the System
Γ_j and Ω_{tt}: Resource demand and capacity for each Job each TaskTracker correspondingly, as used by the auxiliary function $room\_for\_new\_job\_slot$

{——————— **Place Reducers** ———————}

for job in J **do**
  Sort TT in increasing order of overall number of reduce tasks placed (first criteria), and increasing order of number of reducers job placed (second criteria)
  for tt in TT **do**
    if $room\_for\_new\_job\_slot(job, tt)$ & $r^{job}_{pend} > 0$ **then**
      $P^R(job, tt) = P^R(job, tt) + 1$
    end if
  end for
end for

{——————— **Place Mappers** ———————}

for $round = 1 \ldots rounds$ **do**
  for tt in TT **do**
    $job_{in} \leftarrow min\ U(job_{in}, P)$, $room\_for\_new\_job\_slot(job_{in}, tt)$,
    $job_{out} \leftarrow max\ U(job_{out}, P)$, $P^M(job_{out}, tt) > 0$
    repeat
      $P_{old} \leftarrow P$
      $job_{out} \leftarrow max\ U(job_{out}, P)$, $P(job_{out}, tt) > 0$
      $P^M(job_{out}, tt) = P^M(job_{out}, tt) - 1$
      $job_{in} \leftarrow min\ U(job_{in}, P)$, $room\_for\_new\_job\_slot(job_{in}, tt)$
    until $U(job_{out}, P) < U(job_{in}, P_{old})$
    $P \leftarrow P_{old}$
    repeat
      $job_{in} \leftarrow min\ U(job_{in}, P)$, $room\_for\_new\_job\_slot(job_{in}, tt)$
      $P^M(job_{in}, tt) = P^M(job_{in}, tt) + 1$
    until $\nexists job$ such that $room\_for\_new\_job\_slot(job, tt)$
  end for
end for
if map phase of a job is about to complete in this control cycle **then**
  switch profile of placed reducers from shuffle to reduce and wait for Task Scheduler to drive the transition.
end if

Reduce tasks are placed first to allow them to be evenly distributed across TaskTrackers. By doing this we allow reduce tasks to better multiplex network resources when pulling intermediate data and also enable better storage usage. The placement algorithm distributes reduce tasks evenly across TaskTrackers while avoiding collocating any two reduce tasks. If this is not feasible—due to the total number of tasks—it then gives preference to avoiding collocating reduce tasks from the same job. Recall that in contrast to other existing schedulers, RAS dynamically adjusts the number of map and reduce tasks allocated per TaskTracker while respecting its resource constraints. Notice also that when

reduce tasks are placed first, they start running in shuffle phase, so that their demand of resources is directly proportional to the number of map tasks placed for the same job. Therefore, in the absence of map tasks for the same job, a reduce task in shuffle phase only consumes memory. It therefore follows that the system is unlikely to be fully booked by reduce tasks[1].

The second step is placing map tasks. This stage of the algorithm is utility-driven and seeks to produce a placement matrix that balances satisfaction across jobs while treating all jobs fairly. This is achieved by maximizing the lowest utility value in the system. This part of the algorithm executes a series of rounds, each of which tries to improve the lowest utility of the system. In each round, the algorithm removes allocated tasks from jobs with the highest utility, and allocates more tasks to the jobs with the lowest utility. For the sake of fairness, a task gets de-allocated only if the utility of its corresponding job remains higher than the lowest utility of any other job in the system. This results in increasing the lowest utility value across jobs in every round. The loop stops after a maximum number of rounds has reached, or until the system utility no longer improves. This process allows for satisfying the optimization objective introduced in Equation 4.

Recall that RAS is resource-aware and hence all decisions to remove and place tasks are made considering the resource constraints and demands in the system. Furthermore, in order to improve system utilization it greedily places as many tasks as resources allow. This management technique is novel and allows for satisfying the optimization objective introduced in Equation 5.

The final step of the algorithm is to identify if any running jobs will complete their map phase during the current control cycle. This transition is important because it implies that reduce tasks for those jobs will start the reduce phase. Therefore, the algorithm has to switch the resource demand profile for the reduce tasks from 'shuffle' to 'reduce'. Notice that this change could overload some TaskTrackers in the event that the 'reduce' phase of the reduce tasks uses more resources than the 'shuffle' phase. RAS handles this by having the Task Scheduler drive the placement transition between control cycles, and provides overload protection to the TaskTrackers.

3.5 Task Scheduler

The Task Scheduler drives transitions between placements while ensuring that the actual demand of resources for the set of tasks running in a TaskTracker does not exceed its capacity. The placement algorithm generates new placements, but these are not immediately enforced as they may overload the system due to tasks still running from the previous control cycle. The Task Scheduler component takes care of transitioning without overloading any TaskTrackers in the system by picking jobs to assign to the TaskTracker that do not exceed its current capacity, sorted by lowest utility first. For instance, a TaskTracker

[1] We present our thoughts on how to handle the pathological case wherein the number of reduce tasks is so large that there is not enough memory for deploying more tasks in Section 6.

that is running 2 map tasks of job A may have a different assignment for the next cycle, say, 4 map tasks of job B. Instead of starting the new tasks right away while the previous ones are still running, new tasks will only start running as previous tasks complete and enough resources are freed. Recall that the scheduler is adaptive as it continuously monitors the progress of jobs and their average task length, so that any divergence between the placement matrix produced by the algorithm and the actual placement of tasks enforced by the Task Scheduler component is noticed and considered in the following control cycle. The Task Scheduler component is responsible for enforcing the optimization objective shown in Equation 6.

3.6 Job Profiles

The proposed job scheduling technique relies on the use of job profiles containing information about the resource consumption for each job. Profiling is one technique that has been successfully used in the past for MapReduce clusters. Its suitability in these clusters stems from the fact that in most production environments jobs are ran periodically on data corresponding to different time windows [4]. Hence, profiles remains fairly stable across runs [8].

Our profiling technique works offline. To build a job profile we run a job in a sandbox environment with the same characteristics of the production environment. We run the job in isolation multiple times in the sandbox using different configurations for the number of map task slots per node (1 map, 2 maps, ..., up to N). The number of reduce tasks is set to the desired level by the user submitting the job. In the case of multiple reduce tasks, they execute on different nodes.

From the multiple configurations, we select that one in which the job completed fastest, and use that execution to build our profile. We monitor CPU, I/O and memory usage in each node for this configuration using `vmstat`. The reasoning behind this choice is that we want to monitor the execution of a configuration in which competition for resources occurs and some system bottlenecks are hit, but in which severe performance degradation is not yet observed.

Note that obtaining CPU and memory is straight forward for the various phases. For example, if the bottleneck is CPU (that is to say, the node experiences 100% CPU utilization) and there are 4 map tasks running, each map task consumes 25% CPU. Profiling I/O in the shuffle phase is less trivial. Each reduce task has a set of threads responsible for pulling map outputs (intermediate data generated by the map tasks): the number of these threads is a configurable parameter in Hadoop (hereafter `parallelCopies`). These threads are informed about the availability and location of a new map output whenever a map task completes. Consequently, independent of the number of map outputs available, the reduce tasks will never fetch more than `parallelCopies` map outputs concurrently. During profiling we ensure that there are at least `parallelCopies` map outputs available for retrieval and we measure the I/O utilization in the reduce task while shuffling. It can therefore be seen that our disk I/O measurement is effectively an upper bound on the I/O utilization of the shuffle phase.

In RAS we consider jobs that run periodically on data with uniform characteristics but different sizes. Since the map phase processes a single input split of fixed size and the shuffle phase retrieves `parallelCopies` map outputs concurrently (independently of the input data size) their resource profile remain similar. Following these observations, the completion time of the map tasks remains the same while the completion time of the shuffle phase may vary depending on the progress rate of the map phase. The case of the reduce phase is more complicated. The reducer phase processes all the intermediate data at once and this one tends to increase (for most jobs we know of) as the input data size increases. In most of the jobs that we consider we observe that the completion time of the reduce phase scales linearly. However, this is not always the case. Indeed, if the job has no reduce function and simply relies on the shuffle phase to sort, we observe that the completion time scales super-linearly ($n.log(n)$). Having said that, our approach can be improved, for example by using historical information.

Profile accuracy plays a role in the performance of RAS. Inaccurate profiles lead to resource under- or overcommitment. This dependency exists in a slot-based system too, as it also requires some form of profiling to determine the optimal number of slots. The optimal slot number measures a job-specific capacity of a physical node determined by a bottleneck resource for the job, and it can be easily converted into an approximate resource profile for the job (by dividing bottleneck resource capacity by the slot number). Provided with these profiles, RAS allows jobs with different optimal slot numbers to be co-scheduled, which is a clear improvement over classical slot-based systems. The profiling technique used in this paper allows multi-resource profiles to be built, which helps improve utilization when scheduling jobs with different resource bottlenecks. Since the sandbox-based method of profiling assumes that resource utilization remains stable among different runs of the same job on different data, it may fail to identify a correct profile for jobs that do not meet this criterion. For those jobs, an online profiler or a hybrid solutions with reinforcement learning may be more appropriate since RAS is able to work with profiles that change dynamically and allows different profiling technologies to be used for different jobs. While we are not addressing them in this paper, such techniques have been studied in [11–13].

4 Evaluation

In this section we include results from two experiments that explore the two objectives of RAS: improving resource utilization in the cluster (Experiment 1) and meeting jobs' completion time goals (Experiment 2). In Experiment 1, we consider resource utilization only, and compare RAS with a state-of-the-art non-resource-aware Hadoop scheduler. In order to gain insight on how RAS improves resource utilization, we set a relaxed completion time goal with each job. This allow us to isolate both objectives and reduce the effect of completion time goals in the algorithm. In Experiment 2, we consider completion time goals on the same workload. Thus effectively evaluating all capabilities of RAS.

4.1 Experimental Environment and Workload

We perform all our experiments on a Hadoop cluster consisting of 22 2-way 64-bit 2.8GHz Intel Xeon machines. Each machine has 2GB of RAM and runs a 2.6.17 Linux kernel. All machines in the cluster are connected via a Gigabit Ethernet network. The version of Hadoop used is 0.23. The cluster is configured such that one machine runs the JobTracker, another machine hosts the NameNode, and the remaining 20 machines each host a DataNode and a TaskTracker.

Table 1. Workload: 3 Applications, 3 Job instances each (Big, Medium, and Small)

	Sort	**Combine**	**Select**
Instance Label	J1/J8/J9	J2/J6/J7	J3/J4/J5
Input size	90GB/19GB/6GB	5GB/13GB/50GB	10GB/25GB/5GB
Submission time	0s/2,500s/3,750s	100s/600s/1,100s	200s/350s/500s
Length in isolation	2,500s/350s/250s	500s/750s/2,500s	400s/280s/50s
	Experiment 1		
Completion time	3,113s/3,670s/4,100s	648s/3,406s/4,536s	1,252s/608s/623s
	Experiment 2		
Completion time	3,018s/3,365s/4,141s	896s/2,589s/4,614s	802s/550s/560s
Completion time goal	3,000s/3,400s/4,250s	850s/2,600s/6,000s	1,250s/1,100s/950s

To evaluate RAS we consider a representative set of applications included in the Gridmix benchmark, which is part of the Hadoop distribution. These applications are Sort, Combine and Select. For each application we submit 3 different instances with different input sizes, for a total of 9 jobs in each experiment. A summary of the workload can be seen in Table 1, including the label used for each instance in the experiments, the size of its associated input data set, the submission time, and the time taken by each job to complete if the entire experimental cluster is dedicated to it. Additionally, we include the actual completion times observed for each instance in Experiment 1 and 2. Finally, for Experiment 2, we include also the completion time goal associated to each instance.

The resource consumption profiles provided to the scheduler are shown in Table 2. They were obtained following the description provided in Section 3.6. The values are the percentage of each TaskTracker's capacity that is used by a single execution of the sub-phase in question.

Table 2. Job profiles (shuffle: consumed I/O per map placed, upper bound set by `parallelCopies`, the number of threads that pull map output data)

	Sort	**Combine**	**Select**
	Map/Shuffle/Reduce	Map/Shuffle/Reduce	Map/Shuffle/Reduce
CPU	30%/-/20%	25%/-/10%	15%/-/10%
I/O	45%/0.15%/50%	10%/0.015%/10%	20%/0.015%/10%
Memory	25%/-/60%	10%/-/25%	10%/-/25%

4.2 Experiment 1: Execution with Relaxed Completion Time Goals

The goal of this experiment is to evaluate how RAS improves resource utilization compared to the Fair Scheduler when completion time goals are so relaxed that the main optimization objective of the algorithm is to maximize resource utilization (see Equation 5). To this end we associate with each job instance an highly relaxed completion time goal. We run the same workload using both the Fair Scheduler and RAS and compare different aspects of the results.

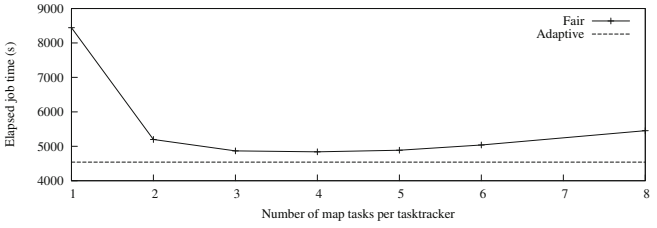
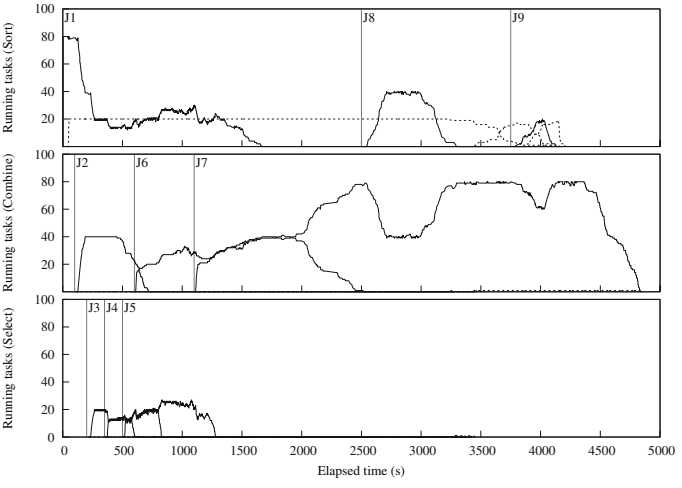


Fig. 3. Experiment 1: Workload makespan with different Fair Scheduler configurations (Y-axis starts at 4000s)

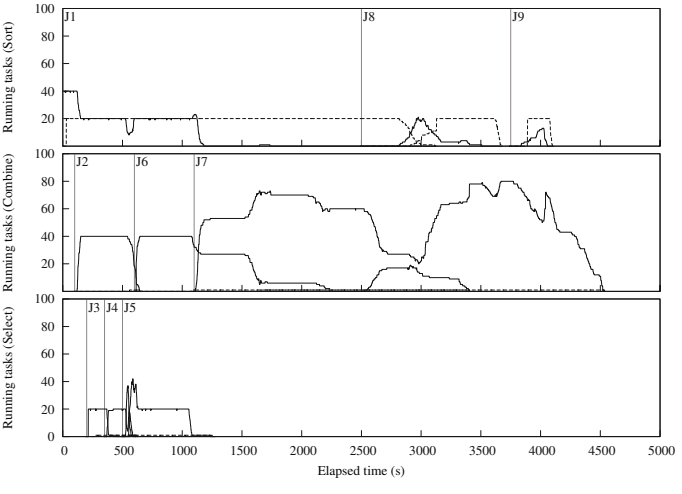
Dynamic task concurrency level per TaskTracker. Our first objective in this experiment is to study how the dynamic management of the level of task concurrency per-TaskTracker improves workload performance. To this end, we run the same workload using the Fair Scheduler with different concurrency level configurations: specifically, we vary the maximum number of map slots per TaskTracker from 1 to 8, and compare the results with the execution using RAS. Results are shown in Figure 3. As can be seen, the best static configuration uses 4 concurrent map tasks per TaskTracker (80 concurrent tasks across 20 TaskTrackers). Configurations that result in low and high concurrency produce worse makespan due to resources being underutilized and overcommitted, respectively.

We observe that RAS outperforms the Fair Scheduler for all configurations, showing an improvement that varies between 5% and 100%. Our traces show that the average task concurrency level in RAS was 3.4 tasks per TaskTracker. Recall that the best static configuration of per-TaskTracker task concurrency depends on the workload characteristics. As workloads change over time in real systems, even higher differences between static and dynamic management would be observed. RAS overcomes this problem by dynamically adapting task concurrency level based on to the resource usage in the system.

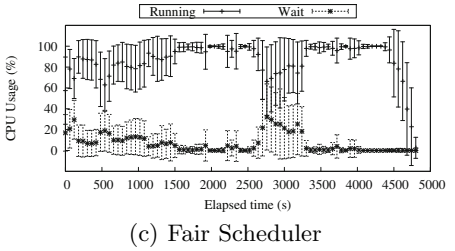
Resource allocation and Resource utilization. Now we look in more detail at the execution of the workload using RAS as compared to the Fair Scheduler running a static concurrency level of 4. Figures 4(a) and 4(b) show the task assignment resulting from both schedulers. For the sake of clarity, we group jobs corresponding to the same application (Sort, Combine or Select) into different rows. Each row contains solid and dotted lines, representing the number of running map and reduce tasks respectively. The submission time for each job is shown by a (labeled) vertical line, following the convention presented in Table 1.



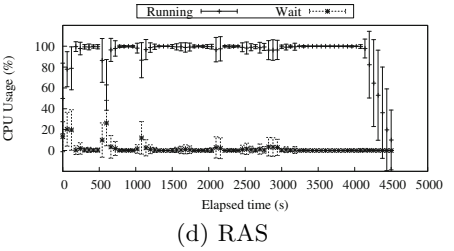
(a) Fair Scheduler



(b) RAS



(c) Fair Scheduler



(d) RAS

Fig. 4. Experiment 1: Workload execution and CPU utilization: (a) and (c) correspond to Fair Scheduler using 4 slots per TaskTracker; (b) and (d) correspond to RAS using a variable number slots per TaskTracker

Combine and Select are configured to run one single reduce task per job since there is no benefit from running them with more reduce tasks on our testing environment; the dotted line representing the reduce is at the bottom of the chart. As it can be observed, RAS does not allocate more concurrent map slots than the Fair Scheduler during most of the execution. Moreover, the sum of reduce and map tasks remains lower than the sum of reduce and map tasks allocated by the Fair Scheduler except for a small time interval ($\sim$ 100s) immediately after the submission of Job 6 (J6). RAS is able to improve the makespan while maintaining a lower level of concurrency because it does better at utilizing resources which ultimately results in better job performance. To get a better insight on how RAS utilizes resources as compared to the Fair Scheduler we plot the CPU utilization for both schedulers in Figures 4(c) and 4(d). These figures show the percentage of CPU time that TaskTrackers spent running tasks (either in system or user space), and the time that the CPU was waiting. For each metric we show the mean value for the cluster, and the standard deviation across TaskTrackers. Wait time represents the time that the CPU remains idle because all threads in the system are either idle or waiting for I/O operations to complete. Therefore, it is a measure of resource wastage, as the CPU remains inactive. While wait time is impossible to avoid entirely, it can be reduced by improving the overlapping of tasks that stress different resources in the TaskTracker. It is noticeable that in the case of the Fair Scheduler the CPU spends more time waiting for I/O operations to complete than RAS. Further, modifying the number of concurrent slots used by the Fair Scheduler does not improve this result. The reason behind this observation is key to our work: other schedulers do not consider the resource consumption of applications when making task assignment decisions, and therefore are not able to achieve good overlap between I/O and CPU activity.

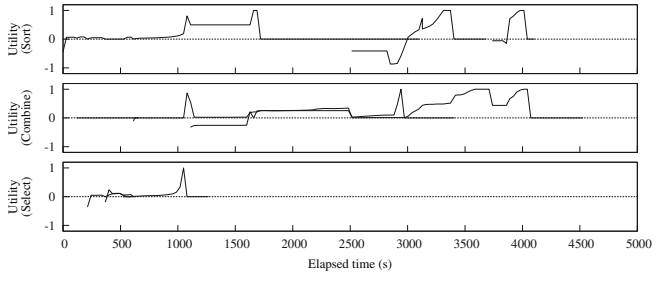


Fig. 5. Experiment 1: Job Utility

Utility guidance. Finally, to illustrate the role of the utility function in RAS, Figure 5 shows the utility value associated with each job during the execution of the workload. Since the jobs have extremely lax completion time goals, they are assigned a utility value above 0 immediately after one task for each job is placed. As can be seen, the allocation algorithm balances utility values across jobs for most of the execution time. In some cases, though, a job may get higher utility than the others: this is explained by the fact that as jobs get closer to

completion, the same resource allocation results in higher utility. This is seen in our experiments: for all jobs, the utility increases until all their remaining tasks are placed. In this experiment we can also see that Job 7 has a very low utility right after it is launched (1,100s) in contrast with the relatively high utility of Job 1, even though most resources are actually assigned to Job 7. This is because while Job 1 has very few remaining tasks, no tasks from Job 7 have been completed and thus its resource demand estimation is not yet accurate. This state persists until approximately time 1,650s).

4.3 Experiment 2: Execution with Tight Completion Time Goals

In this experiment we evaluate the behavior of RAS when the applications have stringent completion time goals. To do this we associate a tight completion time goal with the workload described for our previous experiment.

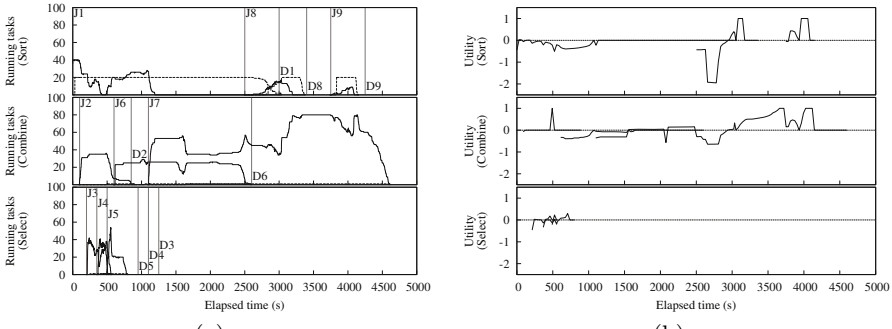


Fig. 6. Experiment 2: Workload execution and Job utility

Figure 6(a) shows the number of concurrent tasks allocated to each job during the experiment. We use vertical lines and labels to indicate submission times (labeled J1 to J9) and completion time goals (labeled D1 to D9) for each of the nine jobs in the workload. To illustrate how RAS manages the tradeoff between meeting completion time goals and maximizing resource utilization, we look at the particular case of Job 1 (Sort), Job 7 (Combine) and Job 8 (Sort), submitted at times J1 (0s), J7 (1,100s) and J8 (2,500s) respectively. In Experiment 1 their actual completion times were 3,113s, 4,536s and 3,670s, while in Experiment 2 they completed at times 3,018s, 4,614s and 3,365s respectively. Because their completion time goals in Experiment 2 are 3,000s, 6,000s and 3,400s (a factor of 1.2X, 1.9X and 2.5X compared to their length observed in isolation), the algorithm allocates more tasks to Job 1 and Job 8 at the expense of Job 7, which sees its actual completion time delayed with respect to Experiment 1 but still makes its more relaxed goal. It is important to remark again that completion time goals in our scheduler are soft deadlines used to guide the workload management as opposed to strict deadlines in which missing a deadline is associated with

strong penalties. Finally, notice that Job 1 and Job 8 would have clearly missed their goals in Experiment 1: here, however, RAS adaptively moves away from the optimal placement in terms of resource allocation to adjust the actual completion times of jobs. Recall that RAS is still able to leverage a resource model while aiming at meeting deadlines, and still outperforms the best configuration of Fair Scheduler by 167 seconds, 4,781s compared to 4,614s.

To illustrate how utility is driving placement decisions, we include Figure 6(b), which shows the utility of the jobs during the workload execution and gives a better intuition of how the utility function drives the scheduling decisions. When a job is not expected to reach its completion time goal with the current placement, its utility value goes negative. For instance, starting from time 2,500s when J8 is launched and the job still has very few running tasks, the algorithm places new tasks to J8 at the expense of J7. However, as soon as J8 is running the right amount of tasks to reach the deadline, around time 3,000s, both jobs are balanced again and the algorithm assigns more tasks to J7.

5 Related Work

Much work have been done in the space of scheduling for MapReduce. Since the number of slots in a Hadoop cluster is fixed through out the lifetime of the cluster, most of the proposed solutions can be reduced to a variant of the *task-assignment* or *slot-assignment* problem. The Capacity Scheduler [14] is a pluggable scheduler developed by Yahoo! which partition resources into pools and provides priorities for each pool. Hadoop's Fair Scheduler [9] allocates equal shares to each tenant in the cluster. Quincy scheduler [15] proposed for the Dryad environment [16] also shares similar fairness goals. All these schedulers are built on top of the slot model and do not support user-level goals.

The performance of MapReduce jobs has attracted much interest in the Hadoop community. Stragglers, tasks that take an unusually long time to complete, have been shown to be the most common reason why the total time to execute a job increases [2]. Speculative scheduling has been widely adopted to counteract the impact of stragglers [2, 9]. Under this scheduling strategy, when the scheduler detects that a task is taking longer than expected it spawns multiple instances of the task and takes the results of the first completed instance, killing the others [9]. In Mantri [5] the effect of stragglers is mitigated via the 'kill and restart' of tasks which have been identified as potential stragglers. The main disadvantage of these techniques is that killing and duplicating tasks results in wasted resources [5, 9]. In RAS we take a more proactive approach, in that we prevent stragglers resulting from resource contention. Furthermore, stragglers caused by skewed data cannot be avoided at run-time [5] by any existing technique. In RAS the slow-down effect that these stragglers have on the end-to-end completion time of their corresponding jobs is mitigated by allocating more resources to the job so that it can still complete in a timely manner.

Recently, there has been increasing interest in user-centric data analytics. One of the seminal works in this space is [6]. In this work, the authors propose a scheduling scheme that enables soft-deadline support for MapReduce jobs. It

differs from RAS in that it does not take into consideration the resources in the system. Flex [7] is a scheduler proposed as an add-on to the Fair Scheduler to provide Service-Level-Agreement (SLA) guarantees. More recently, ARIA [8] introduces a novel resource management framework that consists of a job profiler, a model for MapReduce jobs and a SLO-scheduler based on the Earliest Deadline First scheduling strategy. Flex and Aria are both slot-based and therefore suffers from the same limitations we mentioned earlier. One of the first works in considering resource awareness in MapReduce clusters is [17]. In this work the scheduler classifies tasks into good and bad tasks depending on the load they impose in the worker machines. More recently, the Hadoop community has also recognized the importance of developing a resource-aware scheduling for MapReduce. [18] outlines the vision behind the Hadoop scheduler of the future. The framework proposed introduces a resource model consisting of a 'resource container' which is—like our 'job slot'—fungible across job tasks. We think that our proposed resource management techniques can be leveraged within this framework to enable better resource management.

6 Conclusions

In this paper we have presented the Resource-aware Adaptive Scheduler, RAS, which introduces a novel resource management and job scheduling scheme for MapReduce. RAS is capable of improving resource utilization and job performance. The cornerstone of our scheduler is a resource model] based on a new resource abstraction, namely 'job slot'. This model allows for the formulation of a placement problem which RAS solves by means of a utility-driven algorithm. This algorithm in turn provides our scheduler with the adaptability needed to respond to changing conditions in resource demand and availability.

The presented scheduler relies on existing profiling information based on previous executions of jobs to make scheduling and placement decisions. Profiling of MapReduce jobs that run periodically on data with similar characteristics is an easy task, which has been used by many others in the community in the past. RAS pioneers a novel technique for scheduling reduce tasks by incorporating them into the utility function driving the scheduling algorithm. It works in most circumstances, while in some others it may need to rely on preempting reduce tasks (not implemented in the current prototype) to release resources for jobs with higher priority. Managing reduce tasks in this way is not possible due to limitations in Hadoop and hence it affects all existing schedulers. In RAS we consider three resource capacities: CPU, memory and I/O. It can be extended easily to incorporate network infrastructure bandwidth and storage capacity of the TaskTrackers. Nevertheless, network bottlenecks resulting from poor placement of reduce tasks [5] can not be addressed by RAS without additional monitoring and prediction capabilities.

Our experiments, in a real cluster driven by representative MapReduce workloads, demonstrate the effectiveness of our proposal. To the best of our knowledge

RAS is the first scheduling framework to use a new resource model in MapReduce and leverage resource information to improve the utilization of resources in the system and meet completion time goals on behalf of users.

Acknowledgements. This work is partially supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by IBM through the 2010 IBM Faculty Award program, and by the HiPEAC European Network of Excellence (IST-004408).

References

1. Adaptive Scheduler, https://issues.apache.org/jira/browse/MAPREDUCE-1380
2. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI 2004, San Francisco, CA, pp. 137–150 (December 2004)
3. Hadoop MapReduce, http://hadoop.apache.org/mapreduce/
4. Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R., Liu, H.: Data warehousing and analytics infrastructure at facebook. In: Proceedings of the 2010 International Conference on Management of Data, SIGMOD 2010, pp. 1013–1020. ACM, New York (2010)
5. Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., Harris, E.: Reining in the outliers in map-reduce clusters using mantri. In: OSDI 2010, pp. 1–16. USENIX Asoc., Berkeley (2010)
6. Polo, J., Carrera, D., Becerra, Y., Steinder, M., Whalley, I.: Performance-driven task co-scheduling for MapReduce environments. In: Network Operations and Management Symposium, NOMS, pp. 373–380. IEEE, Osaka (2010)
7. Wolf, J., Rajan, D., Hildrum, K., Khandekar, R., Kumar, V., Parekh, S., Wu, K.-L., Balmin, A.: Flex: A Slot Allocation Scheduling Optimizer for Mapreduce Workloads. In: Gupta, I., Mascolo, C. (eds.) Middleware 2010. LNCS, vol. 6452, pp. 1–20. Springer, Heidelberg (2010)
8. Verma, A., Cherkasova, L., Campbell, R.H.: ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. In: 8th IEEE International Conference on Autonomic Computing, Karlsruhe, Germany (June 2011)
9. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving mapreduce performance in heterogeneous environments. In: OSDI 2008, pp. 29–42. USENIX Association, Berkeley (2008)
10. Tang, C., Steinder, M., Spreitzer, M., Pacifici, G.: A scalable application placement controller for enterprise data centers. In: Procs. of the 16th Intl. Conference on World Wide Web, pp. 331–340. ACM, NY (2007)
11. Herodotou, H., Babu, S.: Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. In: VLDB (2010)
12. Pacifici, G., Segmuller, W., Spreitzer, M., Tantawi, A.N.: Dynamic estimation of cpu demand of web traffic. In: Lenzini, L., Cruz, R.L. (eds.) VALUETOOLS. ACM International Conference Proceeding Series, vol. 180, p. 26. ACM (2006)
13. Tesauro, G., Jong, N.K., Das, R., Bennani, M.N.: A hybrid reinforcement learning approach to autonomic resource allocation. In: Proceedings of the 2006 IEEE International Conference on Autonomic Computing, pp. 65–73. IEEE Computer Society, Washington, DC (2006)
14. Yahoo! Inc. Capacity scheduler, http://developer.yahoo.com/blogs/hadoop/posts/2011/02/capacity-scheduler/

15. Isard, M., Prabhakaran, V., Jon Currey, U.W., Talwar, K., Goldberg, A.: Quincy: fair scheduling for distributed computing clusters. In: SOSP 2009 (2009)
16. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, EuroSys 2007, pp. 59–72. ACM, New York (2007)
17. Dhok, J., Varma, V.: Using pattern classification for task assignment, http://researchweb.iiit.ac.in/~jaideep/jd-thesis.pdf
18. Murthy, A.: Next Generation Hadoop, http://developer.yahoo.com/blogs/hadoop/posts/2011/03/mapreduce-nextgen-scheduler/

A Content-Based Publish/Subscribe Matching Algorithm for 2D Spatial Objects

Athanasios Konstantinidis[1], Antonio Carzaniga[2], and Alexander L. Wolf[1]

[1] Department of Computing, Imperial College London, UK
[2] Faculty of Informatics, University of Lugano, CH

Abstract. An important concern in the design of a publish/subscribe system is its expressiveness, which is the ability to represent various types of information in publications and to precisely select information of interest through subscriptions. We present an enhancement to existing content-based publish/subscribe systems with support for a 2D spatial data type and eight associated relational operators, including those to reveal overlap, containment, touching, and disjointedness between regions of irregular shape. We describe an algorithm for evaluating spatial relations that is founded on a new dynamic discretization method and region-intersection model. In order to make the data type practical for large-scale applications, we provide an indexing structure for accessing spatial constraints and develop a simplification method for eliminating redundant constraints. Finally, we present the results of experiments evaluating the effectiveness and scalability of our approach.

1 Introduction

The data models of existing content-based publish/subscribe systems embody only the most basic and primitive types: numbers, strings, and dates. This is true, for example, of the Java Message Service specification (JMS), which is restricted to the primitive data types of Java.[1] In this paper we present an extended data model supporting 2D spatial objects. The representation of 2D objects is essential in many problem domains, but perhaps its primary use is in the representation and processing of geographical information. This type of information has concrete applications in agriculture, transportation, logistics, infrastructure management, and more recently various personal applications.

The simplest approach to incorporating 2D spatial objects into a content-based publish/subscribe system would be to establish some sort of convention for indicating regions in an x/y coordinate space, such as by giving the locations of opposite corners of rectangular regions or by giving the center point of a circular region of some radius. The coordinate space itself might be abstract or it might be associated with a standard reference model such as longitude and

[1] JMS defines an SQL-like selection feature for its content-based subscriptions. However, this selection feature is practically equivalent to a set (i.e., a disjunction) of subscriptions based on name-operator-value constraints.

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 208–227, 2011.

latitude. Notice that this simple approach has the benefit of requiring only some conventional use of the primitive data types and relational operators already commonly supported by publish/subscribe systems.

The simplest matching problem would be to test whether a point in a space is contained within a given region. This concept has been explored for so-called *location-based services* [1,9,10,16,29], and used in practice to underpin practical applications such as Facebook Places.[2] It has also been explored for use within the virtual world of games, where game state and actions are bound to a notion of virtual location [22].

Our goal is to support a richer concept of region and region matching. In particular, we seek to support regions having irregular, and therefore more realistic boundary shapes, whereas prior work has focused on simple shapes, such as axis-aligned rectangles [24,27]. Moreover, we seek to support the matching of regions to other regions, not just points to regions, which implies a much richer set of potential matching relations, such as *overlaps*, where two regions share some interior points but not others, *meets*, where two regions touch but do not overlap, and *equals*, where two regions mutually cover each other.

Consider, for example, an event notification emitted by a severe-weather warning system that reports a storm affecting a geographical area A with maximum expected wind speed w and precipitation level p. A facility management company might be interested in receiving such notifications, but only when A overlaps with one of the company's facilities (e.g., a building), and if the wind speed w and precipitation level p are above certain safety thresholds. In this and similar applications, subscriptions would include 2D geographical regions obtained from, say, a geographical information system (GIS) database, while notification messages would contain 2D observation overlays, such as weather systems, pollutant concentrations, group/herd movement, and the like. In order to capture such regions and their relationships within a content-based publish/subscribe system, a new data type for 2D spatial objects is required.

We derive the new data type from a standard model found in geographical information systems in which a 2D region is defined as a the space enclosed within a boundary consisting of line segments. Given two such boundary-delimited regions, the model induces eight binary relations between the regions. The theoretical basis for the evaluation of the eight relations is the 4-intersection model developed in the seminal work of Egenhofer and Franzosa [14] and used in many spatial database management systems. In this model, each binary relation between two regions A and B can be evaluated by testing the emptiness of four intersections between the boundary, interior, and exterior point sets of A and B.

Starting from this abstract model, we develop a concrete representation of regions based on a *discretization* of their boundaries, and use the eight basic binary relations between regions to form the relational operators of the type. We also derive a small set of concrete conditions that lead to an efficient evaluation of the relations. The boundary discretization, and the corresponding evaluation of the spatial relations, are based on a newly formulated variant of the Egenhofer

[2] http://www.facebook.com/places/

and Franzosa 4-intersection model. We were driven to develop this new model because the original requires all interior points of a region to be inspected and conventional discretization methods [17,18] rely on a global grid structure.

The original model and conventional methods were developed for traditional GIS database applications, where the problem is to compute and store all the spatial relations among large numbers of relatively static regions, and then quickly process a spatial query against those stored relations [26]. In that context it is reasonable to incur the considerable computational costs of a global-grid discretization and a full interior point inspection, since they can be performed in an off-line preprocessing step. Publish/subscribe systems, by contrast, face the problem of having to perform an on-line computation to reveal the spatial relations that exist between previously unseen regions contained in high-rate message traffic and large numbers of stored regions representing spatial constraints. This demands the new approach introduced in this paper in which boundary discretization is dynamic and point inspections can be significantly reduced.

The ideas presented here provide the theoretical basis for accommodating 2D spatial objects in the basic matching/filtering/forwarding function of publish/subscribe systems, as well as the routing function of distributed versions of such systems. In this paper we focus on the matching problem, describing a specific algorithm, indexing structure, and logical simplification method for 2D spatial constraints, integrated and evaluated within a general content-based matching algorithm. The indexing structure, which we call the *CR-tree*, is an extension of the well-known R-tree developed by Guttman [20]. Our extension allows the matching algorithm to efficiently evaluate a large set of spatial constraints, as we demonstrate experimentally. We also demonstrate the effectiveness of the spatial-constraint simplification method.

In summary, we make the following contributions: (1) the use of topological relations between 2D spatial objects within content-based publish/subscribe systems; (2) a discretization of a common 2D model that admits to an efficient representation and use of 2D objects in matching; (3) an indexing structure to process large numbers of 2D spatial constraints during the matching process; and (4) a logical simplification method for 2D spatial constraints. We provide background on general spatial modeling in Section 2, and then define a specific spatial model, its discretization, and a matching algorithm in Section 3. The CR-tree index and spatial simplification method are described in Section 4. We present the results of an experimental evaluation in Section 5.

2 Background

To support 2D objects within a content-based publish/subscribe system, we must define spatial concepts and models that would lead to efficient and robust implementations. In particular, we seek a fast matching algorithm capable of evaluating incoming 2D regions (represented as attribute values in messages) against potentially large numbers of 2D spatial constraints (contained in subscriptions). In this section, we lay out such definitions and models. Furthermore,

since we develop the 2D spatial model as an extension of a concrete content-based matching algorithm, namely the Siena Fast Forwarding algorithm [8], we also review the algorithm and its existing indexing structures.

2.1 Spatial Concepts and Spatial Modeling

Space is regarded as being composed of an infinite number of points forming a continuum, the so-called Euclidean model. We represent a region as a polygon. More specifically:

Definition 1. *A* region *is a simple polygon, that is, the portion of the Euclidean space delimited by a closed finite sequence of line segments such that any two adjacent segments share an end point, and no end point belongs to more than two segments. Furthermore, no point other than an end point is shared by two segments (i.e., line segments do not intersect).*

As a concrete example, consider a message used within an environmental management system to warn about leaks of dangerous chemicals or other polluting agents in lakes or oceans. A hypothetical message of this type is shown in Figure 1. The warning indicates the affected area by specifying a 2D region (attribute "area"). The region is defined by the end points of the line segments that form its boundary, given as a sequence (only partially shown) of point pairs.

string warning = "hazardous leak"
region area = $(23.1, 10.9), (30.3, 27.0), (48.0, 19.0), \ldots$
int concentration = 172
int hazard = 4

Fig. 1. A message describing a 2D object as the region "area"

Using the *region connection calculus* introduced by Cohn et al. [12] we can prove the existence of eight *jointly exhaustive* and *pair-wise disjoint* (JEPD) binary topological relations between 2D regions. This means that any pair of regions must have a topology that is characterized by one and only one of the eight relations. The complete set of relations between two regions A and B is illustrated in Figure 2. The first four relations are symmetric; the remaining four should be read as A *relation* B (e.g., A *inside* B). These qualitative topological relations between regions define the constraints that can be expressed in subscriptions that select 2D spatial objects. Since our objective is to identify all matching constraints for a given region, we must develop an algorithmic evaluation of these relations.

In order to implement such constraints in subscription predicates, we need to model the concept of topological relations between regions. Following the analysis of Egenhofer et al. [13,14,15], if $A \subseteq \mathbb{R}^2$ is a region, then there exists a set $U \subseteq \mathbb{R}^2$ such that $U = (A^0 \cup \partial A \cup A^-)$, where A^0, A^-, and ∂A are the interior, exterior, and boundary sets of A, respectively. Notice that they are mutually disjoint

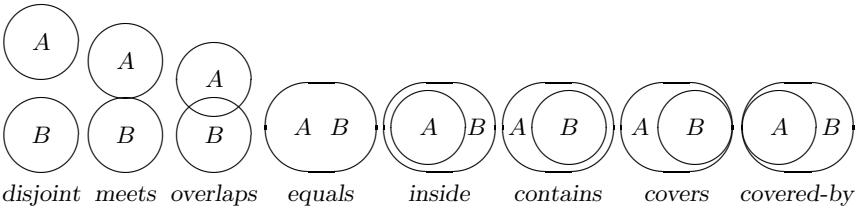


Fig. 2. The eight jointly exhaustive and pair-wise disjoint (JEPD) relations between regions

subsets of U, that is, A^0, ∂A, and A^- form a partition of U. We refer to U as the *universe*. It is then provable that the topological relation between any two regions $A, B \subseteq U$ can be identified by specifying the nine possible intersections between their interiors, exteriors, and boundaries (the 9-intersection model), each of which can be either empty or non-empty. Egenhofer and Franzosa [14] introduced a model based on only four intersections (the 4-intersection model), obtained by removing redundant entries from the 9-intersection model. Below is the list of intersections in the 4-intersection model of Egenhofer and Franzosa.

Intersection 1. Intersection between the boundary of A and the boundary of B denoted by $\partial A \cap \partial B$.
Intersection 2. Intersection between the boundary of A and the interior of B denoted by $\partial A \cap B^0$.
Intersection 3. Intersection between the interior of A and the boundary of B denoted by $A^0 \cap \partial B$.
Intersection 4. Intersection between the interior of A and the interior of B denoted by $A^0 \cap B^0$.

In Section 3 we introduce a discretization of regions, as well as a variation of this 4-intersection model, that are specifically designed to ensure efficient and robust evaluation of the JEPD spatial constraints.

2.2 The Siena Fast Forwarding Algorithm

Siena is a popular distributed publish/subscribe system that uses name-value pairs in published messages and name-operator-value constraints to define subscriptions [7]. Following the terminology used in Siena, a *filter* is a conjunction of constraints (that defines a subscription) and a *predicate* is a disjunction of filters (representing a set of subscriptions).

The matching algorithm developed within the Siena project, called Siena Fast Forwarding (SFF) [8], follows the approach of Yan and Garcia-Molina [30] by using a counting algorithm for predicate matching. In particular, the algorithm builds a *forwarding table* consisting of a global index of all the unique constraints found in a set of predicates, where each predicate is associated with an interface of a content-based broker/router. Given this constraint index and an input message, the matching process amounts to finding the set of interfaces through which

the message must be forwarded. The algorithm proceeds by evaluating each attribute of the message against the index of constraints, counting the numbers of matching constraints per filter. When the count reaches the total number of constraints in a filter associated with a predicate, the algorithm forwards the message to the interface associated with that predicate. As an example, Figure 3 shows the high-level contents of a forwarding table for a broker/router with two interfaces, I_1 and I_2, each associated with a predicate of two filters, $f_{1.1}$ and $f_{1.2}$, and $f_{2.1}$ and $f_{2.2}$, respectively. Notice that the message shown in Figure 1 would match $f_{2.2}$ and, therefore, would be forwarded through interface I_2.

I_1	$f_{1.1}$	*string* stock ="MTK" *int* price < 100
	$f_{1.2}$	*string* stock = "DYS" *int* price > 200 *bool* bubble = true
I_2	$f_{2.1}$	*region* cloud *overlaps* $(10, 5), (7, 12), \ldots$ *int* pressure < 1000
	$f_{2.2}$	*string* warning = "hazardous leak" *region* area *disjoint* $(3, 2), (15, 40), \ldots$ *int* concentration > 100 *int* hazard > 3

Fig. 3. A forwarding table with 2D regions

SFF already implements specific constraint indexes for some basic data types, including integer, float, Boolean, and string, along with their operators. It also provides a way to extend the algorithm with more types and operators, and with type- and operator-specific constraint indexes. We use this extension feature to plug in our implementations for representing 2D regions, a spatial constraint index, and the algorithm for evaluating 2D constraints.

3 Spatial Model

The abstract notion of a 2D object is generally defined by a set of points taken from a continuous space. Therefore, in order to realize a concrete implementation of this abstract model, we must somehow map these continuous 2D objects onto a discrete structure amenable to efficient algorithmic processing. The term *discretization* refers generally to both the mapping of a continuous object onto a discrete set and the algorithmic processing of that discrete set. A conventional approach to the discretization of 2D regions is to construct a discretized universe using a global grid structure [17,18]. However, this method involves the mapping of regions onto grid points, which is a surprisingly complex procedure [19,21] that would introduce unacceptable levels of overhead to a publish/subscribe system.

We follow a completely different approach. The key idea behind this new discretization is to use a point-region check as the primary building block for the

evaluation algorithm. In particular, let POINTCHECK(p, R) be a decision procedure that returns the topological relation between point p and region R as either *in*, *out*, or *meet*, meaning that p is within the interior, exterior, or boundary of R, respectively. This POINTCHECK procedure takes two discrete structures, a point and a finite set of line segments, and can be implemented efficiently using a ray-shooting algorithm from a computational-geometry library [5,23]. We use the procedure to efficiently evaluate the eight JEPD spatial relations through a variant of the 4-intersection model of Egenhofer and Franzosa.

3.1 A New 4-Intersection Model

The 4-intersection model of Egenhofer and Franzosa reduces the identification of the topological relations between two regions A and B to the problem of determining whether each of the four intersections is empty or not. Our general approach to discretizing each intersection decision is to find a "witness" point that would indicate that the intersection is not empty. In particular, the first three of the four intersections can be discretized as follows:

Discretization 1. To decide the emptiness of $\partial A \cap \partial B$, find a point p on the boundary of A that can be tested as a boundary point for B. If such a p is found, then consider the intersection *non-empty*; otherwise *empty*.

Discretization 2. To decide the emptiness of $\partial A \cap B^0$, find a point p on the boundary of A that can be tested as an interior point for B. If such a p is found, then consider the intersection *non-empty*; otherwise *empty*.

Discretization 3. To decide the emptiness of $A^0 \cap \partial B$, find a point p on the boundary of B that can be tested as an interior point for A. If such a p is found, then consider the intersection *non-empty*; otherwise *empty*.

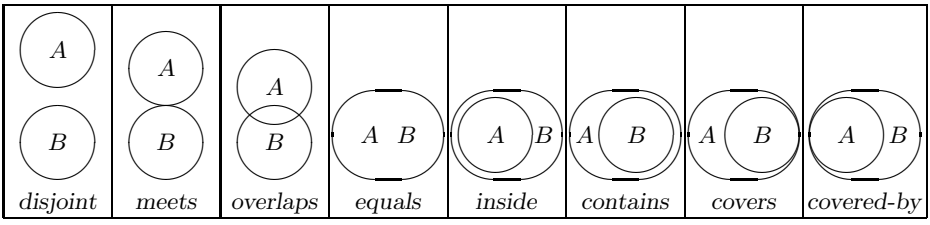
The fourth intersection in the model of Egenhofer and Franzosa is $A^0 \cap B^0$, which involves two interior sets whose discretization is prohibitively complex for achieving fast content-based matching. To overcome this problem we replace the fourth intersection with the intersection between the boundary set of A and the exterior set of B. Thus, we define a new 4-intersection model with the new fourth intersection being $\partial A \cap B^-$, which admits to an efficient discretization similar in form to the previous three:

Discretization 4. To decide the emptiness of $\partial A \cap B^-$, find a point p on the boundary of A that can be tested as an exterior point for B. If such a p is found, then consider the intersection *non-empty*; otherwise *empty*.

Of course, given this change to the intersection model, we must show that all eight topological relations between regions can still be decided unambiguously. We do this in Table 1, which details the precise correspondence between the four intersections and the topological relations.

We can now proceed to implement the evaluation of the topological relations as a concrete algorithm. We emphasize that this algorithm is based on a crucial property of the new 4-intersection model, namely that all its intersections involve

Table 1. The eight JEPD relations as captured by the new 4-intersection model



$(\partial A \cap \partial B, \partial A \cap B^0, A^0 \cap \partial B, \partial A \cap B^-)$

$(\emptyset,\emptyset,\emptyset,1)$	$(1,\emptyset,\emptyset,1)$	$(1,1,1,1)$	$(1,\emptyset,\emptyset,\emptyset)$	$(\emptyset,1,\emptyset,\emptyset)$	$(\emptyset,\emptyset,1,1)$	$(1,\emptyset,1,1)$	$(1,1,\emptyset,\emptyset)$

$\emptyset$: empty, 1: non-empty

at least one boundary set ∂X, and either another boundary set ∂Y, an interior set Y^0, or an exterior set Y^-. Having a boundary set ∂X to start with, the algorithm can proceed first by discretizing ∂X into a finite set of points $p \in \partial X$ and then by checking the relation of each point p with the other region Y (boundary, interior, or exterior) using the POINTCHECK(p, Y) primitive.

3.2 Algorithm

Our method for identifying the topological relation between two regions A and B, each expressed as a sequence of points, is given as Algorithm 1. The algorithm uses four Boolean variables i_1, i_2, i_3, and i_4 that indicate the presence of "witnesses" for the non-emptiness of each of the four intersections in our 4-intersection model. The variables are initialized as false and become true as soon as the algorithm finds one point in the corresponding intersection.

The algorithm can find witness points for the first, second, and fourth intersections (thereby assigning i_1, i_2, and i_4) through a single iteration over the points of the boundary of the first region ∂A (lines 6–18). In this loop the algorithm considers only some points of the boundary ∂A, effectively discretizing that boundary. The selection of points is performed by the DISCRETIZE procedure sketched on lines 46–49 and discussed in detail in Section 3.3, below.

The loop over the points of ∂A may terminate immediately whenever the algorithm finds witnesses for intersections 2 and 4, since they unambiguously identify the *overlaps* relation. At the end of the loop, the algorithm can identify the *equals*, *inside*, and *covered-by* relations. Lines 19–25 implement this decision based on the values of i_1, i_2, and i_4. Notice that some of the conditions are redundant (e.g., when the algorithm reaches line 19, i_2 and i_4 cannot both be true). However, for clarity and ease of verification, we write each condition explicitly and in the same order as the corresponding relation appears in Table 1.

If none of the immediately identifiable relations hold, the algorithm proceeds by checking whether the third intersection is empty or not. The algorithm iterates

Algorithm 1. Topological relation between two regions

```
 1: procedure FINDRELATION(A, B)
 2:     i1 ← FALSE                          ▷ witness: i1 ⇒ ∂A ∩ ∂B ≠ ∅
 3:     i2 ← FALSE                          ▷ witness: i2 ⇒ ∂A ∩ B^0 ≠ ∅
 4:     i3 ← FALSE                          ▷ witness: i3 ⇒ A^0 ∩ ∂B ≠ ∅
 5:     i4 ← FALSE                          ▷ witness: i4 ⇒ ∂A ∩ B^- ≠ ∅
 6:     for each p ∈ DISCRETIZE(∂A) do
 7:         T ← POINTCHECK(p, B);
 8:         if T = meet then
 9:             i1 ← TRUE
10:         else if T = in then
11:             i2 ← TRUE
12:         else if T = out then
13:             i4 ← TRUE
14:         end if
15:         if i2 ∧ i4 then
16:             return overlaps                 ▷ (?, 1, ?, 1)
17:         end if
18:     end for
19:     if i1 ∧ ¬i2 ∧ ¬i4 then
20:         return equals                       ▷ (1, ∅, ?, ∅)
21:     else if ¬i1 ∧ i2 ∧ ¬i4 then
22:         return inside                       ▷ (∅, 1, ?, ∅)
23:     else if i1 ∧ i2 ∧ ¬i4 then
24:         return covered-by                   ▷ (1, 1, ?, ∅)
25:     end if
26:     for each p ∈ DISCRETIZE(∂B) do
27:         T ← POINTCHECK(p, A);
28:         if T = in then                      ▷ A^0 ∩ ∂B ≠ ∅
29:             i3 ← TRUE
30:             break (goto line 35)
31:         else if T = out then    ▷ no overlap here, so it must be A^0 ∩ ∂B = ∅
32:             break (goto line 35)
33:         end if
34:     end for
35:     if ¬i1 ∧ ¬i3 then                       ▷ no need to check ¬i2 ∧ i4
36:         return disjoint                     ▷ (∅, ∅, ∅, 1)
37:     else if i1 ∧ ¬i3 then
38:         return meets                        ▷ (1, ∅, ∅, 1)
39:     else if ¬i1 ∧ i3 then
40:         return contains                     ▷ (∅, ∅, 1, 1)
41:     else if i1 ∧ i3 then
42:         return covers                       ▷ (1, ∅, 1, 1)
43:     end if
44: end procedure
45:
46: procedure DISCRETIZE(X)          ▷ parameter D is the linear point density
47:     ℓ ← length of ∂X
48:     return ℓD equally spaced points on ∂X
49: end procedure
```

through the points of the boundary ∂B, checking each point against A. As soon as a point in ∂B is found to be an interior point of A, the algorithm breaks out of the loop with $i_3 = \mathit{true}$. The algorithm also breaks out of the loop if a point in ∂B is found to be an exterior point of A, since we know that A and B do not overlap. Finding one point of ∂B outside of A implies that no points will be found inside, so the loop terminates with $i_3 = \mathit{false}$.

After the loop, i_1 and i_3 can be used to identify one of the remaining possible relations, namely *disjoint*, *meets*, *contains*, or *covers* (lines 35–43).

3.3 Boundary Discretization

The algorithm tests a finite number of points in a boundary to find a witness for each intersection. These points are chosen uniformly along the boundary by the procedure DISCRETIZE based on a *linear point density* parameter D. More specifically, DISCRETIZE starts with the points that define the boundary (i.e., the end points of the line segments that compose the boundary) and then adds equally spaced points on each segment to reach a total density of D points per unit of length. Let A be a region defined by n line segments of total length ℓ. Assuming $D \geq n/\ell$, DISCRETIZE selects all the n end points of the line segments, plus another $\ell D - n$ equally spaced points along ∂A. Of course, the point density parameter D has a crucial effect on performance and precision. Higher densities mean higher precision but also potentially higher execution times. The setting of this parameter is therefore application specific.

Notice that the algorithm is likely to iterate only over the first region boundary ∂A, with a run-time complexity proportional to the length of ∂A. Therefore, one way to save some time is to evaluate the topological relation between B and A instead of A and B when the length of ∂B is smaller than the length of ∂A. In this case, the algorithm must translate the result of the evaluation between B and A back to the corresponding relation between A and B. This is easily done, since *disjoint*, *meets*, *overlaps*, and *equals* are symmetric relations that do not need translation, and A *contains* B is equivalent to B *inside* A, and A *covered-by* B is equivalent to B *covers* A.

Finally, to increase numerical robustness in the algorithm, we introduce a halo around each point with radius $r = d/2$, where d is the (shortest) distance between the point and its adjacent points along the same line segment. This provides a more robust and qualitatively better result by providing a smoother transition from *disjoint* to *overlaps* via an extended *meets* region.

3.4 Complexity

In the worst case, Algorithm 1 executes two full loops over the discretized boundaries of the input regions. Each loop starts with an invocation of the DISCRETIZE procedure, and each iteration of the loop invokes POINTCHECK. Let n be the total number of line segments in the boundaries of the two input regions, and let ℓ be the total length of those boundaries. Then, the complexity of the POINTCHECK

algorithm we use is $O(n)$, and with a point density D, the complexity of DISCRETIZE is $O(\ell D)$. Therefore, since there are a total of ℓD iterations, the overall complexity of Algorithm 1 is $O(n\ell D)$.

4 Indexing and Simplification

The algorithm described in Section 3 simply evaluates one spatial constraint at a time. If we were to incorporate that algorithm as is within a broker/router, the broker/router would have to process each incoming message using a linear scan of all spatial constraints in its forwarding table. In this section we present an indexing structure and simplification method that are designed to avoid linear scans and reduce the sheer number of spatial constraints that must be checked.

4.1 Spatial Index

The indexing structure presented here is an extension of the well-known Guttman R-tree index [20]. Many optimized versions and variants of the original R-tree have been proposed, including the R*-tree [2], TV-tree [25], X-tree [4], and R+-tree [28]. They are mainly used in computer-aided design, GIS database, and computer graphics applications, where fast spatial searching is a necessity.

R-trees have also been employed in publish/subscribe systems, but for a completely different purpose: Rather than representing constraints on 2D spatial objects, they have been used to represent constraints on values drawn from primitive data types, such as integers. For example, Bianchi et al. [6] introduce an R-tree index called the DR-tree to capture the covering relation among subscriptions, such that when a match for a subscription s_1 implies a match for a subscription s_2 they can avoid processing s_2 if a match is found for s_1. Such uses of R-tree-like indexing structures view the constraints as forming axis-aligned, rectangular "value spaces" and simply evaluate them for what we would call here the *covers* or *covered-by* relations. Instead, we develop a variant of the R-tree to represent constraints on true 2D objects of irregular shape, and evaluate them for a broader set of relations meaningful in the realm of physical space.

Guttman's R-tree represents complex spatial objects by covering them with less complex ones. Specifically, the R-tree uses *minimum bounding rectangles* (MBRs) to represent n-dimensional objects. R-trees are structurally similar to other search trees, particularly B-trees. Each node t in an R-tree holds an MBR R_t that contains or covers all spatial objects stored in t's subtree. This property guides the search over the R-tree: The search algorithm starts with a query MBR R_q and walks the tree by descending into every subtree t whose MBR R_t overlaps with R_q. The search then returns all visited objects that are contained in R_q.

The spatial index we have developed is based on the original R-tree and uses MBRs to represent sets of stored objects. However, unlike an R-tree that stores spatial objects, this index must store *constraints* on spatial objects. We call such a structure a *constraint R-tree*, or *CR-tree*. A CR-tree extends the algorithms of the original R-tree to implement a fast search over a potentially large set of

Table 2. MBR relations for each type of constraint

Constraints	MBR relations
disjoint (D)	$D \vee M \vee O \vee I \vee cB \vee Ct \vee Cv$
meets (M)	$M \vee O \vee E \vee I \vee cB \vee Ct \vee Cv$
overlaps (O)	$O \vee E \vee I \vee cB \vee Ct \vee Cv$
inside (I)	I
contains (Ct)	Ct
covered-by (cB)	$cB \vee E$
covers (Cv)	$Cv \vee E$
equal (E)	E

spatial constraints. In particular, the search takes a query object q representing a value in a message, and returns all the spatial constraints stored in the CR-tree that are satisfied by q. A CR-tree stores constraints in its leaves, but is otherwise structurally identical to an R-tree. Each node t in a CR-tree holds the MBR of the union of the regions that define all the constraints stored in t's subtree. In addition, node t holds a list of all the *disjoint* constraints stored in t's subtree. CR-tree insertion and deletion use the corresponding R-tree algorithms, plus a simple linked-list maintenance operation to update the list of *disjoint* constraints.

The CR-tree search algorithm is also based on the R-tree algorithm. It starts by computing the MBR R_q of the query object q and then walks through the CR-tree to find potential matching constraints. To decide whether to visit a node t in the CR-tree, the search algorithm evaluates the topological relation between the query MBR R_q and the node MBR R_t. If t is an internal node, the search proceeds as in an R-tree by visiting t whenever R_q intersects R_t. If R_q does not intersect R_t, then the search algorithm does not visit t, but instead treats all the *disjoint* constraints associated with t as having been immediately matched.

If t is a leaf node, then the search algorithm visits t and evaluates the constraint stored in t if any one of a set S_t of specific relations holds between R_q and R_t. The specific set S_t associated with node t depends on the spatial constraint stored in t. For example, if t stores a *covers* constraint with comparison region X (bounded by t's MBR R_t), then a search with query MBR R_q must visit t and consider that *covers* constraint if R_q *covers* R_t or if R_q *equals* R_t. Notice that these relations are evaluated between rectangles (i.e., MBRs), and can therefore be checked in constant time without resorting to Algorithm 1, which is used only at the point where individual constraints must be evaluated. Table 2 shows the complete mapping between spatial constraints stored in leaf nodes of the CR-tree and the corresponding set of MBR relations [11].

Thus, we can view the CR-tree as a means to reduce the set of constraints that must be thoroughly checked using Algorithm 1. Moreover, the CR-tree does not simply exclude constraints because they do not match, but also allows us to immediately decide that some of those constraints *do* match, again without resorting to Algorithm 1. In this sense, the CR-tree, unlike the R-tree from which it is derived, is more than just a traditional search tree, but also a participant in the evaluation process.

Another important element of an R-tree is the splitting algorithm, which is used to split nodes when they become too large due to the insertion of many objects. In our CR-tree we use the *quadratic-cost* splitting algorithm [20], which provides a good balance between simplicity and performance.

4.2 Simplification

Simplification is the process of removing redundant constraints from a forwarding table. For example, a filter on integer values $x \neq 0 \wedge x > 10$ can be rewritten simply as $x > 10$ because $x > 10 \Rightarrow x \neq 0$ for all x or, in other words, because $x \neq 0$ "covers" $x > 10$. We develop a similar form of simplification scheme for 2D spatial constraints. As is commonly done in the publish/subscribe literature, we first define "covers" and "conflicts" relations between spatial constraints. Let C_1 and C_2 be two spatial constraints defined by spatial relation rel_1 and region R_1, and relation rel_2 and region R_2, respectively. We then say that C_1 *covers* C_2 iff $X\ rel_2\ R_2 \Rightarrow X\ rel_1\ R_1$ for all regions X. Similarly, we say that C_1 *conflicts* with C_2 iff $\neg(X\ rel_1\ R_1 \wedge X\ rel_2\ R_2)$ for all regions X.

As is the case for basic data types, the covers and conflicts relations between two spatial constraints C_1 and C_2 can be evaluated efficiently on the basis of the topological relation between R_1 and R_2, and can therefore be used to simplify predicates. Let P be a predicate consisting of a disjunction of n filters $f_1, f_2, \ldots, f_n$, where each filter f_i consists of a conjunction of m_i constraints $C_{i,1}, C_{i,2}, \ldots, C_{i,m_i}$. Pairwise redundant constraints in a filter, entire filters, and pairwise redundant filters can now be identified and eliminated through the following simplification rules.

Rule 1. A filter f_i can be removed from predicate P if, for any pair of constraints $C_{i,j}$ and $C_{i,k}$ in f_i, $C_{i,j}$ *conflicts* with $C_{i,k}$. This is because, from the definition of the conflict relation, f_i is always *false*.

Rule 2. If $C_{i,j}$ and $C_{i,k}$ are two constraints in the same filter f_i, then $C_{i,j}$ can be removed if $C_{i,j}$ *covers* $C_{i,k}$. This is because, from the definition of the cover relation, any region that satisfies $C_{i,k}$ also satisfies $C_{i,j}$.

Rule 3. Let f_h and f_i be two filters in predicate P. We can eliminate filter f_i from P if for all constraints $C_{h,k}$ in f_h there exists a constraint $C_{i,j}$ in f_j such that $C_{h,k}$ *covers* $C_{i,j}$. This is because, from the definition of the covers relation, f_i is satisfied by a subset of the messages that satisfy f_h. This rule can also be seen as the definition of a covering relation between filters.

We implement a simplification algorithm based on these rules. The implementation realizes the covers and conflicts relations for all combinations of spatial relations in the intuitive way. For example, (x *covered-by* A) *conflicts with* (x *overlaps* B) if (A *disjoint* B). We discuss the effectiveness of spatial simplifications as part of the experimental evaluation presented Section 5.

5 Evaluation

We now present an experimental evaluation of the matching performance and general scalability of the spatial model and its implementation. In particular, we

ask whether the absolute performance is acceptable and, more importantly, how the matching time scales with the number of spatial constraints. For this purpose we use both synthetic workloads and workloads derived from real GIS data. The synthetic workloads are useful in highlighting the scalability of the matching algorithm, especially in worst-case configurations, while the GIS workloads show how the matching algorithm performs in realistic situations.

We also compare the approximate matching of our algorithm against the exact matching of a *polygon-intersection* algorithm. Polygon intersection is a binary decision problem that is unable to distinguish among the specific relations of the region-connection calculus. In particular, a *false* result from polygon intersection indicates either *meets* or *disjoint*, while a *true* result indicates either *overlaps*, *equals*, *inside*, *contains*, *covers*, or *covered-by*. Despite this shortcoming, it is important to consider polygon intersection, since it is a well-studied problem in classical computational geometry having known efficient algorithms [3] and well-engineered implementations.[3] We use it as a performance benchmark.

Synthetic Regions. We generate synthetic regions, serving as spatial constraints or message attributes, within a rectangular universe U of size U_X by U_Y. We initially choose two points within U, uniformly at random, that represent the minimum bounding rectangle R for the region we generate. We then select three or four points each on a separate segment on the perimeter of R. We complete the generation of the region by adding points chosen uniformly at random in R, up to a total of 20 points, and then order the points to form the boundary of the region in such a way that the boundary has no intersecting segments (as per Definition 1 in Section 2.1). Finally, we exclude regions with an area smaller than 10% of the size of the universe U so as to obtain regions of similar size and, therefore, a more diverse combination of topological relations.

Implementation. The experiments use an implementation of the spatial model integrated within the Siena Fast Forwarding (SFF) matching framework[4] and executed with a sequential, single-threaded matching process on an Intel i5 processor with 6GB of DDR3 memory. A crucial parameter in the implementation is the linear point density D. The first set of experiments use a fixed point density of 2000 points per universe half-length, that is $D = 2000/(U_X + U_Y)$. Because we use a square universe ($U_X = U_Y$), the chosen point density intuitively can be seen as a 1000×1000 pixel image. We experimented with other point densities and found that the performance varies more or less linearly with the point density. In the second set of experiments, we study the trade off between accuracy and performance of the matching algorithm under various point densities.

5.1 Worst-Case Performance and Scalability

In our first experiment we measure the time needed to match an input region against synthetic workloads of increasingly larger sets of constraints. This experiment is intended to evaluate the performance of Algorithm 1 and the impact of

[3] http://www.cgal.org (CGAL: Computational Geometry Algorithms Library).
[4] http://www.inf.usi.ch/carzaniga/cbn/forwarding/

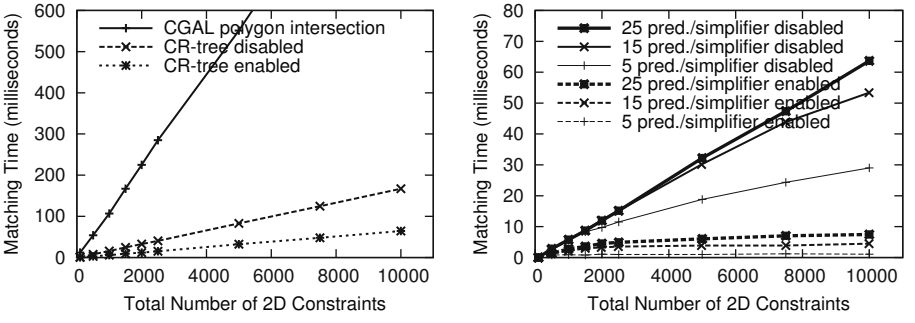


Fig. 4. Performance of the matching algorithm: (a) with and without the CR-tree constraint index enabled and (b) with and without spatial simplification enabled

the CR-tree index. To do so, we generate workloads consisting of N predicates (N up to 10000), each composed of a single filter with a single constraint. All constraints are on the same attribute and all messages contain the same single attribute associated with a 2D region, so each message carries an input region that must in principle be evaluated against all N constraints.

Figure 4a shows the performance of Algorithm 1 in isolation and together with the CR-tree, and in comparison with an exact polygon-intersection algorithm. Each data point represents the average matching time of several input regions over several sets of the same number of constraints. The variability of the matching time is minimal (1–2ms), so we do not show this in the plots. The solid line represents a sequential execution of CGAL's polygon-intersection algorithm over a list of all the constraints. The dashed line represents the same sequential execution of Algorithm 1, while the dotted line represents the use of the CR-tree index to reduce the number of examined constraints.

With the point density used in this experiment, Algorithm 1 incurs an error rate of only 0.089% and runs about six times faster than the exact CGAL polygon-intersection algorithm. Thus, the experiment shows that the approximate algorithm offers a good combination of performance and accuracy. We repeated this experiment with different point densities and found that the approximate algorithm is still faster than the exact algorithm with a point density of 12000 points per universe half-length and a corresponding error rate of only about 0.015%.

The experiment demonstrates that the absolute matching times are within reasonable bounds. Notice that it represents an extreme worst-case scenario, requiring the evaluation of thousands of constraints on the same attribute and without the possibility of taking shortcuts. The results also show that the CR-tree is effective in reducing the matching time of a linear scan. This, too, is a worst-case scenario for the CR-tree, since we use regions that are large with respect to the universe and, therefore, they do not lend themselves to an effective partition under disjoint MBRs. In the experiment of Section 5.2 we see that when applied to realistic GIS workloads the performance is substantially better.

In a second experiment, shown in Figure 4b, we evaluate the performance of the constraint index (Algorithm 1 within a CR-tree) in a scenario of more articulated predicates. In particular, we construct workloads in which a total of N constraints (N up to 10000) are distributed over 5, 15, and 25 predicates, each consisting of filters of two constraints. Even though this experiment explores a more realistic set of predicates, we still focus only on the performance of the 2D spatial components of the matching algorithm. Therefore, as in the first experiment, we use a single attribute in all constraints and in all messages.

Each line in Figure 4b shows the matching time as a function of the total number of constraints. We can see that the behavior of the SFF matcher extended with support for 2D spatial objects is consistent with that of the original SFF matcher [8], and that the absolute matching times are also reasonable. In particular, the matching times tend to be flat for larger and larger predicates. This effect is not due to the 2D spatial constraint index, but rather to the structure of the SFF algorithm and also to the nature of the matching problem: with fewer and larger predicates, a message is likely to match at least one filter for all predicates, thereby cutting short the full evaluation.

The results in Figure 4b also show the effectiveness of the spatial simplifier. The solid lines show the performance for non-simplified workloads, while the corresponding dashed lines show the performance under simplification. Notice that the simplification method effectively accelerates the flattening of the matching times. This is because as more filters (and constraints) are added to the same predicate, more of those filters (and constraints) are likely to become redundant.

5.2 Accuracy and Performance in Realistic Configurations

To evaluate our algorithm in realistic configurations, we derive workloads from three GIS data sets. The first contains 85550 polygons representing vegetation in southern California, the second 17048 polygons representing natural features (parks, forests, woods, water areas, etc.) in the Rhône-Alpes district of France, and the third 4170 polygons representing buildings in central London.[5]

Using the process described at the beginning of this section we generate three different types of input regions for placement into messages, referred to here as A, B, and C. Type A represents regions that are relatively round and uniform shapes, such as clouds and storms, having nearly square MBRs and at most 10 boundary points. Type B represents regions of irregular and more complex shapes, such as the movement of herds and environmental hazards such as forest fires. These regions exhibit correspondingly less-regular MBRs and between 30 and 50 boundary points. Finally, type C represents narrow and long regions, described using up to five boundary points, such as the paths taken by aircraft at high altitudes. The sizes of all three types of regions are chosen randomly to have an MBR area between 1% and 25% of the universe.

We first study how the point density affects the accuracy of the matching algorithm. We run the matching algorithm using 100 of each of the three types

[5] We obtained the first data set from `http://atlas.ca.gov`, and the second two from `http://www.openstreetmap.org`.

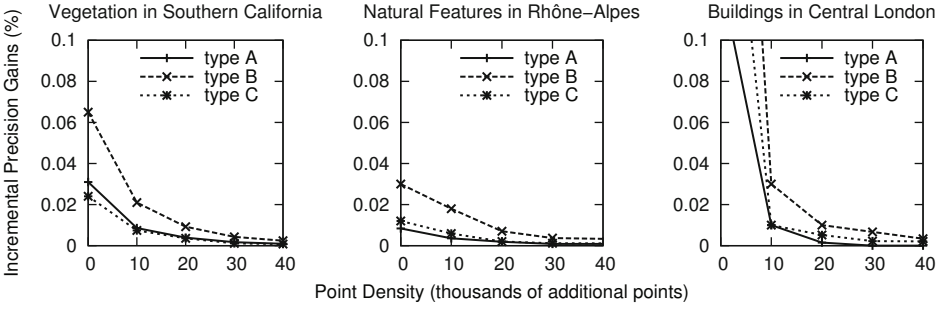


Fig. 5. Incremental precision gain as a function of the point density

of input regions over each of the three GIS-derived sets of spatial constraints. We vary the point density from 0 to 40000 points per universe half-length. Note that these are boundary points considered by the matching algorithm *in addition to* the points that define the boundary (so, a point density of 0 makes sense). For each point density D we compare the results of the matching process for density D with those using density $D + 10000$, and record the difference between the two sets of matched predicates.

We plot the results of this analysis in Figure 5. At a high level we can see that all the curves show very low differences and converge toward an exact match. To appreciate the meaning of each value, consider an incremental precision gain of 0.02% at density $D = 10000$. That 0.02% can be interpreted as the probability of the matching results differing by one predicate (matching in one case and not matching in the other) at densities $D = 10000$ and $D + 10000 = 20000$.

We next evaluate the performance of the matching algorithm. For this experiment we set the point density to $D = 10000$ (intuitively corresponding to a 5000×5000 universe). The experiments are set up according to two scenarios corresponding to the two sets of experiments presented in Section 5.1. For the first scenario, we construct a filter with one constraint from each polygon in the GIS data set, choosing 2D operators uniformly at random, and then we assign one filter per predicate. With one filter per predicate, this scenario is intended to stress-test the matching algorithm with and without the CR-tree enabled. In the more realistic second scenario, we construct 25 predicates by building filters consisting of two constraints generated from the regions in the GIS data set, and then simply combine the generated filters into the 25 predicates.

The results are reported in Table 3. The table shows the average matching times of several input regions, for each input region type, and over the three data sets. (Variance in matching times is minimal so not shown.) Notice that the results in both scenarios are consistent with the general qualitative characteristics of the matching algorithm outlined in the synthetic workload experiments of Section 5.1. However, in this more realistic case, the absolute performance of the algorithm is substantially better, with total matching times of fractions of

Table 3. Average matching times for GIS-derived workloads

			Scenario 1 1 filter per predicate 1 constraint per filter		Scenario 2 25 predicates 2 constraints per filter	
source data set	number of constraints	region type	CR-tree		simplifier	
			disabled	enabled	disabled	enabled
Vegetation in southern California	85550	A	1841.00 ms	*73.00 ms*	67.00 ms	*0.18 ms*
		B	4000.00 ms	*89.00 ms*	78.00 ms	*0.54 ms*
		C	1512.00 ms	*7.00 ms*	4.40 ms	*0.12 ms*
Natural features in Rhône-Alpes	17048	A	951.00 ms	*108.00 ms*	59.00 ms	*1.40 ms*
		B	1323.00 ms	*93.00 ms*	169.00 ms	*6.40 ms*
		C	851.00 ms	*7.00 ms*	72.00 ms	*0.53 ms*
Buildings in central London	4170	A	59.00 ms	*1.40 ms*	0.95 ms	*0.75 ms*
		B	169.00 ms	*6.40 ms*	5.00 ms	*1.24 ms*
		C	72.00 ms	*0.53 ms*	0.32 ms	*0.05 ms*

milliseconds. This is because regions within the GIS data sets represent real objects and, therefore, tend to overlap considerably less as well as tend more often to be mutually disjoint than those of the worst-case synthetic workloads.

6 Conclusions

We have presented an enhancement to existing content-based publish/subscribe systems with support for a 2D spatial data type and eight associated relational operators. We described an algorithm for evaluating the spatial relations that is founded on a dynamic discretization method. In order to make the use of this new data type practical we developed an indexing structure for spatial constraints, called the CR-tree, as well as a simplification method for eliminating redundant spatial constraints. Our experimental evaluation demonstrates the effectiveness and scalability of our approach when integrated into a state-of-the-art publish/subscribe matching engine.

In future work we plan to further refine the CR-tree by exploring improved methods for splitting nodes as the number of constraints grows. The method we currently use is a generic one developed for the original R-tree spatial-object index. We suspect that there might be more effective methods, possibly heuristic in nature, tailored to an index of spatial constraints.

References

1. Bauer, M., Rothermel, K.: Towards the observation of spatial events in distributed location-aware systems. In: Proceedings of the International Conference on Distributed Computing Systems Workshops, pp. 581–582 (2002)
2. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: An efficient and robust access method for points and rectangles. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 322–331 (1990)

3. Bentley, J.L., Ottmann, T.A.: Algorithms for reporting and counting geometric intersections. IEEE Transactions on Computers 28, 643–647 (1979)
4. Berchtold, S., Keim, D.A., Kriegel, H.P.: The X-tree: An index structure for high-dimensional data. In: Proceedings of the 22nd International Conference on Very Large Data Bases, San Francisco, California, pp. 28–39 (September 1996)
5. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry: Algorithms and Applications. Springer, Heidelberg (2008)
6. Bianchi, S., Datta, A., Felber, P., Gradinariu, M.: Stabilizing peer-to-peer spatial filters. In: Proceedings of the 27th International Conference on Distributed Computing Systems (June 2007)
7. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems 19(3), 332–383 (2001)
8. Carzaniga, A., Wolf, A.L.: Forwarding in a content-based network. In: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pp. 163–174 (2003)
9. Chen, X., Chen, Y., Rao, F.: An efficient spatial publish/subscribe system for intelligent location-based services. In: Proceedings of the 2nd International Workshop on Distributed Event-Based Systems, pp. 1–6. ACM (2003)
10. Chen, Y., Chen, X., Rao, F., Yu, X.L., Li, Y., Liu, D.: LORE: An infrastructure to support location-aware services. IBM Journal of Research and Devevelopment 48, 601–615 (2004)
11. Clementini, E., Sharma, J., Egenhofer, M.J.: Modelling topological spatial relations: Strategies for query processing. Computers and Graphics 18(6), 815–822 (1994)
12. Cohn, A.G., Renz, J.: Qualitative spatial representation and reasoning. In: Handbook of Knowledge Representation, Foundations of Artificial Intelligence, vol. 3, pp. 551–596. Elsevier (2008)
13. Egenhofer, M.J., Franzosa, R.D.: Point-set topological spatial relations. International Journal of Geographical Information Systems 5(2), 161–174 (1991)
14. Egenhofer, M.J., Franzosa, R.D.: On the equivalence of topological relations. International Journal of Geographical Information Systems 8(6), 133–152 (1994)
15. Egenhofer, M.J., Herring, J.R.: Categorizing binary topological relations between regions, lines, and points in geographic databases. Tech. rep., Department of Surveying Engineering, University of Maine (1990)
16. Eugster, P., Garbinato, B., Holzer, A.: Location-based publish/subscribe. In: Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications, pp. 279–282 (2005)
17. Greene, D.H., Yao, F.F.: Finite-resolution computational geometry. In: Proceedings of the Annual IEEE Symposium on Foundations of Computer Science, pp. 143–152 (1986)
18. Guting, R.H., Schneider, M.: Realms: A Foundation for Spatial Data Types in Database Systems. In: Abel, D.J., Ooi, B.-C. (eds.) SSD 1993. LNCS, vol. 692, pp. 14–35. Springer, Heidelberg (1993)
19. Guting, R.H., Schneider, M.: Realm-based spatial data types: The ROSE algebra. The International Journal on Very Large Data Bases 4(2), 243–286 (1995)
20. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 47–57 (1984)
21. Hoffmann, C.M.: The problems of accuracy and robustness in geometric computation. Computer 22, 31–40 (1989)

22. Hu, S.Y., Wu, C., Buyukkaya, E., Chien, C.H., Lin, T.H., Abdallah, M., Jiang, J.R., Chen, K.T.: A spatial publish subscribe overlay for massively multiuser virtual environments. In: Proceedings of the International Conference on Electronics and Information Engineering, pp. 314–318 (2010)
23. Laszlo, M.J.: Computational Geometry and Computer Graphics in C++. Prentice-Hall (1995)
24. Leung, H.K.Y., Burcea, I., Jacobsen, H.A.: Modeling location-based services with subject spaces. In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, Toronto, Canada, pp. 171–181 (2003)
25. Lin, K.I., Jagadish, H., Faloutsos, C.: The TV-tree: An index structure for high-dimensional data. The International Journal on Very Large Data Bases 3(4), 517–542 (1994)
26. Papadias, D., Theodoridis, Y.: Spatial relations, minimum bounding rectangles, and spatial data structures. International Journal of Geographical Information Science 11, 111–138 (1997)
27. Ranganathan, A., Al-Muhtadi, J., Chetan, S.K., Campbell, R., Mickunas, M.D.: MiddleWhere: A Middleware for Location Awareness in Ubiquitous Computing Applications. In: Jacobsen, H.-A. (ed.) Middleware 2004. LNCS, vol. 3231, pp. 397–416. Springer, Heidelberg (2004)
28. Sellis, T.K., Roussopoulos, N., Faloutsos, C.: The R-tree: A dynamic index for multi-dimensional objects. In: Proceedings of the 13th International Conference on Very Large Data Bases, pp. 507–518 (1987)
29. Xu, Z., Jacobsen, H.A.: Adaptive location constraint processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 581–592 (June 2007)
30. Yan, T.W., Garcia-Molina, H.: Index structures for selective dissemination of information under the Boolean model. ACM Transactions on Database Systems 19(2), 332–364 (1994)

FAIDECS: Fair Decentralized Event Correlation*

Gregory Aaron Wilkin, K.R. Jayaram, Patrick Eugster, and Ankur Khetrapal

Department of Computer Science
Purdue University
{gwilkin,jayaram,peugster}@cs.purdue.edu,
ankur@alumni.purdue.edu

Abstract. Many distributed applications rely on *event correlation*. Such applications, when not built as ad-hoc solutions, typically rely on centralized correlators or on *broker overlay* networks. Centralized correlators constitute performance bottlenecks and single points of failure; straightforwardly duplicating them can hamper performance and cause processes interested in the same correlations to reach different outcomes. The latter problem can manifest also if broker overlays provide redundant paths to tolerate broker failures as events do not necessarily reach all processes via the same path and thus in the same order.

This paper describes FAIDECS, a generic middleware system for *fair* decentralized correlation of events *multicast* among processes: processes with identical interests reach identical outcomes, and subsumption relationships among subscriptions are considered for respectively delivered *composite* events. Based on a generic subset of FAIDECS's predicate language, we introduce properties for composite event deliveries in the presence of process failures and present novel decentralized algorithms implementing these properties. Our algorithms are compared under various workloads to solutions providing equivalent guarantees.

Keywords: event, correlation, fair, reliable, multicast, decentralized.

1 Introduction

The abstraction of application *events* is useful not only for reasoning about distributed systems [18], but also for building such systems [5,26].

Events: composition and correlation. Event *correlation* [8] enables higher-level reasoning about interactions by supporting the assembly of *composite* events from elementary events [20,19]. Traditional uses of correlation include intrusion detection [17]; network monitoring [16] enables the improvement of resource usage, e.g., in data centers. More recent application scenarios for correlation include embedded and pervasive systems [13], and sensor networks [22]. *Complex event processing* (CEP) is a computing paradigm based on event correlation, with applications to business process management and algorithmic trading.

* Financial support by NSF grants 0644013 and 0834529, DARPA grant N11AP20014. Any opinions, findings, conclusions, or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF or DARPA.

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 228–248, 2011.

Challenges for event correlation middleware. Reasoning about event composition is, however, involving. Early work in active databases [6] explored syntax and semantics of correlation, pinpointing options. Consider a sequence of events $e_1^1 \cdot e_2^1 \cdot e_1^2$, where e_l^k is a the l-th received event (instance) of event type T_k. This sequence can be matched by a "subscription" correlating two event types T_1 and T_2 as $[e_1^1, e_1^2]$ (*first received* first) or as $[e_2^1, e_1^2]$ (*most recent* first). However, corresponding systems are centralized and consider events to be *uni*cast.

Many theoretical and practical efforts on event correlation in publish/subscribe systems [5] consider decentralized setups and *multi*cast but focus on efficiency or the number of aggregations, yielding only best-effort guarantees on event delivery. Consider an online auction where the bidding price of a product or advertisement slot is event-driven. If two processes participating in the auction observe the same events in different orders (e.g., one receives the sequence above, the second one receiving $e_1^2 \cdot e_2^1 \cdot e_1^1$), then the event correlation middleware might be unfair to the first process if e_1^2 has information that is critical to placing an optimal bid. Or, consider assembly line surveillance through two monitors for fault tolerance. If they observe events differently, they might yield contradicting reports or alarms. During decentralized event correlation, one might not only expect that processes with identical subscriptions deliver identical sets of events, but also that if the subscription of a first process p_i "covers" that of a second process p_j, then p_i would deliver anything that p_j does. In production chains, the same complex events triggering alarms can be combined with further events for taking actions further down the chain or triggering more specific alarms. Such *subsumption* is natural in publish/subscribe systems and even key to scalability [5]. Of course, correlation-based systems can currently be designed individually to achieve such properties, e.g., by using proxy processes to merge and multiplex event streams to replicas to achieve agreement; corresponding solutions are hardly generic though, and can introduce bottlenecks to performance and dependability.

Contributions. This paper presents FAIDECS (*FAI*r *D*ecentralized *E*vent *C*orrelation *S*ystem – "Fedex"), a middleware system for *fair* decentralized correlation of events *multicast* among processes. Our exact contributions are:

- After presenting related work (Section 2) and introducing the system model and assumptions (Section 3), we present clear and feasible *properties* for *aggregated* deliveries of *sets* of events based on a concise and generic event correlation sub-grammar in FAIDECS (Section 4). While in single event (message) delivery scenarios, several families of properties have been proposed and investigated (e.g., agreed delivery [14], probabilistic delivery [4], ordering properties [11]), corresponding properties for the better understanding of correlation-based systems and ensuring "*logical correctness and integrity*" [21] are namely still lacking. Our properties provide *fairness* in the face of failures of processes responsible for merging events: either all or none of the depending processes cease to receive the desired events, while common overlays (e.g., [19]) might continue to deliver differing sets of events

to subsets of interested processes. Our properties also include a notion of subsumption on correlation patterns.
- We introduce novel pragmatic algorithms implementing our delivery properties (Section 5). For illustration purposes, we first describe simple algorithms based on a group broadcast black box. Then we present decentralized solutions implemented in FAIDECS based on a distributed hash-table (DHT), and present the use of lightweight redundancy mechanisms used for fault tolerance.
- An implementation of our algorithms in FAIDECS is evaluated under different workloads (Section 6). We quantify the benefits of our decentralized approach by comparing them with sequencer-based and token-based total order broadcast protocols providing comparable properties.

We conclude with final remarks in Section 7. Due to space limitations, we refer to a companion technical report [24] for discussions of alternative properties, or a formal proof that agreement on *composite* events *requires* a total order on individual events or an equivalent oracle.

2 Related Work

Many early approaches for composite event detection are based on *active databases* that employ *centralized* detection of events (e.g., [6]). A composite event is a pattern of events that a subscriber may be interested in. A composite *subscription* is a pattern describing the interests of the subscriber.

Event correlation has been vigorously investigated in the context of *content-based publish/subscribe systems*. Most such systems rely on a *broker network* for routing events to the subscribers (e.g., SIENA [5] and Gryphon [2]). *Advertisements* are typically used to form routing trees in order to avoid propagating subscriptions by flooding the broker network. Upon receiving an event e, a broker determines the subset of parties (subscribers and brokers) with matching interests, and forwards e to them. Subscription *subsumption* [5] is used to summarize subscriptions and avoid redundant matching on brokers and redundant traffic among them. If any event e that matches a first subscription also matches a second one, then the latter subscription subsumes the former one.

A broker network can be used to gather all publications for the elementary subscriptions and perform correlation matching. A successful match yields a composite event which is delivered to interested subscribers, where no guarantees are typically provided on correlation. If the events matching a composite subscription shared by two subscribers are produced by several publishers, then unless the subscribers are connected to a same edge broker, they may receive the events through different routes. This leads to different orders among the events and consequently to different composite events for the two subscribers. PADRES [19] performs composite event detection for each subscription at the first broker that accumulates all the individual subscriptions, providing no global properties. In the context of Hermes [20], *complex event detectors* using an interval timestamp model are proposed as a generic extension for existing middleware

architectures. Hermes uses a DHT to determine rendezvous nodes for publishers and subscribers; however, this can create single points of failure. The framework we propose is inspired by Hermes in that our framework uses specific merger nodes for specific combinations of types, determined by a DHT. However, we replicate the mergers for availability and connect them such as to ensure agreement, ordering and subsumption on composite events.

Stream processing is a paradigm closely related to event correlation and much investigated in the last few years. Research around database-backed systems like Aurora [1] or Borealis [23] has led the path. These systems, however, focus on correlation over streams of events with respect to single destinations and do not consider multicasting. Straightforwardly merging two same streams at two different nodes leads to different outcomes. StreamBase[1] is a commercial offspring of these efforts. Cayuga [8] is a generic correlation engine supporting correlation across streams and is based on a very expressive language but is centralized. The Gryphon publish/subscribe systems has similarly added support for streams [26]. Again, the focus is efficiency, leaving properties unclear.

3 Preliminaries

We assume a system Π of *processes*, $\Pi=\{p_1, ..., p_u\}$ connected pairwise by reliable channels [3] offering primitives to SEND (non-blocking) and receive (RECEIVE) messages. We consider a crash-stop failure model [14], i.e., a *faulty* process may stop prematurely and does not recover. We assume the existence of a discrete global clock to which processes do not have access and that an algorithm run R consists in a sequence of events on processes. That is, one process performs an action per clock tick which is either of a (a) protocol action (e.g., RECEIVE), (b) an internal action, or (c) a "no-op". A process is faulty in a run R if it fails during R, otherwise *correct*.

A failure pattern F is a function mapping clock times to processes, where $F(t)$ gives all the crashed processes at time t. Let $crashed(F)$ be the set of all processes $\in \Pi$ that have crashed during R. Thus, for a correct process p_i, $p_i \in correct(F)$ where $correct(F) = \Pi - crashed(F)$ [14].

For brevity and clarity, we adopt in the following a more formal notation for properties than common. Consider, for instance, the well-known problem of Total Order Broadcast (TOBcast) [14] defined over primitives TO-BROADCAST and TO-DELIVER, which will be used for comparison later on. We denote $\text{TO-DELIVER}^i(e)_t$ as the TO-delivery of a message conveying an event e by process p_i at time t, and similarly, $\text{TO-BROADCAST}^i(e)_t$ denotes the TO-broadcasting of e by p_i at time t. We elide any of i, t, or e when not germane to the context. We write $\exists a$ for an action a (e.g., SEND, TO-BROADCAST) as a shorthand for $\exists a \in R$. The specification of Uniform TOBcast thus becomes:

TOB-NO DUPLICATION: $\exists \text{TO-DELIVER}^i(e)_t \Rightarrow \nexists \text{TO-DELIVER}^i(e)_{t'} \mid t' \neq t$

TOB-NO CREATION: $\exists \text{TO-DELIVER}(e)_t \Rightarrow \exists \text{TO-BROADCAST}(e)_{t'} \mid t' < t$

[1] http://www.streambase.com/.

TOB-Validity: $\exists$TO-BROADCAST$^i(e) \ \wedge \ p_i \in correct(F) \Rightarrow \exists$TO-DELIVER$^i(e)$

TOB-Agreement: $\exists$TO-DELIVER$^i(e) \Rightarrow \forall p_j \in correct(F) \backslash \{p_i\}, \exists$TO-DELIVER$^j(e)$

TOB-Total Order: $\exists$TO-DELIVER$^i(e)_{t_i}$, TO-DELIVER$^i(e')_{t'_i}$, TO-DELIVER$^j(e)_{t_j}$, TO-DELIVER$^j(e')_{t'_j} \ \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

4 FAIDECS Model

In this section, we specify composite events in FAIDECS and the properties achieved for corresponding deliveries (DELIVER) with respect to individually generated (MULTICAST) events. In contrast to traditional settings, DELIVER is parameterized by a "subscription" Φ and delivers *ordered sets* of typed messages representing events.

4.1 Predicate Grammar

Sets of delivered events — *relations* — represent events aggregated according to specific subscriptions. Subscriptions are combinations of predicates on events in disjunctive normal form based on the following grammar (extended BNF):

Disjunction	$\Psi ::= \Phi \mid \Phi \vee \Psi$	*Operation*	$op ::= < \mid > \mid \leq \mid \geq \mid = \mid \neq$
Conjunction	$\Phi ::= \rho \mid \rho \wedge \Phi$	*Predicate*	$\rho ::= T[i].a \ op \ v \mid T[i].a \ op \ T[i].a \mid T[i] \mid \top$

$T[i].a$ denotes an attribute a of the i-th *instance* of type T ($T[i]$) and v is a value. As syntactic sugar, we can allow predicates to refer to just T, which can be automatically translated to $T[1]$. We may use this in examples for simplicity. A type T is characterized by an ordered set of attributes $[a_1, ..., a_n]$, each of which has a type of its own – typically a scalar type such as `Integer` or `Float`. An event e of type T is an ordered set of values $[v_1, ..., v_n]$ corresponding to the respective attributes of T. We assume that types of values in predicates conform with the types of events (e.g., through static type-checking [9]). $T(e)$ returns the type of a given event e. It is important to note that we do *not* introduce a set of uniquely identified types $\{T_1, ..., T_w\}$ as we do for processes. This keeps notation more brief in that we can use $[T_1, ..., T_k]$ to refer to an arbitrary ordered set of k types, as opposed to something of the form $[T_{j_1}, ..., T_{j_k}]$.

To later simplify properties, we introduce the *empty* predicate $\top$, which trivially yields *true*. A predicate that compares a single event attribute to a value or two event attributes on the *same* event, i.e., on the same instance of a same type (e.g., $T_k[i].a \ op \ T_k[i].a'$), is a *unary* predicate. When two distinct events (two distinct types or different instances of the same type) are involved, we speak of *binary* predicates ($T_k[i].a \ op \ T_l[j].a', \ k \neq l \vee i \neq j$). We also allow *wildcard* predicates of the form $T[i]$ to be specified; such predicates simply specify a desired type $T[i]$ of events of interest. $T[i]$ implicitly also declares $T[k] \ \forall k \in [1..i-1]$ if not already explicitly declared as part of other predicates in the subscription.

We assume, for presentation brevity, a single subscription per process. The disjunction representing process p_i's subscription is represented as $\Psi(p_i)$.

We also rule out disjunctions with several identical conjunctions. In practice, we can simply remove all but one copy. By abuse of notation but unambiguously, we sometimes handle disjunctions (or conjunctions) as sets of conjunctions (or predicates). We write, for instance, $\rho_l \in \Phi \Leftrightarrow \Phi = \rho_1 \wedge ... \wedge \rho_k$ with $l \in [1..k]$.

For the following, consider an example subscription Ψ_S for an increase in three successive stock quotes after a quarterly earnings report:

$$\begin{aligned}\Psi_S = {} & \texttt{StockQuote}[0].\texttt{time} > \texttt{EarningsReport}[0].\texttt{time} \wedge \\ & \texttt{StockQuote}[1].\texttt{value} > \texttt{StockQuote}[0].\texttt{value} \wedge \\ & \texttt{StockQuote}[2].\texttt{value} > \texttt{StockQuote}[1].\texttt{value}\end{aligned}$$

We would probably want to introduce arithmetic operators on values [15] to express, e.g., that the local publication `time` of the first stock quote is within some interval of that of the earnings report. Our grammar can be easily extended by such *deterministic* constructs but is intentionally kept simple for presentation and to illustrate the independence of our algorithms from specific grammars.

4.2 Predicate Types and Evaluation

We assume that a deterministic order $\prec$ exists within subscriptions based on the names of event types, attributes, etc., which can be used for re-ordering predicates within and across conjunctions. This ordering can be lexical or based on priorities on event types and is necessary for even simplest forms of determinism and agreement. We consider subscriptions to be already ordered accordingly.

The number of events involved in a subscription is given by the number of its types and corresponding instances. More precisely, the types involved in a subscription are represented as *sequences* as they are ordered, and the same type can be admitted multiple times. Such sequences can be viewed as the *signatures* of predicates, defined as follows:

$$\begin{array}{lll} \mathbb{T}(\Phi \vee \Psi) & = \mathbb{T}(\Phi) \uplus \mathbb{T}(\Psi) & \mathbb{T}(T[i].a \ op \ v) = \mathbb{T}(T[i]) \\ \mathbb{T}(\rho \wedge \Phi) & = \mathbb{T}(\rho) \uplus \mathbb{T}(\Phi) & \mathbb{T}(\top) \quad\quad = \emptyset \\ \mathbb{T}(T_1[i].a_1 \ op \ T_2[j].a_2) & = \mathbb{T}(T_1[i]) \uplus \mathbb{T}(T_2[j]) & \mathbb{T}(T[i]) \quad\quad = [\underbrace{T, ..., T}_{i\times}] \end{array}$$

$\uplus$ stands for in-order union of sequences defined below:

$$\emptyset \uplus [T, ...] = [T, ...] \qquad [T, ...] \uplus \emptyset = [T, ...]$$

$$[\underbrace{T_1, ..., T_1}_{i\times}, T_1', ...] \uplus [\underbrace{T_2, ..., T_2}_{j\times}, T_2', ...] = \begin{cases} [\underbrace{T_1, ..., T_1}_{i\times}] \oplus ([T_1', ...] \uplus [\underbrace{T_2, ..., T_2}_{j\times}, T_2', ...]) & T_1 \prec T_2 \\ [\underbrace{T_2, ..., T_2}_{j\times}] \oplus ([T_2', ...] \uplus [\underbrace{T_1, ..., T_1}_{i\times}, T_1', ...]) & T_2 \prec T_1 \\ [\underbrace{T_1, ..., T_1}_{max(i,j)\times}] \oplus ([T_1', ...] \uplus [T_2', ...]) & T_1 = T_2 \end{cases}$$

Above, $\oplus$ represents simple concatenation. In the previous example, the types involved are thus [`EarningsReport`, `StockQuote`, `StockQuote`, `StockQuote`].

Any subscription Ψ thus involves a sequence of event types $\mathbb{T}(\Psi)=[T_1, ..., T_n]$, where we can have for $i, j \in [1..n], i < j$ such that $\forall k \in [i..j], T_k = T_i = T_j$. That

is, we can have subsequences of identical types. Such a subsequence represents a stream of events of the respective type of length $j-i+1$ ($T_k[1], ..., T_k[j-i+1]$).

A subscription is correspondingly evaluated for an ordered set of events $[e_1, ..., e_n]$, where e_i is of type T_i. The evaluation of a conjunction Φ on a relation is written as $\Phi[e_1, ..., e_n]$. For evaluation of an attribute a on an event e_i, we write $e_i.a$. Evaluation semantics for predicates are defined as follows:

$$(\Phi \vee \Psi)[e_1, ..., e_n] = \Phi[e_1, ..., e_n] \vee \Psi[e_1, ..., e_n] \qquad (T)[e_1, ..., e_n] = \mathit{true}$$
$$(\rho \wedge \Phi)[e_1, ..., e_n] = \rho[e_1, ..., e_n] \wedge \Phi[e_1, ..., e_n] \qquad (\top)[e_1, ..., e_n] = \mathit{true}$$

$$(T[i].a \ op \ v)[e_1, ..., e_n] = \begin{cases} e_{k+i-1}.a \ op \ v & T(e_k) = T \wedge (T(e_{k-1}) \neq T \\ & \quad \vee (k-1) = 0) \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$(T_1[i].a_1 \, op \, T_2[j].a_2)[e_1, ..., e_n] = \begin{cases} e_{k+i-1}.a_1 \, op \, e_{l+j-1}.a_2 & T(e_k) = T_1 \wedge (T(e_{k-1}) \neq T_1 \\ & \quad \vee (k-1) = 0) \wedge T(e_l) = T_2 \\ & \quad \wedge (T(e_{l-1}) \neq T_2 \vee (l-1) = 0) \\ \mathit{false} & \text{otherwise} \end{cases}$$

For brevity we may write simply $\Phi[...]$ for $\Phi[...] = \mathit{true}$.

A process p_i delivers events in response to its subscription $\Psi(p_i)$ through DELIVER. We consider this primitive to be generically typed, i.e., we write DELIVER$_\Phi([e_1, ..., e_n])$ to deliver a relation $[e_1, ..., e_n]$, where e_j is of type T_j such that $\mathbb{T}(\Phi)=[T_1, ..., T_n]$. DELIVER$^i_\Phi([e_1, ..., e_n])_t$ denotes a delivery on process p_i in response to Φ at time t, and MULTICAST$^i(e)_t$ defines the multicast of an event e by p_i at time t. Again i, t, etc. may be omitted when not germane to the context.

4.3 Properties

We now present properties for composite events in FAIDECS defined over primitives MULTICAST and DELIVER. From here on, *deliver* refers to DELIVER (vs. *TO-deliver* for TO-DELIVER), and *multicast* refers to MULTICAST (vs. *TO-broadcast*). See [24] for detailed discussions of alternative properties.

Basic safety properties. The basic safety properties for FAIDECS are MDM-NO DUPLICATION, MDM-NO CREATION and ADMISSION as shown below:

MDM-NO DUPLICATION: $\exists \text{DELIVER}^i_\Phi([..., e, ...])_t \Rightarrow \nexists \text{DELIVER}^i_\Phi([..., e, ...])_{t'} \mid t' \neq t$

MDM-NO CREATION: $\exists \text{DELIVER}_\Phi([..., e, ...])_t \Rightarrow \exists \text{MULTICAST}(e)_{t'} \mid t' < t$

ADMISSION: $\exists \text{DELIVER}^i_\Phi([e_1, ..., e_n]) \mid \mathbb{T}(\Phi) = [T_1, ..., T_n] \Rightarrow \Phi \in \Psi(p_i) \wedge \Phi[e_1, ..., e_n] \wedge \forall k \in [1..n] : T(e_k) = T_k$

The MDM-NO DUPLICATION property implies that a same event is delivered at most once for a given conjunction, which may be opposed to certain systems that allow a same event to be correlated multiple times. Our property could easily be substituted to allow a delivery for *every instance* of a type in a given conjunction. We omit this for simplicity of the presented properties and algorithms. MDM-NO CREATION is similar to TO-broadcast specifications [14] in that an event

may only be delivered if multicast. ADMISSION ensures type safety and that all events in a relation match the subscription.

Liveness. ADMISSION can trivially hold while not delivering anything. We have to be careful about providing strong delivery properties on *individually* multicast events though, as events may depend on others to match a given conjunction. Nonetheless, we want to rule out bogus implementations which simply discard all events. We thus propose the following complementary liveness properties:

CONJUNCTION VALIDITY: $\exists \text{MULTICAST}(e_l^k), k \in [1..n], l \in [1..\infty] \ \wedge \ p_i \in \mathit{correct}(F) \ \wedge \ \exists \Phi \in \Psi(p_i) \mid \Phi[e_l^1, ..., e_l^n] \ \Rightarrow \exists \text{DELIVER}_\Phi^i([...])_{t_j} \mid j \in [1..\infty]$

EVENT VALIDITY: $\exists \text{MULTICAST}^i(e^x), \text{MULTICAST}^{k,l}(e_l^k), k \in [1..n] \backslash x, \ l \in [1..\infty] \ \{p_i, p_j, p_{k,l}\} \subseteq \mathit{correct}(F) \mid \Phi \in \Psi(p_j) \wedge \mathbb{T}(\Phi) = [T_1, ..., T_n] \ \wedge \ \forall z \in [w..y], T_z = T(e^x) \wedge \nexists (T(e^x)[x-w+1].a_1 \ op \ T[r].a_2) \in \Phi \mid (T \neq T(e^x) \vee r \neq x-w+1) \wedge \Phi[e_l^1, ..., e_l^{x-1}, e^x, e_l^{x+1}, ..., e_l^n] \Rightarrow \exists \text{DELIVER}_\Phi^j([..., e^x, ...])$

These two properties handle the two possible cases that can arise. The first property deals with dependencies across events and can be paraphrased as follows: "If for a correct process p_i, there is an infinite number of relations of matching events that are successfully multicast, then p_i will deliver infinitely many such relations." This property is reminiscent of the FINITE LOSSES property of fair-lossy channels [3]. It allows matching algorithms to discard *some* events for practical purposes such as agreement and ordering, yet ensures that when matching events are continuously multicast, a corresponding process will continuously deliver. From the example presented in Section 4.1, as long as events of both types are inifinitely published such that infinitely often, three successive, increasing stock quotes are multicast after an earnings report, there will be an infinite number of delivered relations.

EVENT VALIDITY provides a property analogous to validity for single-message deliveries (e.g., TOBcast): If an event is multicast by a correct process p_i, and its delivery in response to a conjunction on some correct process p_j is *not* conditioned by binary predicates with other event types, then the event must be delivered by p_j if matching events of all other types are continuously multicast. This latter condition is necessary because the delivery of the event, even in the absence of binary predicates, requires the *existence* of other events (by nature of correlation). The condition also ensures that any unary predicates on the respective event type are satisfied. Note that in the case of multiple instances of that type, for each of which there are only unary predicates that match, the property does not force an event to be delivered more than once as the position of the event is not fixed in the implied delivery. The example in Section 4.1 does not present a unary predicate, and thus would not be affected by this property. If the subscription Ψ_S were extended to trigger only if the value of the U.S. dollar is below some value v as in $\Psi'_S = \Psi_S \ \wedge$ `USDollar.value < v`, then any event matching this predicate will be delivered with the entire relation given by Ψ_S.

Note also that none of these properties is impacted by the presence of multiple instances of a same type in a conjunction. An infinite flow of events of some type implies a multiple (a finite number) of infinite flows of that type.

Agreement. The properties so far ensure that as long as matching events are being multicast, processes will eventually deliver relations. We are, however, interested in stronger properties for these delivered relations, which ensure fairness for relations delivered across processes. We define COVERING AGREEMENT:

$$\text{COVERING AGREEMENT: } \exists \text{DELIVER}^{i}_{\Phi\wedge\Phi'}([e_1,...,e_n,...]) \mid ((\mathbb{T}(\Phi)=[T_1,...,T_n])\cap\mathbb{T}(\Phi'))=\emptyset \Rightarrow \forall p_j \in correct(F)\backslash\{p_i\} \mid \Phi \in \Psi(p_j) : \exists \text{DELIVER}^{j}_{\Phi}([e_1,...,e_n])$$

Subsumption only allows "extending conjunctions to the right" as determinism requires some given order for matching. Intuitively, subsumption in the presence of binary predicates is limited since, when comparing two subscriptions with same types, an event of a first type might match both subscriptions without implying that the same holds for a second event.

Note that COVERING AGREEMENT is not defined in a symmetric way (with $\Phi \wedge \Phi'' \in \Psi(p_j)$), as the presence of a matching set of events for a conjunction Φ' does not imply a timely or even eventual occurrence of a matching set for another sub-relation Φ'' conjoined by p_j with Φ.

Thus, the example subscriptions Ψ_S, as defined in Section 4.1, and Ψ'_S, defined in 4.3, would exhibit the necessary conditions for COVERING AGREEMENT. That is, the common predicates over the `EarningsReport` and `StockQuote` types would yield the same (sub)-relations for Ψ_S and Ψ'_S, where Ψ'_S would deliver relations containing the above with an additional event of type `USDollar`.

4.4 Total Order

Intuitively, and as we will illustrate in the following sections, a total order on individual events can be used to achieve agreement on relations. In fact, it is necessary to do so (see [24] for a formal proof). On the upside, this can be exploited to provide corresponding relation-level properties. We define three types of total order properties below:

$$\text{EVENT TOTAL ORDER: } \exists \text{DELIVER}^{i}_{\Phi}([...,e,...])_{t_i}, \text{DELIVER}^{i}_{\Phi}([...,e',...])_{t'_i}, \text{DELIVER}^{j}_{\Phi'}([...,e,...])_{t_j}, \text{DELIVER}^{j}_{\Phi'}([...,e',...])_{t'_j} \mid T(e)=T(e') \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$$

$$\text{CONJUNCTION TOTAL ORDER: } \exists \text{DELIVER}^{i}_{\Phi\wedge\Phi'}([e_1,...,e_n,...])_{t_i}, \text{DELIVER}^{i}_{\Phi\wedge\Phi'}([e'_1,...,e'_n,...])_{t'_i}, \text{DELIVER}^{j}_{\Phi\wedge\Phi''}([e_1,...,e_n,...])_{t_j}, \text{DELIVER}^{j}_{\Phi\wedge\Phi''}([e'_1,...,e'_n,...])_{t'_j} \mid ((\mathbb{T}(\Phi)=[T_1,...,T_n])\cap\mathbb{T}(\Phi'))=\emptyset \wedge (\mathbb{T}(\Phi)\cap\mathbb{T}(\Phi''))=\emptyset \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$$

$$\text{DISJUNCTION TOTAL ORDER: } \exists \text{DELIVER}^{i}_{\Phi}([e_1,...,e_n])_{t_i}, \text{DELIVER}^{i}_{\Phi'}([e'_1,...,e'_m])_{t'_i}, \text{DELIVER}^{j}_{\Phi}([e_1,...,e_n])_{t_j}, \text{DELIVER}^{j}_{\Phi'}([e'_1,...,e'_m])_{t'_j} \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$$

None of the properties includes any of the others. EVENT TOTAL ORDER ensures that there is a total (sub-)order on the events of a same type. CONJUNCTION TOTAL ORDER ensures that (sub-)*relations* delivered to identical (sub-)conjunctions are delivered in a total order. An implementation which *never* enforces CONJUNCTION TOTAL ORDER, i.e., delivers no two same relations on two processes with identical (sub-)conjunctions, could still ensure EVENT TOTAL ORDER. Perhaps more obvious is that, inversely, EVENT TOTAL ORDER does not imply

Conjunction Total Order. Disjunction Total Order further sets our model apart from many single-event delivery multicast settings (e.g., traditional publish/subscribe), where subscriptions are conjunctions, and disjunctions are viewed as being expressed independently through multiple conjunctions. Our property strives for total order *across* relations delivered to *distinct* conjunctions in a *same* disjunction.

5 Algorithms

We now present ways to implement the properties proposed in the previous section. For illustration purposes, we first outline an approach relying straightforwardly on a total order across multicast events of *all* types. Then, we present novel decentralized algorithms achieving the same properties, leveraging our notion of subscription subsumption.

5.1 Total Order Broadcast Black Box

A straightforward solution for deterministic event correlation across all processes is to rely on a Total Order Broadcast "black box," with primitives to-broadcast and to-deliver for individual events, ensuring that all correct processes eventually TO-deliver all TO-broadcast events in the same order. To multicast an event e of any type, a process simply performs to-broadcast(e); a to-deliver(e) is handled in a deterministic manner described shortly. Many implementations exist, tolerating different failure patterns [7].

Conjunctions. For simplicity, we first focus on single conjunctions for the algorithm in Figure 1 before expounding on generic disjunctions. That is, subscription Ψ_i of process p_i consists in a single conjunction Φ_i. Disjunction Total Order, in this case, becomes subsumed by Conjunction Total Order.

The algorithm in Figure 1 uses *first received* matching semantics and *prefix+infix* disposal. In short, the former means that events are matched on a process in the order received by that process. The latter implies the following: Upon a successful match $[e_1, \ldots, e_n]$, for each event e_i, all events of the same type received prior to e_i are discarded via the garbage collection mechanism dequeue. These semantics are further elaborated on below.

Each process p_i maintains one queue Q per event type in its conjunction $\Phi{=}\Psi(p_i)$. For example, for a conjunction $\Phi = \rho_1 \wedge \rho_2$ where $\rho_1 = T_1.a_1 < T_2.a_2$ and $\rho_2 = T_1.a_1 < 20$, the subscriber maintains one queue for events of type T_1 and one for events of type T_2. When TO-delivering an event, p_i will loop *once* by line 20 and first checks whether the type of the event is in p_i's subscription. If so, p_i attempts to enqueue the event. $Q[T(e)] \oplus e$ denotes the appending of event e to the queue of type $T(e)$. The enqueue primitive returns *true* if the event has been enqueued, which means that it satisfies all unary predicates on the respective types in the conjunction. Then p_i proceeds to matching. Any single received event may complete up to one relation. If a match $[e_1, \ldots, e_n]$

Executed by every process p_i

```
1: init
2:     Ψ ← Φ_1 ∨ ... ∨ Φ_o
3:     Φ_l ← ρ_1 ∧ ... ∧ ρ_m
4:     Q_l[T] ← ∅
5: To MULTICAST(e):
6:     TO-BROADCAST(e)
7: function MATCH ([e'_1, ..., e'_n], Φ, Q)
8:     T ← T_{n+1} | T(Φ) = [T_1, ..., T_{n+1}, ...]
9:     l ← max(j | Q[T] = e_1 ⊕ ... ⊕ e_j ⊕ ... ⊕ e_h) |
         ∃k ∈ [1..n] : e_j = e'_k
10:    for all k = (l + 1)..h do
11:        if |T(Φ)| = n + 1 then
12:            if Φ[e'_1, ..., e'_n, e_k] then
13:                return [e'_1, ..., e'_n, e_k]
14:        else
15:            E ← MATCH([e'_1, ..., e'_n, e_k], Φ, Q)
16:            if E ≠ ∅ then
17:                return E
18:    return ∅
19: upon TO-DELIVER(e) do
20:    for all Φ_l ∈ Ψ | T(e) ∈ T(Φ_l) in order do
21:        if ENQUEUE(e, Φ_l, Q_l) then
22:            [e_1, ..., e_k] ← MATCH(∅, Φ_l, Q_l)
23:            if k ≠ 0 then
24:                DEQUEUE([e_1, ..., e_k], Q_l)
25:                DELIVER_{Φ_l}([e_1, ..., e_k])
26: function ENQUEUE (e, Φ, Q)
27:    win ← max(j | ∃...T(e)[j].a... ∈ Φ)
28:    if ∀j = 1..win ((∃ρ = (T(e)[j].a op v) ∈
         Φ | ¬ρ[e]) ∨ (∃(ρ = T(e)[j].a op
         T(e)[j].a') ∈ Φ | ¬ρ[e])) then
29:        return false
30:    else
31:        Q[T(e)] ← Q[T(e)] ⊕ e
32:        return true
33: procedure DEQUEUE([e_1, ..., e_m], Q)
34:    for all Q[T] = ... ⊕ e_k ⊕ e ⊕ ..., k ∈ [1..m]
       do
35:        Q[T] ← e ⊕ ...
```

Fig. 1. Conjunctions/disjunctions with Total Order Broadcast

is identified, the corresponding events are discarded (DEQUEUE) and for each event e_i, all preceding events of the same type are discarded from the respective queue for that type. MATCH iterates through the queues deterministically. The semantics attempt to find the *first* instance of the first type in Φ for which there are events of the remaining types with which Φ is satisfied. Among all such possibilities, the algorithm recursively seeks for a match with the *first* instance of the second type in Φ, etc. until a match is found or all possibilities are exhausted. For multiple instances of a same type, a first instance is recursively matched with the *first follow-up instance* in the same queue until the needed number of instances is found for that type or the queue is exhausted.

Assuming that the underlying TOBcast primitive ensures TOB-NO CREATION and TOB-NO DUPLICATION (see Section 3), it is easy to see how the algorithm of Figure 1 ensures the corresponding MDM-NO CREATION and MDM-NO DUPLICATION properties defined in Section 4.3. An event e, matching all unary predicates of a conjunction Φ, is successfully added to the corresponding queue $Q[T(e)]$ in ENQUEUE (line 31, Figure 1). The only way in which e can be removed (and delivered) is together with a matching set of other events fulfilling Φ (line 23, Figure 1), thus ensuring ADMISSION. If matching sets of such events are continuously TO-broadcast, then a match will eventually be determined at line 12 thus ensuring EVENT VALIDITY. CONJUNCTION VALIDITY holds by a similar line of reasoning. The first matching, together with prefix+infix disposal, and the independent handling of events of distinct types ensures EVENT TOTAL ORDER. If two processes p_i and p_j define conjunctions $\Phi \wedge \Phi'$ and Φ respectively, as long as Φ and Φ' are type-disjoint, then events that match with Φ are independent of any events that match with Φ'. Thus, if there is a matching relation for p_i, there is a subset of the relation for which Φ is true. Since garbage collection is deterministic and is triggered every time an event of a type in $\mathbb{T}(\Phi)$ is TO-delivered and in the same order on p_i and p_j with respect to those

deliveries, p_i and p_j will handle respective events identically, ensuring COVERING AGREEMENT. Similarly, CONJUNCTION TOTAL ORDER holds as all processes TO-deliver all relevant events. When p_i identifies a match for $\Phi \wedge \Phi'$, with Φ and Φ' type-disjoint, p_j will have TO-delivered the respective subset of events in Φ already in the same sub-order and thus DELIVERs the respective sub-relations in the same order with any events identified for a Φ'' type-disjoint with Φ.

Disjunctions. When the subscription is a disjunction of several conjunctions, a process maintains one event queue per event type *per* conjunction. For example, for a disjunction $\Psi = \Phi_1 \vee \Phi_2$ where $\mathbb{T}(\Phi_1)=\mathbb{T}(\Phi_2)=[T_1, T_2]$, a process maintains two queues for type T_1 and then two queues for type T_2, one each for Φ_1 ($Q_1[T_1]$ and $Q_1[T_2]$) and for Φ_2 ($Q_2[T_1]$ and $Q_2[T_2]$).

Figure 1 supports multiple conjunctions in a single disjunction. The primary distinction is in the response to TO-deliveries. The primitive dispatches events to conjunctions *in order* of subscriptions. In contrast to subscriptions of one conjunction, an event can lead to multiple MATCHes and DELIVERies.

Because the MATCHing is performed deterministically, as explained previously for a given conjunction, and all processes ENQUEUE the same sets of events in the same order, COVERING AGREEMENT across any two conjunctions is met for the same reasons as for single conjunctions. This property would also be met by any unordered dispatching for multiple conjunctions. The other properties established for conjunctions remain valid due to the duplication of events appearing in distinct conjunctions of a same subscription.

DISJUNCTION TOTAL ORDER is met as any p_i and p_j defining two identical separate conjunctions TO-deliver the respective events (possibly interleaved by those for other conjunctions in $\Psi(p_i)$ and $\Psi(p_j)$ respectively) in the same order. Thus, the correlation for respective relations occurs in the same order.

A simple optimization of the algorithm for subscriptions containing several conjunctions Φ_1,...,Φ_m with a common event type T, omitted for brevity, consists in sharing the queue for T across conjunctions. An event in a queue is then tagged by the index k of a conjunction Φ_k to indicate that the event has previously been used in a match and DELIVERed for Φ_k. Earlier events of that type should then also be tagged with k. Events with tags $\{1, ..., m\}$ may then be discarded. Also, the portrayed matching algorithm performs an exhaustive search and is thus not efficient; however, it suffices to illustrate the relevant properties and can be represented concisely. More elaborate and efficient matching algorithms exist, which offer the same semantics. A common approach consists in storing *partial matches* in specialized data-structures to avoid matching a given event multiple times with same events (cf. [9]). In our implementation of FAIDECS and all evaluated algorithms, we make use of the Rete [10] matching algorithm.

5.2 FAIDECS Decentralized Ordered Merging

One of the simplest and most popular approaches in practice for Total Order Broadcast consists in a sequencer, which orders *all* events. As long as the sequencer remains available (e.g., through replication), the properties presented

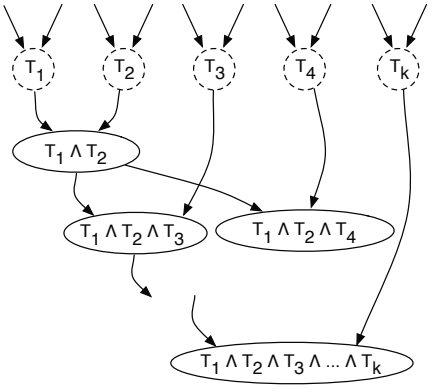


Fig. 2. $T_1 \wedge ... \wedge T_j$ denotes the *conjunction* merger for the respective types $\sqcup[T_1, ...T_j]$ (single instance per type)

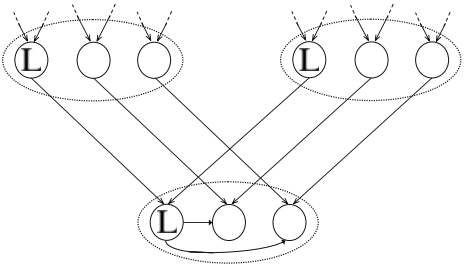


Fig. 3. Small-scale FAIDECS merger replication. Dotted ovals are "logical" mergers; circles are processes. L denotes the leader.

earlier hold under respective assumptions on failure patterns. A Consensus-based textbook Total Order Broadcast [14] yields the same properties with much better fault tolerance (typically a minority of all processes may fail), yet with a higher overhead. We now present a decentralized solution implementing the same properties, yet with much better scalability characteristics than both and inherently better fault-tolerance than a sequencer-based approach. The solution assumes a distributed hashtable (DHT) or similar mechanism for uniquely identifying a process for a given "role." Lightweight replication mechanisms used for fault-tolerance of such roles are discussed separately thereafter.

Conjunctions. We first describe an algorithm focusing on single conjunctions, providing the same properties as that of Figure 1. All processes with conjunctions on a sequence of event types $[T_1, ..., T_k]$ send their subscriptions to a same process, identified as p_j=PROCESS($\sqcup[T_1, ..., T_k]$), responsible for handling all conjunctions on the involved sequence of types *without duplicates*[2]:

$$\sqcup[T_1, ..., T_1, T_2, ...] = [T_1] \oplus \sqcup[T_2, ...]$$

The function PROCESS relies on a DHT (e.g., a deterministic lookup facility) to deterministically identify such responsible processes, called *mergers*. Lodged at the root of the thereby created overlay network (see Figure 2) are mergers responsible for individual event types T_1, T_2, etc. To ensure the properties with respect to extensions of conjunctions to the right, events undergo an *ordered merge by type* where a merger p_j=PROCESS($\sqcup[T_1, ..., T_k]$) gets events of types T_1, ..., T_k from two processes: those identified as PROCESS($\sqcup[T_1, ..., T_{k-1}]$) and PROCESS($[T_k]$). We term processes in the *role* of subscribers/publishers as *clients*.

Figure 4 presents the algorithm for merging event types and handling subscriptions corresponding to the merged types. Figure 5 presents the algorithm

[2] We could use different mergers but deduplication simplifies the algorithm.

Executed by every process p_i=PROCESS($\sqcup[T_1, ..., T_k]$)

```
1:  init
2:      left ← PROCESS([⊔T_1, ..., T_{k−1}])
3:      right ← PROCESS([T_k])
4:      subs[p_j]
5:      kids[p_j]
6:      INITPARENTS()
7:  procedure INITPARENTS()
8:      Ψ' ← ⋁_{Ψ∈kids∪subs} Ψ \
            {ρ ∈ Ψ | 𝕋(ρ) ∉ {[T_1], ..., [T_{k−1}]}}
9:      SEND(CON, Ψ') to left
10:     Ψ'' ← ⋁_{Ψ∈kids∪subs} Ψ \
            {ρ ∈ Ψ | 𝕋(ρ) ≠ [T_k]}
11:     SEND(CON, Ψ'') to right
12: upon RECEIVE(CON, Ψ) from p_j do
13:     kids[p_j] ← Ψ
14:     INITPARENTS()
15: upon RECEIVE(SUB, Φ) from p_j do
16:     subs[p_j] ← Φ \ {ρ ∈ Φ | |𝕋(ρ)| > 1}
17:     INITPARENTS()
18: upon RECEIVE(EV, e) do
19:     for all Ψ = kids[p_j] do
20:         if ∃l, Φ ∈ Ψ | ∀ρ = T(e)[l]... ∈ Φ : ρ[e]
            then
21:             SEND(EV, e) to p_j
22:     for all Φ = subs[p_j] do
23:         if ∃l | ∀ρ = T(e)[l]... ∈ Φ : ρ[e] then
24:             SEND(EV, e) to p_j
```

Fig. 4. Ordered merging for conjunctions: mergers

Executed by every p_i. Reuses ENQUEUE, MATCH, DEQUEUE of Figure 1

```
1:  init
2:      Ψ ← Φ
3:      Φ ← ρ_1 ∧ ... ∧ ρ_m
4:      Q[T] ← ∅
5:      SEND(SUB, Φ) to PROCESS(⊔𝕋(Φ))
6:  To MULTICAST(e):
7:      SEND(EV, e) to PROCESS([T(e)])
8:  upon RECEIVE(EV, e) do
9:      if ENQUEUE(e, Φ, Q) then
10:         [e_1, ..., e_l] ← MATCH(∅, Φ, Q)
11:         if l > 0 then
12:             DEQUEUE([e_1, ..., e_l], Q)
13:             DELIVER_Φ([e_1, ..., e_l])
```

Fig. 5. Ordered merging for conjunctions: clients

for client processes. Unary predicates are propagated from subscribers to mergers (line 16, Figure 4), and from mergers to their ancestor mergers in the form of disjunctions (lines 8-11) since a potential match (i.e., compliant with any unary predicates) for *any* merger or subscriber means a potential match for a parent merger. Forwarding of events received by mergers from their respective parent mergers ($left$) or processes for merged event types ($right$) happens without interruptions by other events and can be achieved by simple local synchronization.

For simplicity, the algorithm in Figure 5 handles event queues at clients. The use of shared queues on mergers as described at the end of Section 5.1, could lead to savings in global memory overhead by avoiding redundancies. In practice, we have observed that this, however, overburdens mergers, just like a propagation of complete conjunctions instead of only unary predicates to mergers.

Assuming that all subscribers are connected to mergers which are connected to each other before events are multicast, the properties described in Section 4.3 are also met by the algorithm in Figures 4 and 5 thanks to the type-ordered merging of events. COVERING AGREEMENT and CONJUNCTION TOTAL ORDER are ensured as processes with a common "prefix" in their conjunctions, which is type-disjoint with any conjoined predicates, will receive the same events for the prefix and in the same order from the corresponding conjunction merger process.

Disjunctions. For disjunctions, we essentially need to solve Total Order *Multicast* [12] on the event sequences output by conjunction mergers. Using timestamps and extending the conjunction algorithm of Figures 4 and 5, order of events is established for clients as needed for disjunctions. More precisely, conjunction mergers following the algorithm of Figure 6 timestamps all received

Executed by every process p_i=PROCESS($\sqcup[T_1, ..., T_k]$). Reuses lines 1-11 of Figure 4

```
18: upon RECEIVE(EV, e)                {Rplcs lines 18-24 }
19:     for all Ψ = kids[p_j] do
20:         if ∃l, Φ ∈ Ψ | ∀ρ = T(e)[l]... ∈ Φ : ρ[e]
            then
21:             SEND(EV, e) to p_j              {end for}
22:     time ← current time {cont frm Line 21}
23:     for all  Φ = subs[p_j] do
24:         if  ∃l | ∀ρ = T(e)[l]... ∈ Φ : ρ[e] then
25:             SEND(EV, e, time) to p_j
```

Fig. 6. Disjunction-enabled ordered merging for conjunctions: mergers

Executed by every p_i. Reuses ENQUEUE, MATCH, DEQUEUE of Figure 1

```
1: init
2:     Ψ ← Φ_1 ∨ ... ∨ Φ_o
3:     Φ_l ← ρ_1 ∧ ... ∧ ρ_m
4:     Q_l[T] ← ∅
5:     R ← ∅
6:     S[T] ← 0
7:     for all Φ_l ∈ Ψ do
8:         SEND(SUB, Φ_l) to PROCESS(⊔T(Φ_l))
9: To MULTICAST(e):
10:    SEND(EV, e) to PROCESS([T(e)])
11: upon RECEIVE(EV, e, ts) do
12:    if ts > S[T(e)] then
13:        S[T(e)] ← ts
14:        R' ← {⟨e', t'⟩ ∈ R | t' < ts}
15:        R'' ← {⟨e', t'⟩ ∈ R | t' > ts}
16:        R ← R' ∪ {⟨e, ts⟩} ∪ R''
17:        for all ⟨e', t'⟩ ∈ R ordered on t' |
             t' < MIN_T(S[T]) do
18:            for all Φ_l in order do
19:                if ENQUEUE(e', Φ_l, Q_l) then
20:                    R ← R\{⟨e', t'⟩}
21:                    [e_1, ..., e_k] ← MATCH(∅, Φ_l, Q_l)
22:                    if k > 0 then
23:                        DEQUEUE([e_1, ..., e_k], Q_l)
24:                        DELIVER_Φl([e_1, ..., e_k])
```

Fig. 7. Ordered merging for conjunctions and disjunctions: clients

messages before passing them to clients which do the actual correlation (Figure 7). There is no need for specialized disjunction mergers, which are thus omitted here for simplicity. (If using dedicated disjunction mergers, these can be arbitrarily connected among each other to cover the respective conjunctions.)

If processes send timestamps with events, to achieve order of DELIVERy for relations, an event is only ENQUEUEd (and correspondingly MATCHed) when a receiving process has received events for all other types in its subscription, and the timestamp of that event is less than all the other respective timestamps of other types. As long as all processes which are MULTICASTing events of the respective types continue to do so, for any receiving process, an event will eventually be ENQUEUEd after other events with lower timestamps of other types. This guarantees that all processes receiving the same events over a set of types will ENQUEUE and thus perform a MATCH on them one by one in the same order.

If there are any processes which multicast events at a slower rate than others, then the approach may not be as efficient with the requirement that each event of a type (before being ENQUEUEd) must wait for events of every other type with higher timestamps to be received. To solve this problem for the algorithm in Figure 7, if an event has not been received in some time interval by a *conjunction* merging process, then an "empty" event $e_\perp$ may be sent to all processes in $subs[p_j]$, indicating that pending events of other types may be respectively ENQUEUEd. Depending on the targeted scenarios (e.g., publication rate, topology) other information such as rates may be used (additionally).

MDM-No CREATION and MDM-No DUPLICATION are met as ENQUEUE and MATCH are only performed on received events, and for a given type, only events with a higher timestamp than the last event of that type are further

added to the ordered set R and queue Q_l. Since an event is never ENQUEUEd unless its type exists in the process's subscription, and MATCH is performed over every received event, ADMISSION holds. As in Section 5.2, EVENT VALIDITY and CONJUNCTION VALIDITY are retained here despite the filtering and discarding of certain events. It is easy to see that the timestamps generated by mergers follow the observed order of event reception, thus respecting CONJUNCTION TOTAL ORDER. Given that events are compared based on timestamps and merged in order of conjunctions, DISJUNCTION TOTAL ORDER is also ensured.

Joining. The algorithms presented so far *all* rely on a consistent set of event queues across all processes with the same composite subscription if any subscription is issued prior to publications. However, this consistency is violated when two such related processes subscribe to an event stream at different times with respect to the multicasting of events. In order to maintain consistency, we thus employ a simple synchronization algorithm between (a) a joining subscriber process, (b) the corresponding conjunction merger(s), and (c) one of the *existing* subscriber processes with identical conjunctions, if any. This ensures that a joining process starts with a valid state of the respective queues copied from any existing subscriber and does not miss any subsequent events from the merger received also by that existing subscriber after copying the state of its queues.

Fault tolerance. For presentation simplicity, the algorithms described thus far stipulated single processes returned by function PROCESS() as responsible for given conjunctions, which obviously provides little fault tolerance. In FAIDECS, PROCESS() returns a small fixed number of processes; i.e., the underlying DHT determines a set of replicas for such merger roles. A membership layer monitors the merger processes and ensures that their membership is consistent. Figure 3 provides an overview of the replication. A role, or "logical" merger process, is represented by 3 replicas which are contoured by a dotted line. L represents a *leader* process which determines the order between the merged types and communicates that *order* (only) to its peers. These receive the *actual events* independently as depicted in the figure. When a physical merger process (solid circles) p_i fails, its descendant(s) connect to one of p_i's peers. To ensure that no events are missed in the meantime, all replicas regularly acknowledge received and forwarded events to each other; events prior to such acknowledgements are buffered. If a process lags or fails, its peers will attempt to replace it. Using majority-based voting, a minority of (suspected) process failures can typically be tolerated at a time. In addition to benefitting fault tolerance, this small-scale replication also benefits load distribution, in that down-stream processes, including subscribers, distribute uniformly over the replicas.

6 Evaluation

To demonstrate the scalability of our decentralized algorithms and explore overall performance benefits and tradeoffs, we compare a Java implementation of

FAIDECS to the algorithm of Figure 1 with 3 different JGroups-based[3] implementations for the Total Order Broadcast black box: (1) a sequencer algorithm, (2) a replicated sequencer (3 replicas) and (3) a token-based algorithm. Figure 10 summarizes our findings. An extended version of this report [25] presents further descriptions and results.

6.1 Metrics and Experimental Setup

We used two metrics – *Throughput:* the average number of events delivered per second by a subscriber, and *Latency:* the average delay between the multicasting time of an event and its delivery to a subscriber. The number of subscribers was increased from 10 to 600, and each subscriber had a randomly generated set of subscriptions. Each event consisted of 3 integer attributes with values chosen uniformly at random within [0..1000]. All processes were run on 65 nodes in a LAN. Each node is equipped with an Intel Xeon 3.2GHz dual-core processor and 2GB RAM, and runs Linux. A maximum of 15 subscriber processes were run on a single node. The maximum multicast rates varied by setup (e.g., different components became the bottleneck, selectivity of subscriptions varied). We tested scalability of FAIDECS first in terms of conjunctions and then disjunctions.

For conjunctions, we used 3 different distributions of subscriptions, which led to different workloads for actual routing and filtering of events. In scenarios A and B, we followed the setup of Figure 8, increasing the maximum number of conjoined types (and thus the depth) k from 2 to 4. For scenario A, *all* filtering occurred at end nodes rather than in mergers through the selectivity of binary predicates, which differed across conjunctions to achieve the same expected delivery rates at all subscribers in a respective level. This scenario demonstrated the limits of the overlay. In scenario B, events were filtered at the mergers through unary predicates propagated upwards from subscriptions, allowing higher aggregate multicast rates than in scenario A. Scenario C invariably had 4 event types, and subscriptions were over all 6 possible conjunctions ($\binom{4}{2}$). This allowed us to explore the potential of traffic separation. For evaluating scalability with respect to disjunctions, we used scenario D, which is the merger overlay shown in Figure 9. The maximum level was also varied (from 2 to 4). Subscribers were uniformly distributed across all merger processes and throughput/latency values were averaged for each group of subscribers for a given level.

We expect that the bottleneck in our decentralized algorithms would occur at the merger process(es) which would merge all involved types, limiting throughput consistently for all k. All values are normalized with respect to the values obtained with FAIDECS with 10 subscribers connected to a single merger for 2 types in scenario A, and with respect to the relations with the largest number of types (independent of the algorithm). Throughput here was approximately 31,400 events/s and latency 150ms. Normalization does not introduce any bias but makes comparison clear, so that values could be reported independent of subscriptions, and so that values may be reported for each level independently.

[3] http://www.jgroups.org

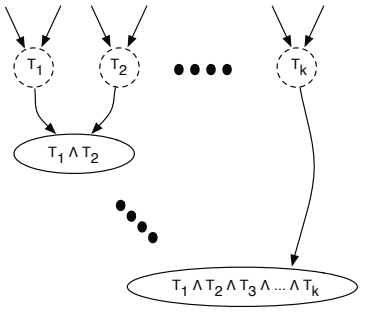


Fig. 8. Setup for conjunctions (scenarios A and B)

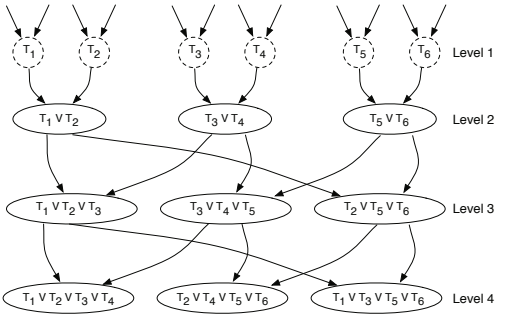


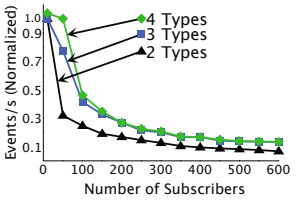
Fig. 9. Setup for disjunctions (scenario D)

6.2 Conjunctions

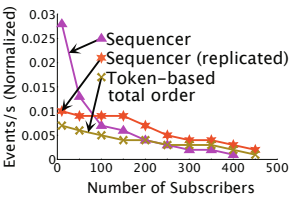
Figure 10(a) displays the trend in throughput as the system scales to more subscribers in scenarios A and B with varying number of event types/levels k (see Figure 8). FAIDECS scales very well compared to the approaches shown in Figure 10(b), shown separately for a clear relationship among the three implementations since the values start at nearly 3% (about 950 events/s) and remained consistent in all scenarios. Note that IP-multicast was turned off in the test environment which could help throughput for both FAIDECS and the JGroup implementations. In Figure 10(b), the token-based algorithm starts with a higher throughput than the sequencer-based one as there were few multicasters competing over the token, but its performance degrades faster due to the inherent cost of its high fault tolerance. Replication helps performance in both FAIDECS and the replicated sequencer due to the load balancing of replicas of a same logical merger process, though less and with an initial cost for the replicated sequencer. The total throughput remained approximately the same in scenarios A and B since propagation of events by mergers was the bottleneck.

Figure 10(c) illustrates the scalability and the high throughput of FAIDECS when subscriber interests are in largely disjoint types, following scenario C. Thus, FAIDECS scales very well with the addition of an arbitrary number of types to a system, even with transitive correlation across them as in scenario C, given enough merger process nodes to support them – the high throughput (about double that of two types for scenario A) occurs because every merger only handles relatively few subscribers compared to the other scenarios.

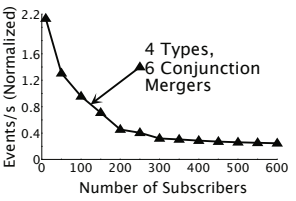
Figure 10(d) reports the latency of our algorithms for scenario A. As expected, increased depth (conjunctions with increasing number of types) leads to increased latency. Here the "depth" k is fixed to 4, but latency is reported independently at different depths. The observed latency, averaged over all subscribers within each level, was approximately the same with replicated and non-replicated mergers.



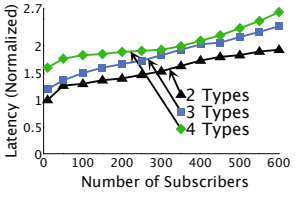
(a) Scenario A/B throughput for conjunctions.



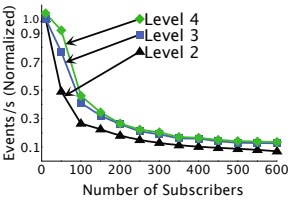
(b) Other total order implementations.



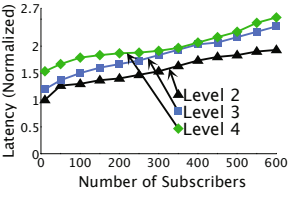
(c) Scenario C throughput for conjunctions.



(d) Scenario A latency for conjunctions.



(e) Scenario D throughput for disjunctions.



(f) Scenario D latency for disjunctions.

Fig. 10. Comparing conjunction/disjunction algorithms to a sequencer based approach

6.3 Disjunctions

Figure 10(e) compares the scalability of FAIDECS with respect to throughput in scenario D. The 3 curves represent different depths of the hierarchy (between 2 to 4 levels). For each curve, the throughput is averaged at the respective level. We observe that the impact on throughput is minimal when the disjunctions are made more complex. As shown in Figure 10(f), the latency for 4 types improves slightly. This is because disjunctions provide more than one possibility for event delivery, and the system is no longer throttled by the rate of the slowest upstream process as with conjunctions.

7 Conclusions

We have presented decentralized algorithms for event correlation implemented in FAIDECS. Our algorithms provide clear properties, hinging on a novel notion of subscription subsumption tailored to correlation. The same properties can be achieved by less specialized solutions such as sequencer-based schemes, yet our solutions are inherently more scalable and reliable, leading to strong properties with practical performance; our solutions are also more scalable than peer-based approaches, e.g., relying on tokens, while still achieving practical fault-tolerance. We are currently exploring extensions of our algorithms and additional properties (e.g., causal order).

References

1. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: A New Model and Architecture for Data Stream Management. VLDB Journal (2003)
2. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., Chandra, T.D.: Matching Events in a Content-Based Subscription System. In: PODC (1999)
3. Basu, A., Charron-Bost, B., Toueg, S.: Simulating Reliable Links with Unreliable Links in the Presence of Failures. In: Babaoğlu, Ö., Marzullo, K. (eds.) WDAG 1996. LNCS, vol. 1151, pp. 105–122. Springer, Heidelberg (1996)
4. Birman, K.P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., Minsky, Y.: Bimodal Multicast. In: ACM TOCS (1999)
5. Carzaniga, A., Rosenblum, D., Wolf, A.: Design and Evaluation of a Wide Area Event Notification Service. In: ACM TOCS (2001)
6. Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.-K.: Composite Events for Active Databases: Semantics, Contexts and Detection. In: VLDB (1994)
7. Défago, X., Schiper, A., Urbán, P.: Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. In: ACM CSUR (2004)
8. Demers, A., Gehrke, J., Hong, M., Riedewald, M., White, W.M.: Towards Expressive Publish/Subscribe Systems. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 627–644. Springer, Heidelberg (2006)
9. Eugster, P., Jayaram, K.R.: EventJava: An Extension of Java for Event Correlation. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 570–594. Springer, Heidelberg (2009)
10. Forgy, C.L.: On the efficient implementation of production systems. PhD thesis, Carnegie Mellon University (1979)
11. Garcia-Molina, H., Spauster, A.: Message Ordering in a Multicast Environment. In: ICDCS (1989)
12. Guerraoui, R., Schiper, A.: Genuine Atomic Multicast in Asynchronous Distributed Systems. In: TCS (2001)
13. Grimm, R., Davis, J., Lemar, E., MacBeth, A., Swanson, S., Anderson, T.E., Bershad, B.N., Borriello, G., Gribble, S.D., Wetherall, D.: System Support for Pervasive Applications. In: ACM TOCS (2004)
14. Hadzilacos, V., Toueg, S.: Fault-Tolerant Broadcasts and Related Problems. In: Distributed Systems, 2nd edn. (1993)
15. Koch, G.G., Koldehofe, B., Rothermel, K.: Cordies: Expressive Event Correlation in Distributed Systems. In: DEBS (2010)
16. Kompella, R.R., Yates, J., Greenberg, A.G., Snoeren, A.C.: IP Fault Localization Via Risk Modeling. In: NSDI (2005)
17. Krügel, C., Tóth, T., Kerer, C.: Decentralized Event Correlation for Intrusion Detection. In: Kim, K.-c. (ed.) ICISC 2001. LNCS, vol. 2288, pp. 114–131. Springer, Heidelberg (2002)
18. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. CACM (1978)
19. Li, G., Jacobsen, H.-A.: Composite Subscriptions in Content-Based Publish/Subscribe Systems. In: Alonso, G. (ed.) Middleware 2005. LNCS, vol. 3790, pp. 249–269. Springer, Heidelberg (2005)
20. Pietzuch, P.R., Shand, B., Bacon, J.: A Framework for Event Composition in Distributed Systems. In: Endler, M., Schmidt, D.C. (eds.) Middleware 2003. LNCS, vol. 2672, pp. 62–82. Springer, Heidelberg (2003)

21. Rabinovich, E., Etzion, O., Ruah, S., Archushin, S.: Analyzing the Behavior of Event Processing Applications. In: DEBS (2010)
22. Sánchez, C., Sankaranarayanan, S., Sipma, H.B., Zhang, T., Dill, D.L., Manna, Z.: Event Correlation: Language and Semantics. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 323–339. Springer, Heidelberg (2003)
23. Tatbul, N., Çetintemel, U., Zdonik, S.B.: Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In: VLDB (2007)
24. Wilkin, G.A., Eugster, P.: Multicast with Aggregated Deliveries (2010), http://www.cs.purdue.edu/homes/peugster/MDMcastTR.pdf
25. Wilkin, G.A., Jayaram, K.R., Eugster, P., Khetrapal, A.: Fair Decentralized Event Correlation with FAIDECS (2011), http://www.cs.purdue.edu/homes/peugster/EventJava/FAIDECSTR.pdf
26. Zhao, Y., Strom, R.E.: Exploiting Event Stream Interpretation in Publish-Subscribe Systems. In: PODC (2001)

AmbiStream: A Middleware for Multimedia Streaming on Heterogeneous Mobile Devices

Emil Andriescu[1], Roberto Speicys Cardoso[2], and Valérie Issarny[1]

[1] ARLES Project-Team, INRIA Paris-Rocquencourt,
Domaine de Voluceau, Rocquencourt, Le Chesnay 78153, France
{emil.andriescu,valerie.issarny}@inria.fr
[2] Ambientic
Domaine de Voluceau, Rocquencourt, Le Chesnay 78153, France
roberto.speicys_cardoso@ambientic.com

Abstract. Multimedia streaming when smartphones act as both clients and servers is difficult. Indeed, multimedia streaming protocols and associated data formats supported by today's smartphones are highly heterogeneous. At the same time, multimedia processing is resource consuming while smartphones are resource-constrained devices. To overcome this complexity, we present AmbiStream, a lightweight middleware layer solution, which enables applications that run on smartphones to easily handle multimedia streams. Contrarily to existing multimedia-oriented middleware that propose a complete stack for multimedia streaming, our solution leverages the available highly-optimized multimedia software stack of the smartphones' platforms and complements them with additional, yet resource-efficient, layers to enable interoperability. We introduce the challenges, present our approach and discuss the experimental results obtained when executing AmbiStream on both Android and iOS smartphones. Our results show that it is possible to perform adaptation at run time and still obtain streams with satisfactory quality.

Keywords: multimedia streaming, mobile, smartphone, middleware.

1 Introduction

The present generation of smartphones enables a number of applications that were not supported by previous generation cellular phones. Particularly, the greater processing power, better network connectivity and superior display quality of these devices allow users to consume rich content such as audio and video streams while moving. Not surprisingly, radios[1] and television channels[2] today provide mobile applications that allow access to their live media streams. Even video rental services[3] provide mobile applications that support movie streaming to smartphones.

[1] www.npr.org/services/mobile
[2] www.nasa.gov/connect/apps.html
[3] itunes.apple.com/us/app/netflix/id363590051

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 249–268, 2011.

All those applications, however, assume a centralized architecture where a powerful server (or a farm of servers) provide streams to lightweight mobile devices. Node heterogeneity also remains an issue: most of those applications are available for a single smartphone platform. Indeed, to support multiple phone platforms, developers must (i) modify the mobile application to support different sets of decoders, streaming protocols and data formats and (ii) generate multiple data streams on the server side to be consumed by each mobile platform. Hence, when a resourceful server is not available, as in the case of a smartphone to smartphone streaming scenario, this approach is impractical.

In this paper, we introduce AmbiStream, a middleware-layer solution to enable multimedia streaming among heterogeneous smartphones. Such a solution can be beneficial to a large number of applications. Examples of such applications include:

- Streaming a live event directly to other devices reachable on the network;
- Sharing media on the fly between different devices (phone to tablet/TV);
- Voice call applications;
- Distributed processing of a video stream;
- Mixing augmented reality with live remote user interaction for, e.g., a networked game;
- Multimedia-rich collaboration among mobile users;
- Audio/video sharing in crisis situations when infrastructure is unavailable;
- Private communication of multimedia data between peers (without involving a third party server);

Today, to create such applications, developers must overcome a number of constraints. First, smartphones run different mobile operating systems, each supporting a different set of media encoders, decoders and streaming protocols. Second, communication is performed over wireless networks that are unstable and that do not support resource reservation, and thus streaming quality is managed by the protocol without cooperation from the network layer. Finally, the multimedia streaming software stack of each platform is highly optimized to deliver high quality audio and video while reducing resource usage.

Existing system support for multimedia streaming is unsuitable to face the smartphone challenges described above. Indeed, architectures for multimedia streaming on the Internet such as [10,23] suppose the existence of powerful servers that can adapt content on behalf of clients, which is infeasible when the streaming server is a resource-constrained smartphone. Solutions for multimedia streaming on ad hoc networks either do not consider the problem of content adaptation [2,12,24] or are cross-layered, such as those surveyed in [14]. Cross-layered solutions require cooperation between application layers and networking layers, e.g., integration between the video codec and the routing protocol to optimize streaming quality.

To enable multimedia streaming among heterogeneous devices, two main challenges must be solved. First, multiple incompatible protocols for multimedia streaming exist today, and each platform supports one or a small subset of them. As a result, smartphones must overcome the streaming protocol heterogeneity problem to be able to exchange multimedia streams with heterogeneous

devices. Second, each smartphone platform generates and stores multimedia data using some specific container format, usually depending on the streaming protocols it supports. These data cannot be directly transmitted through a different streaming protocol because the media container format is specific to the protocol. Smartphones, then, must also adapt the media container format to enable translation from the native streaming protocol to non-native protocols supported by other peers.

To address the above challenges, we propose a lightweight middleware layer that complements existing software stack for multimedia streaming on smartphones with components that enable interoperability. The proposed layer defines an intermediate protocol and the associated container format for multimedia streaming among heterogeneous nodes. This layer also mediates the native media container formats and protocols to/from the intermediate streaming protocol.

The remainder of the paper is organized as follows. In the next section we review existing work on multimedia streaming in mobile environments, as well as research related to automated protocol adaptation. In Section 3 we detail the challenges involved in creating a layer to adapt multimedia streams in mobile heterogeneous environments. Section 4 presents the architecture of AmbiStream layer and explains how the main components operate: the format adapter, the protocol translator and the local media server. Section 5 discusses our initial experimental results on Android and iOS devices, which show that it is possible to adapt data and protocols at run time and also obtain streams with satisfactory quality. Finally, in Section 6, we draw our conclusions and discuss future work.

2 Related Work

Many multimedia-oriented middleware have been proposed in the literature. One of the earliest efforts in this direction was proposed in [9], which provided applications with mechanisms for late binding based on QoS constraints. The proposed platform was later extended in [8] to leverage CORBA's mechanisms for inspection and adaptation and enable applications to adapt the stream quality based on information obtained by inspecting middleware components. However, as predicted in [4], the lack of mature multimedia support at the middleware level led the industry to develop platform-specific solutions to handle multimedia streaming quality. As a result, today, most existing streaming protocols integrate mechanisms to adapt video quality to network conditions. Other middleware solutions have been proposed to provide multimedia streaming services. Chameleon [11] is a middleware for multimedia streaming in mobile heterogeneous environments. It is implemented using pure Java Core APIs in order to be portable to all Java and JavaME handsets. In Chameleon, servers send streams with different levels of quality to different multicast groups, so that clients can select the best quality according to their available resources and also adapt to changes on resource availability by selecting a multicast group providing a stream with lower quality. This approach imposes a heavy burden on the server side, which has to keep multiple streams in parallel regardless of the number of

clients. Furthermore, Chameleon implements the whole software stack required for streaming, which has a negative impact on performance.

Fewer works take into account the capabilities of current smartphones and their impact on mobile multimedia streaming. The evaluation of streaming mechanisms in [18] for Android 1.6 and iOS 3.0 tries to identify which design is better suited for mobile devices. Traditional metrics such as bandwidth overhead, start-up delay and packet-loss are used to evaluate the quality of multimedia streaming in various test situations. They observe that high network delays can result in non-continuous playback when using the HTTP Live protocol from iOS, while RTP streaming remains unaffected on Android.

Our approach to solve heterogeneity issues and enable multimedia streaming between heterogeneous mobile devices specifically stems from research on protocol translation and mediation. The work in [21] proposes a framework to formalize the process of synthesizing connectors that mediate two incompatible protocols, and suggests that data mediation can be solved through ontology integration. However, it falls short from addressing the specifics of multimedia streaming protocols, where messages are dependent on time and where message data must also be adapted during mediation.

Nakazawa et. al. [15] propose a taxonomy of bridging solutions among communications middleware platforms and present uMiddle, a system for universal interoperability, which supports mediation (entities and protocols are translated to an intermediate common representation) and is deployed as an infrastructure-provided service. This design choice is appropriate for bridging communications middleware, since it requires communication through different transport technologies that may not be available on all nodes. However, in our scenario, we want to enable peer-to-peer streaming between smartphones without using an untrusted third party server. As such, it is desirable that clients and servers are able to perform mediation independently from the infrastructure.

Another approach for the automatic translation of protocols is z2z [7], which combines a language for specification of protocols and messages, a compiler that automatically generates protocol gateways using C code, and a runtime that executes and manages protocol gateways. Z2z can translate a large number of protocols, but it does not take into account timing requirements typical from real-time streaming protocols. In such protocols, state transitions are not defined by a fixed set of message exchanges but rather by the time deadlines that the protocol must meet. With regard to message contents, z2z protocol gateways can adapt messages by rearranging data from an input message to an output message. This is also not sufficient to overcome the complexity of multimedia streaming protocols, where timing limits may require that messages are processed and regenerated when adapting protocols. Z2z evolved to Starlink [6] which enables protocol translation dynamically at run time, a particularly important feature in systems where existing protocols are unknown at compile time. Our approach adopts an intermediate protocol and requires only clients to adapt their native protocols to the intermediate protocol, which can be done at compile time.

3 Challenges for Mobile Interoperable Media Streaming

As we mentioned in Section 1, two challenges must be solved to enable peer-to-peer streaming of multimedia data between heterogeneous smartphones: (a) how to enable interoperability among incompatible streaming protocols, and (b) how to adapt media containers to consume multimedia data transmitted through an incompatible streaming protocol.

Here, we detail the challenges introduced above. Specifically, Section 3.1 reviews the process of streaming multimedia data from a server to heterogeneous clients. Then, based on this general schema, Section 3.2 details the challenges involved when translating multimedia streaming protocols, while Section 3.3 explains the issues caused by the different media container formats available on current smartphones.

3.1 The Streaming Process

Streaming to heterogeneous devices is classically done by servers supporting a set of audio/video **codecs**, **media container formats** and **streaming protocols**, and comprises three phases: **media capture**, **media transmission** and **media presentation**. The steps commonly required to stream multimedia between two devices are depicted in Figure 1 and are detailed below.

Considering the sequence of steps in Fig. 1, media container formats are used in multiple cases. At Step 1 the demuxing (or demultiplexing) phase refers either to the unwrapping from a disk container if the media source is a file, or to a streamable container if it is a media server or a camera. The elementary stream obtained from Step 1 can be transcoded to a different video/audio compression format and it is then re-multiplexed to a streamable container format in Step 3. The format of multiplexing used is dependent on the streaming protocol, since in most of the cases streaming protocols support a single format.

Media Capture: Media content can originate either from a camera, stored data or from a remote source via a streaming protocol. The input can be already wrapped inside a media container (for instance, MPEG-TS or RTP) by the

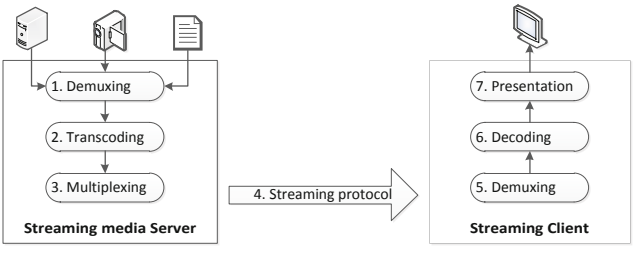


Fig. 1. Multimedia streaming process

source hardware, so the first (Demuxing) step is optional. Possibly the most important characteristic of multimedia content is its audio/video encoding. Indeed, being a highly resource demanding operation, multimedia encoding is subject to software and hardware optimizations on both personal computers and embedded devices. The availability of encoders and decoders therefore varies depending on the mobile operating system, platform and device.

If a client does not support a decoder compatible with the server's encoder, the client cannot consume the media. When a server supports multiple encoders, multimedia data can be re-encoded on a format compatible with the client supported decoders (Step 2 in Fig. 1), but this process is resource consuming and can affect performance, especially when streaming live content.

Media Transmission: Since video and audio frames cannot be directly transferred over an IP network, they are wrapped within media containers that provide the necessary meta-information to facilitate the decoding and correct presentation at the receiver (i.e., client) side. The process of wrapping and unwrapping audio/video frames from a media container is also referred to as multiplexing and demuxing, respectively. This is related to the fact that in some container formats, frames (or frame fragments) from multiple audio and/or video tracks are interleaved. The media transmission also requires control and signalling. This task is assured by means of a communication protocol specifically designed to transport multimedia content. Streaming protocols can be divided in two subgroups:

Real-time streaming protocols are best suited for conversational content such as video conferences where user interaction with the streamed content is important.

Video on-demand protocols are designed to offer better scalability and connectivity; are usually based on the higher level Hypertext Transport Protocol (HTTP) and introduces acceptable delays.

Media Presentation: In order to correctly reproduce an audio/video stream on a mobile phone, it is required that the platform supports the given streaming protocol, media container format, the audio/video codecs and the codec profile used by the encoder. Being a resource consuming activity, multimedia decoding is usually managed by the mobile platform through hardware decoders or by efficient native code implementations. To offer a satisfactory multimedia user experience on resource-constrained devices, mobile platforms provide a default **media player** that applications can access through a standard API. This approach has the advantage of providing a uniform multimedia experience regardless of applications. However, it limits the possibilities to improve audio/video handling in mobile devices since the exposed API is generally limited. For instance, existing decoders used by the player to display multimedia content might be inaccessible for use or extension by applications.

3.2 Streaming Protocol Heterogeneity

Most smartphone platforms support at least one streaming protocol client. The most well known protocols used in mobile phones today are: Real Time

Streaming Protocol (RTSP) [19], Apple HTTP Live Streaming (HLS) [16], Microsoft Smooth Streaming[4] and Adobe HTTP Dynamic Streaming (HDS)[5] (provided that the mobile platform supports Adobe Flash). The most commonly found on mobile platforms is RTSP, but because it uses UDP as transport protocol on unprivileged ports it is inappropriate for use in restricted networks such as 3G and public WiFi hotspots. A standard extension defined in [20] enables interleaving messages over the TCP control connection, but is not supported by most implementations. Protocols designed for video-on-demand scenarios, such as HLS and HDS, are almost equivalent in terms of functionality and concept, but differ in message formats and media containers.

Table 1. Audio/video decoders supported for streaming on smartphones

Decoder / Platform	iOS	Android	BlackBerry OS	Windows Phone 7
H.263	–	+	+	–
H.264	+	+	+	+
MPEG-4	+	+	+	+
AAC-LC, AAC+, eAAC+	+	+	+	+
AMR-NB	–	+	+	+
MP3	+	+	–	+

Still, even if the streaming protocols are incompatible by default, the encoded video and audio elementary streams may be compatible with multiple devices. For example, HLS uses H.264 codec for video, but the same codec is also largely used to stream video over RTSP to Android devices. As it can be seen in Table 1, there exists a common set of video and audio decoders available on multiple mobile phone platforms. In contrast, streaming protocol support is increasingly heterogeneous on mobile platforms, with the arrival of new proprietary protocols such as HTTP Live Streaming and Microsoft Smooth Streaming. The currently supported streaming protocols on mobile phone platforms are presented in Table 2. From both tables, we conclude that multimedia data can be exchanged between heterogeneous smartphones without the need to perform costly transcoding operations. However, it is still necessary to adapt streaming protocols to enable streaming between heterogeneous devices.

Table 2. Streaming protocols supported on smartphones

Protocol / Platform	iOS	Android	BlackBerry OS	Windows Phone 7
RTSP	–	+	+	–
RTSP interleaved	–	–	–	–
RTSP - SRTP	–	–	–	–
HTTP Live Streaming	+	+	–	–
HLS with SSL	+	–	–	–
MS Smooth Streaming	–	–	–	+
MSS with SSL	–	–	–	+

[4] http://www.microsoft.com/silverlight/smoothstreaming/
[5] http://www.macromediastudio.biz/products/httpdynamicstreaming/

3.3 Media Container Adaptation

The conversion between different media container formats is a critical requirement for assuring interoperability between heterogeneous streaming protocols. Supporting both real-time and video-on-demand protocols makes this task more complex due to the mismatching of properties of the protocol groups.

Encoded elementary multimedia data is stored on disk using a media container format (e.g., 3GPP, MP4, AVI). Such containers are designed to be used only in random access scenarios and therefore are not suited for streaming over a network connection. Another type of containers are streamable media containers (e.g., MPEG-TS, ASF, PIFF). They are designed to be transported over IP packet networks, provide methods for fragmenting audio and video streams and may also offer synchronization and recovery mechanisms to cope with network delays or packet losses. The wrapped media packets can contain multiplexed audio/video tracks (e.g., MPEG-TS, PIFF) or single tracks (e.g., RTP). Depending on the streaming protocol type (real-time/on-demand), multimedia fragments differ in size and structure. In general, real-time protocols use lightweight headers and small packet sizes, usually less than the MTU[6] in order to reduce the transfer delay by avoiding packet fragmentation. Video-on-demand protocols regularly use large video fragments composing 10-30 seconds of audio/video each. Such formats commonly rely on the ISO base media file format[7] structure which supports storing of multiple interleaved frames inside a single fragment, [5,1]. Larger fragments reduce the need of receiver buffers but also introduce a start-up delay which is at least equal to the duration of the first fragment.

Real-time streaming protocols are generally designed over the UDP transport protocol because timeliness is much more important than the reliability offered by TCP. Consequently, simple reliability features, such as sequence numbers, sequence identification, synchronization codes, continuity counters, flags and timestamps are integrated in the media container layer to cope with the unreliable nature of the transport. Such features are not necessarily found in the same configuration in all formats. As a result, transforming a real-time stream to a video-on-demand fragment requires complex buffering and efficient transformation of real-time data. Such requirements impose strict temporal constraints for the transformation. It is true that real-time to on-demand protocol translation is less desirable, but interoperability should still remain possible.

4 AmbiStream Architecture

The aim of the AmbiStream middleware is to allow smartphones supporting different streaming protocols to directly connect to, and receive live multimedia content from, other smartphones without using an untrusted server for adaptation. AmbiStream consists of a set of portable server and client components as well as a plug-in interface, designed to reduce the effort of adding support for

[6] Maximum transmission unit (less than 1500 bytes for Ethernet).
[7] ISO/IEC 14496-12:2008.

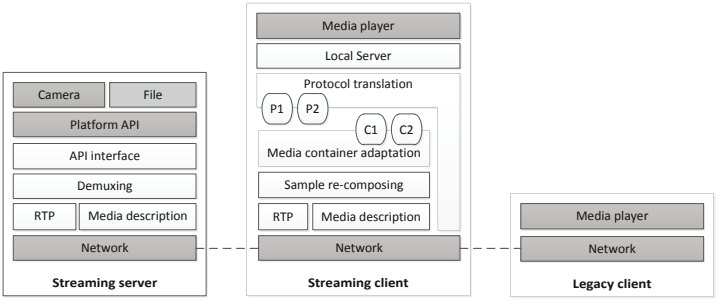


Fig. 2. The AmbiStream middleware architecture

new protocols. The structure of the middleware is presented in Fig. 2, where the greyed components are not part of AmbiStream, but are elements of the mobile platform architecture or external components.

Our work extends the approach proposed by Starlink [6] in two directions. First, our approach enables the translation between real-time and on-demand streaming protocols, which requires buffering, dropping and combination of messages to deliver time-sensitive data at the right moment. Second, we support the translation of container formats, which in the case of multimedia is dependent on the streaming protocol.

The communication is realised in a client-server mode. The streaming server, as well as the client, reside entirely on mobile devices. The current server implementation is designed to support a single streaming protocol. This protocol is translated by the client device to another protocol depending on its native protocol support. On the server side, a platform specific *API interface* has to be designed to access the *Camera* data stream. On the other hand, *File* access for streaming from pre-recorded content can be designed in a portable fashion. We assume that any input data is already multiplexed in the platform's native format (e.g., RIMM proprietary video format for BlackBerry). It is true that demuxing might not be needed if the platform's API gives access to elementary frame buffers. That is why there is an initial demuxing step in Fig. 2. Once the data is *Demuxed* (unwrapped) from its container we obtain the elementary stream tracks (e.g., the audio track) and the necessary meta-data such as sample sizes and frame durations. Considering that the middleware should enable applications to stream both in real-time and on-demand, we use *RTP* [19] as an intermediate streaming protocol. As a consequence, the middleware translates from the intermediate protocol to each existing streaming protocol, thus considerably reducing the total number of bridges required. However, this does not imply that any of the mobile platforms or devices should support this intermediate protocol natively. RTP alone is not sufficient to describe the payload characteristics such as audio/video encoders, sampling frequency, packet fragmenting method, and other media information. AmbiStream thus introduces a negotiation phase where the server sends a XML-formatted description message

to clients for the middleware to correctly instantiate the client protocol bridge and the media container format adapter.

The client receives the *Media description*, and instantiates the appropriate protocol translator (e.g., *P1* or *P2*) and media container adapter. Streaming protocol translators and media container adapters (e.g., *C1* or *C2*) are used as pluggable components created at compile time. To simplify support for a large array of protocols, these components are generated automatically from descriptions of messages and behaviour given in the form of DSL (domain specific language), as detailed in Section 4.1. The plug-ins could as well be generated at run-time, but since the required plugin of the platform is known at compile-time the only use would be to support more legacy devices. Received RTP packets are *Re-composed* into elementary streams and, once a sample is complete, they are passed to the *Media container adapter* (which is detailed in Section 4.2). Depending on the adapted protocol, the samples might be buffered at this point. A *Local server* is managed by the protocol translator that composes the necessary control messages for establishing a streaming session.

The adaptation server running on the client device can be also used as a mediator agent to solve interoperability for streaming enabled legacy devices. This is done using at least three nodes: a server running AmbiStream, a mediator also relying on our middleware and a legacy client (i.e., without AmbiStream or any other additional software installed). The mediator smartphone translates the server streaming protocol to the one supported by the legacy device.

The AmbiStream architecture enables smartphones to stream multimedia between each other without involving a third party server, since all the adaptation is performed on the client side. In terms of privacy, this solution is superior to other architectures that require the stream to pass through an untrusted server for adaptation and/or distribution. So, even though data passes through probably untrusted peers, the authenticity of the stream can still be established using an efficient security protocol such as TESLA [17]. Even legacy clients, that receive the streaming from an intermediate node instead of directly from the server can select a trusted peer based on any trust establishment protocol. The diversity of existing legacy devices such as TVs, tablets, and mobile phones motivate the use of distributed translation nodes instead of a centralised server.

4.1 Streaming Protocol Translation

Because writing protocol adapters for each existing streaming protocol implies a high development effort for a large number of platforms, we introduce an automated protocol translation solution, to enable easier integration of additional protocols. To achieve this, we base our solution on existing research in the domain of automated protocol translation. However, while advanced solutions for interoperability between heterogeneous protocols exist [7,22,6], streaming protocols tend to be more complex because of the constant data flow, time constraints and multimedia wrapper formats.

Our approach is inspired by Starlink [6], a run-time solution for protocol interoperability. Although run-time adaptation of the media format and protocol is

more flexible and enables adapting protocols that are unknown at compile-time, in our case the availability is only subject to the support of mobile platforms, thus making possible to know in advance the adaptation requirements of each mobile device. Also, the adaptation only concerns the client-side since, at the server side we use an intermediate protocol. We thus propose a simpler compile-time interoperability solution based on Starlink.

Streaming protocols are a mix of control and complex data messages. We discuss the translation of the control part of streaming protocols below, while dealing with multimedia data adaptation in Section 4.2. To create a new protocol translator, the developer must provide a high level description in the form of two DSL-based models. One describes the format and structure of messages and the other outlines the protocol states, transitions and the sequence of actions performed at each protocol state. The model is expressive enough for generating message parsers and composers for multiple existing streaming protocols. The model obtained in this form is passed on to a compiler (which is part of the currently presented solution) that produces multi-language (Java, C and C#) protocol bridges in the form of plug-ins (e.g., *P1* and *P2* in Fig. 2) for our middleware.

An schematic example of a message description for HLS protocol is shown in Fig. 3. The description is divided in *Input* and *Output* to differentiate between incoming messages that should be parsed into structured data types and outgoing messages that are composed. This distinction is more important with text protocols, where messages have loose requirements in terms of line order, optional parameters, delimiters, spacing characters and so on. The DSL proposed here supports protocols that use either binary, text or XML message formats. To assure a sufficiently expressive message description, we extract the required fields using value capture patterns defined using Posix regular expressions for text protocols, XPath for XML and based on field size and location for binary protocols. The choice of Posix regular expression for text protocols was driven by its availability on most of the platforms, most notably that it is part of the GNU C library and is compatible with the regular expressions integrated in Java standard library (java.util.regex).

Real time protocols do not usually follow a request-response messaging pattern, as implemented by on-demand ones, but rather a one-way pattern. The problem here is that a protocol translator can not produce a response by calling the real-time inner protocol. In fact, the translator must buffer the messages of the real-time protocol and, upon a request of the video-on-demand client, generate the corresponding message.

4.2 Media Container Format Adaptation

Translating the control part of streaming protocols is not sufficient to distribute multimedia between incompatible protocols. The format in which audio/video content is wrapped also differs depending on the protocol. To achieve a complete solution, the translation between media container formats must also be taken into account. The most important factors that led to the decision to separate

```
<Protocol type="text">
  <Input>
    <Header name="http_head">
        <Var name="Url" type="String"/>
        <Rule test="capture_order(Url)">1</Rule>
        <Capture var="Method"> [RegEx] </Capture>
        <Finish test="empty_line"/>
    </Header>

    <Message name="GET_IDX">
        <Insert>http_head</Insert>
        ...
    </Message>
    ...
  </Input>
  <Output>
    <Message name="IDX">
       <Var name="$TargetDuration" type="Integer"/>
       <Line>#EXTM3U</Line>
       ...
    </Message>
    ...
  </Output>
</Protocol>
```

Fig. 3. DSL describing message formats for the HLS protocol

this part from the protocol translation model are: the much higher complexity of multimedia packets, the dependence relation between messages (order, timing, fragmenting), the buffering requirements, and the multiplexer logic required to interleave multiple media tracks inside one packet/message.

We further divide the media container adaptation in four distinct steps: sample fragmenting, fragment packaging, multiplexing and final adjustment. The process of adapting a stream composed of two tracks (one audio and one video) is presented in Fig. 4. Each of the four phases is defined by the developer using a DSL to describe multimedia containers, different from the ones used for protocol description. Similarly to the generation of protocol translation plug-ins, the description of the multimedia container adaptation is compiled to be deployed to designated platforms. To simplify the description, a number of media packet-related parameters are exposed through the DSL. Parameters include: the length of the media payload, media encoding, fragmentation flag, sampling frequency, sequence number, inner frame sequence number and first/last fragment flag. The components for protocol description and container adaptation are considered to be independent, thus allowing, for example, a protocol to choose between multiple supported data formats. The *Sample Re-composing* middleware component (see Fig. 2) provides real-time input to the container adapter in the form of elementary stream samples for audio and frames for video.

Because we use a real-time protocol (i.e., RTP) for transporting multimedia data, the problem of timing should also be taken into account. We thus add a time-stamp reference to each packet resulting from any of the four phases of media format adaptation. Fragments of one frame share the same time-stamp information, while messages composing multiple frames contain the time-stamp of the first frame and their duration. The time required for a frame to pass through

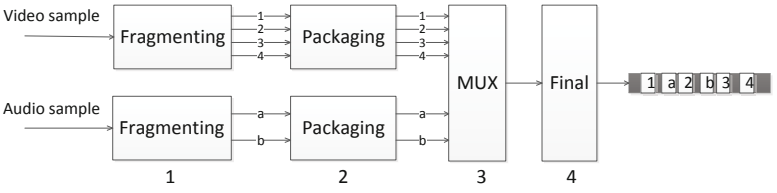


Fig. 4. Adapting the media container format

all of the phases required by the format should not exceed the sampling interval of the content. Failing to assure this property can cause the client to run out of buffered data, resulting in playback stalls. In order to prevent such behaviour, frames should be explicitly dropped such that the output of the conversion is completed at the right time to assure a fluent playback. At this moment, no QoS related limits of packet drops are considered.

The fragmenting step defines the way large audio or video samples are divided into smaller segments according to the limits imposed by the streaming protocol, by the media container or by the network configuration. For example, in the case of MPEG-TS, the samples are split into fragments which are inferior in size to 184 bytes, such that they can be correctly wrapped inside the standard 188 byte packets. For RTP, fragmentation follows the standard RTP Payload Format depending on the codec used (for instance, the one described in [25]). We note that a simplified description of the packet format is very useful in the case of RTP, where there are multiple payload formats depending on the media encoder used. In the case where media content is composed of multiple tracks (i.e., one video and one audio track), two separate fragmenting units are used. The number of fragments created from single frames is variable. Each fragment contains a reference to the time-stamp of its originating frame. The time required for fragmenting one frame should never exceed the sampling interval of the content.

The packaging stage adds individual packet headers. This transformation conforms to [19] for RTP packets and [13] for MPEG-TS. Depending on the protocol, the resulting packets are passed to the multiplexer or sent directly to the protocol translator.

The multiplexing phase assures time-division multiplexing for a set of given fragments or frames of multiple data tracks. Depending on the format, the multiplexing is done at a frame level or at a frame-fragment level. In order to achieve multiplexing at frame level, phase one of the adaptation should be skipped. This phase outputs only at a given time or data limit. Such a limit is necessary to be able to produce media fragments of specified duration or size. The split is always done at random access points of the stream, such that no reference between frames is lost.

The final transformation adds extra headers or packets, such that the resulting fragment is recognised as valid by standard client protocol implementations.

Many existing media container formats also contain a number of specific fields which are particularly hard to model. One example is the MPEG2 Transport Stream [13], which requires a 32-bit cyclic redundancy check value to be added to the Program Association Table package. In such a case we offer the possibility to add function"hooks" inside the DSL media container description. The compiler uses these to generate function templates, that developers can later implement.

5 Experimental Results

In order to evaluate the presented solution, we have implemented AmbiStream in Java and Objective-C and used it on Android and iOS. The goal of the experiments presented here is to evaluate the overall performance of the middleware and the achievable stream quality. The experiments were performed on both Android and iPhone smartphones.

Table 3. Test smartphones used

Device	Samsung GT-I9000	Google Nexus One	iPhone 3G
Role	Server	Client	Client
Platform	Android 2.2.1	Android 2.3.4	iOS 4.2.1
CPU	1 GHz (S5PC110)	1 GHz (QSD8250)	412 MHz
Memory	512 MB	512 MB	128 MB
Media framework	PV OpenCORE	Stagefright	AV Foundation
Stream support	RTSP	RTSP/HLS	HLS

In both of the experiments presented below, the same set of source media files was used. The test files have a duration of 210 seconds, are encoded with a single (H.264-avc video) track, have a CIF frame-size (352 by 288), and a frame-rate of 30 fps. The test is conducted for 16 different bit-rates between 50kbps and 1500kbps using the mentioned file format and content. Each set of tests is repeated at least three times, so each of the metrics presented is characterized by 168 minutes of video streaming to each client device. In total, more than 16 hours of streaming between smartphones were necessary. The mobile phones used are mentioned in Table 3. The first two (Samsung GT-I9000 and Google Nexus One) are used in the first experiment, and all three in the second one.

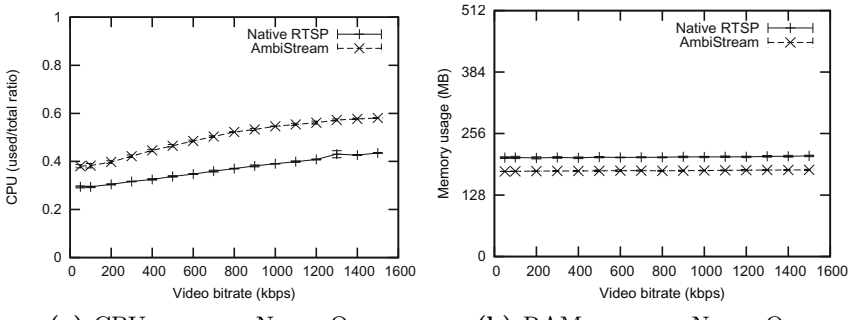
5.1 Collecting Mobile Device Performance Data

Although RTSP provides out-of-band feedback of stream quality through RTCP, we have decided not to use this feature to obtain information related to the quality of service. This is due to the fact that in the case of the media framework Pocket Video OpenCore (used by Android platform in versions preceding 2.3) the information provided is not sufficiently precise. For example, the interval jitter value reported, used to observe the effect of network packet delays, is usually ten times higher than what we found at network level or on the client device. Furthermore, on the newer Stagefright media framework the feedback

always reports no packet loss and inter-arrival jitter equal to zero. Android also provides an information callback from the media player service. Unfortunately, this information is limited to a small set of event codes and does not include any metric.

We have chosen to favour system-wide metrics to more specific ones (i.e., metrics of the application process) because we also make use of native system services and because mobile platforms do not frequently provide equivalent metrics. We use as metrics for device performance: the total CPU utilization and the system-wide used RAM memory. Quality of service metrics considered are the packet delay variation (also referred to as inter-arrival jitter, described in [19]) and packet loss ratio. The quality metrics are only provided for the case where the protocol is adapted. The values are obtained at the middleware level and should indicate the maximum bit-rate achievable while still providing satisfactory quality. The reference test cases, used to compare the overall performance, make use of system media services directly.

On Android mobile phones, the CPU and memory information is obtained by accessing the *proc* filesystem, used as an interface to the operating system kernel on most Linux based distributions. The logs are stored in the internal memory of both Android phones. To avoid that the access to the filesystem and data parsing are influencing the final results, the access to the */proc/stat* and */proc/meminfo* is done every five seconds, and the same file-descriptors are reused multiple times until the end of the test. On the iOS platform, system performance information was collected using the tools integrated with the development kit.



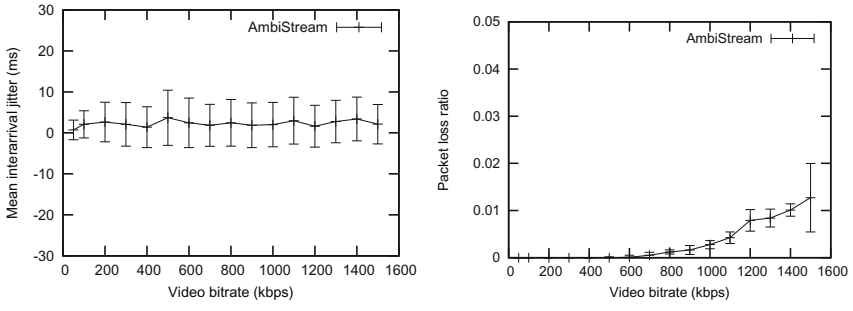
(a) CPU usage on Nexus One **(b)** RAM usage on Nexus One

Fig. 5. AmbiStream performance on Nexus One (RTSP)

5.2 Translating to RTSP between Android Smartphones

In this first experiment, we show that adaptation from the middleware intermediate protocol to RTSP/RTP/UDP is sufficiently efficient to be used in mobile multimedia-enabled applications. Since in this case the message format used by the client protocol is equivalent to the middleware transport protocol, the wrapping and unwrapping of messages is simpler than in other cases. Nevertheless, this client protocol is the only real-time streaming protocol currently available on

mobile phones, and is thus interesting to analyze the feasibility of streaming real-time multimedia data through the middleware. Another experiment involving a more complex media format adaptation is presented at the end of this section. In this test we use two server implementations: one using the AmbiStream intermediate protocol and the other using RTSP. The RTSP server is not part of the solution but it is used in this experiment to determine the overhead of the adaptation (on the client-side) with reference to the native RTSP support.



(a) Inter-arrival jitter **(b)** Packet loss

Fig. 6. Adapted stream quality (RTSP)

In the case of protocol translation to RTSP, the performance of the client device (realising the content and protocol adaptation) is not badly affected, with a processing overhead of less than 20% compared to a native RTSP session (see Fig. 5a). As with all of the experiments conducted, the memory usage remains constant, or increases slightly because of buffers required for higher data-rates (Fig. 5b). The fact that our solution uses slightly less memory than the reference one is due to the way jitter buffers are managed internally by the RTSP client, most probably being influenced by the different transport protocols (UDP and TCP). The quality of the stream remains within acceptable limits in terms of inter-arrival jitter (see Fig. 6a) and packet loss (Fig. 6b), for all the test cases (from 50 to 1500kbps) considered.

5.3 Translating to HLS between Android and iOS Smartphones

The second experiment consists of translating the intermediate middleware protocol to HTTP Live Streaming, using two different client platforms: Android 2.3.4 and iOS 4.2.1. The choice of the smartphones is motivated by their native support of HLS. This way we can reason about the overhead introduced by our middleware layer with two different devices. Contrary to the first experiment, this one requires data conversion between RTP and MPEG-TS. MPEG-TS is one of the most used multimedia formats, most notably for digital television. The conversion from RTP to MPEG-TS requires a large number of transformations, thus providing a good impression of achievable on-the-fly conversion limits of media formats on current generation smartphones.

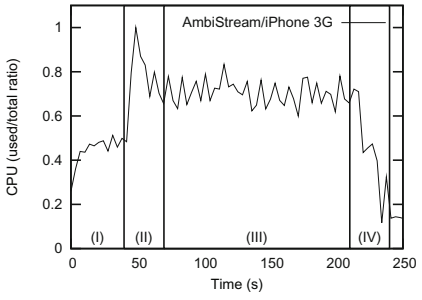


Fig. 7. Data capture (HLS)

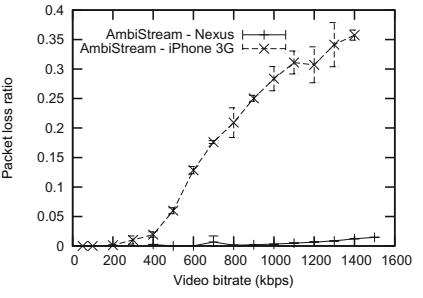
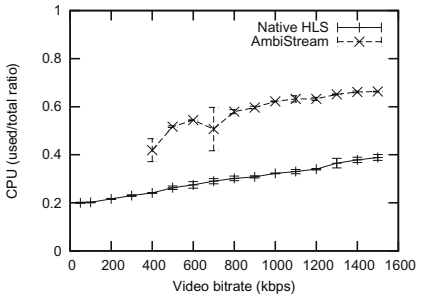
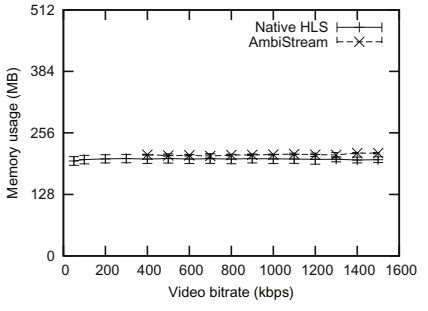


Fig. 8. Packet loss (HLS)



(a) CPU usage

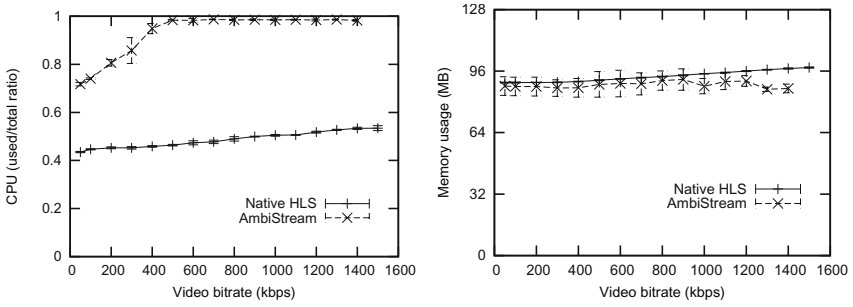


(b) RAM usage

Fig. 9. AmbiStream performance on Nexus One (HLS)

Because HLS protocol requires the existence of a cached amount of content on the server-side before a client can connect (and begin playback), while the intermediate AmbiStream protocol does not, a 30s start-up delay is introduced by the middleware layer to allow protocol translation. This aspect restricts the use of the middleware for real-time applications in this situation. This is not the case when the device supports a real-time protocol. During this delay period, less memory and CPU are used. To better evaluate the performance of the devices, we divide the experiment run in four periods (e.g., as shown in Fig. 7 for CPU utilisation): *(I) the buffering period* (only multimedia data adaptation is performed), *(II) the media-player start-up* (causes a short increase in CPU usage), *(III) the streaming period* (both data adaptation and playback are performed) and *(IV) the stream-end* (the source has finished streaming, but the playback is continued until buffer depletion). Thus, only the part *(III)* of the observation was used to produce the results presented in Figures 9 and 10.

As expected, the difference in container formats (RTP and MPEG-TS), increases the overhead of AmbiStream. For Android platform, the tests for bit-rates inferior to 400kbps (in Figures 9a and 9b) were discarded due to the existence of a minimal caching size, requiring a longer start-up delay. While on the Nexus One, the overhead introduced does not reach a quality limit for bit-rates below



(a) CPU usage **(b)** RAM usage

Fig. 10. AmbiStream performance on iPhone 3G (HLS)

1500kbps, the iPhone 3G is only able to adapt streams of up to 400kbps. Above this limit, the packet loss (see Fig. 8) becomes noticeable and the media-player suffers playback stalls. The results on the iPhone are worse due to the significantly lower processing power and memory (see Fig. 3). Nevertheless, according to the mobile platform providers, a 400kbps video bit-rate is considered to be medium/high quality for smartphones[8] [9]. Considering the results in Figure 10b, we see that the memory usage is decreasing (in the case of AmbiStream) for higher video bit-rates. This behaviour is normal considering the packet loss (see Figure 8).

6 Conclusions and Future Work

In this paper we have identified the challenges raised by the heterogeneity of the streaming protocols of existing mobile phone platforms. Further, we have introduced the AmbiStream multimedia-oriented middleware architecture, designed to enable the multi-platform and multi-protocol interoperability of streaming services. We have also shown the applicability of the presented solution with an experiment on two different platforms and two different streaming protocols.

AmbiStream was modelled taking into consideration the architecture of modern smartphone platforms, such that resource critical operations (e.g., multimedia decoding) are managed by each platform internally. We prove that automated streaming protocol adaptation can be done locally on mobile phone platforms without sacrificing performance or extensibility. Furthermore, we enable legacy devices to employ unsupported streaming protocols by using an AmbiStream-enabled device as mediator intermediary.

We intend to continue this work by extending the current model, taking into account challenges such as routing over different networks and multi-peer collaboration. We will then integrate AmbiStream with iBICOOP [3], a middleware

[8] http://developer.apple.com/library/ios/#technotes/tn2224/_index.html
[9] http://developer.android.com/guide/appendix/media-formats.html

designed to enrich the user collaboration and provide seamless access across different networks and devices. Such an integration will complement the existing solution with features such as discovery, distributed storage and partnership management, enabling the development of rich cross-platform and multimedia-enabled applications. We will also port the solution to Blackberry and Windows Phone to evaluate the approach on a greater number of mobile platforms.

Acknowledgement. This work is partially supported by the FP7 ICT FET IP Project CONNECT.

References

1. Adobe Flash Video File Format Specification, Version 10.1 (August 2010), http://download.macromedia.com/f4v/video_file_format_spec_v10_1.pdf
2. Andronache, A., Brust, M.R., Rothkugel, S.: Multimedia content distribution in hybrid wireless networks using weighted clustering. In: Proceedings of the 2nd ACM International Workshop on Wireless Multimedia Networking and Performance Modeling, WMuNeP 2006. ACM (October 2006)
3. Bennaceur, A., Pushpendra, S., Raverdy, P.G., Issarny, V.: The iBICOOP middleware: Enablers and services for emerging pervasive computing environments. In: PerWare 2009 IEEE Middleware Support for Pervasive Computing Workshop (October 2009)
4. Blair, G.: On the failure of middleware to support multimedia applications. In: Interactive Distributed Multimedia Systems and Telecommunication Services (October 2000)
5. Bocharov, J., Burns, Q., Folta, F., Hughes, K., Murching, A., Olson, L., Schnell, P., Simmons, J.: The Protected Interoperable File Format (PIFF) (March 2010)
6. Bromberg, Y.-D., Grace, P., Réveillère, L.: Starlink: runtime interoperability between heterogeneous middleware protocols. In: Proceedings of 31th International Conference on Distributed Computing Systems, ICDCS (IEEE) (June 2011)
7. Bromberg, Y.-D., Réveillère, L., Lawall, J.L., Muller, G.: Automatic Generation of Network Protocol Gateways. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 21–41. Springer, Heidelberg (2009)
8. Coulson, G., Blair, G., Davies, N., Robin, P., Fitzpatrick, T.: Supporting mobile multimedia applications through adaptive middleware. IEEE Journal on Selected Areas in Communications (September 1999)
9. Coulson, G.: A configurable multimedia middleware platform. IEEE MultiMedia (January 1999)
10. Cruz, R.S., Nunes, M.S., Gonçalves, J.E.: A Personalized HTTP Adaptive Streaming WebTV. In: Daras, P., Ibarra, O.M. (eds.) UCMedia 2009. LNICST, vol. 40, pp. 227–233. Springer, Heidelberg (2010)
11. Curran, K., Parr, G.: A middleware architecture for streaming media over IP networks to mobile devices. In: Wireless Communications and Networking (March 2003)
12. Do, N.M., Hsu, C.H., Singh, J.P., Venkatasubramanian, N.: Massive live video distribution using hybrid cellular and ad hoc networks. In: Proceedings of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2011) (June 2011)

13. ITU-T Rec. H.222.0 — ISO/IEC 13818-1, Generic coding of moving pictures and associated audio information, http://www.iso.org/iso/catalogue_detail?csnumber=44169
14. Lindeberg, M., Kristiansen, S., Plagemann, T., Goebel, V.: Challenges and techniques for video streaming over mobile ad hoc networks. Multimedia Systems 17(1) (February 2011)
15. Nakazawa, J., Tokuda, H., Edwards, W.K., Ramachandran, U.: A bridging framework for universal interoperability in pervasive systems. In: International Conference on Distributed Computing Systems (July 2006)
16. Pantos, R., May, W.: HTTP Live Streaming (Internet-Draft) (March 2011), http://tools.ietf.org/html/draft-pantos-http-live-streaming-06
17. Perrig, A., Song, D., Canetti, R., Tygar, J.D., Briscoe, B.: Timed Efficient Stream Loss-Tolerant Authentication. RFC 4082 (Proposed Standard) (June 2005), http://tools.ietf.org/html/rfc4082
18. Ransburg, M., Jonke, M., Hellwagner, H.: An evaluation of mobile end devices in multimedia streaming scenarios. In: Mobile Wireless Middleware, Operating Systems, and Applications (July 2010)
19. Schulzrinne, H., Casner, S., Frederick, R., Jacobson, V.: RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard) (July 2003), http://tools.ietf.org/html/rfc3550, updated by RFCs 5506, 5761, 6051, 6222
20. Schulzrinne, H., Rao, A., Lanphier, R.: Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard) (April 1998), http://tools.ietf.org/html/rfc2326
21. Spalazzese, R., Inverardi, P., Issarny, V.: Towards a formalization of mediating connectors for on the fly interoperability. In: Joint Working IEEE/IFIP Conference on Software Architecture 2009 & European Conference on Software Architecture 2009 (September 2009)
22. Bissyandé, T.F., Réveillère, L., Bromberg, Y.-D., Lawall, J.L., Muller, G.: Bridging the Gap between Legacy Services and Web Services. In: Gupta, I., Mascolo, C. (eds.) Middleware 2010. LNCS, vol. 6452, pp. 273–292. Springer, Heidelberg (2010)
23. Van Lancker, W., Van Deursen, D., Mannens, E., Van de Walle, R.: Implementation strategies for efficient media fragment retrieval. Multimedia Tools and Applications (March 2011)
24. Vu, L., Nahrstedt, K., Rimac, I., Hilt, V., Hofmann, M.: ishare: Exploiting opportunistic ad hoc connections for improving data download of cellular users. In: 2010 IEEE GLOBECOM Workshops (December 2010)
25. Wang, Y.K., Even, R., Kristensen, T., Jesup, R.: RTP Payload Format for H.264 Video. RFC 6184 (Proposed Standard) (May 2011), http://tools.ietf.org/html/rfc6184

Virtualizing Stream Processing

Michael Duller[1], Jan S. Rellermeyer[2], Gustavo Alonso[1], and Nesime Tatbul[1]

[1] Systems Group, Department of Computer Science,
ETH Zurich, Zurich, Switzerland
{michael.duller,alonso,tatbul}@inf.ethz.ch
[2] IBM Austin Research Laboratory, Austin, TX, U.S.A.
rellermeyer@us.ibm.com

Abstract. Stream processing systems have evolved into established solutions as standalone engines but they still lack flexibility in terms of large-scale deployment, integration, extensibility, and interoperability. In the last years, a substantial ecosystem of new applications has emerged that can potentially benefit from stream processing but introduces different requirements on how stream processing solutions can be integrated, deployed, extended, and federated. To address these needs, we present an *exoengine* architecture and the associated *ExoP* platform. Together, they provide the means for encapsulating components of stream processing systems as well as automating the data exchange between components and their distributed deployment. The proposed solution can be used, e.g., to connect heterogeneous streaming engines, replace operators at runtime, and migrate operators across machines with a negligible overhead.

Keywords: stream processing, federation, virtualization.

1 Introduction

Applications like financial market data processing or network intrusion detection require processing large volumes of continuously arriving data with high throughput and low latency. Stream processing supports such applications using a model whereby data arrives continuously at the stream processing engine (SPE) and triggers the evaluation of queries stored in the SPE. Within the last decade, data stream processing has gone from a research idea (e.g., Aurora [2], STREAM [20], and TelegraphCQ [9]) to a widespread solution, with several commercial products already available [27,29,16,4].

Stream processing has proven to be useful for many applications. However, its applicability is still limited in terms of interoperability and deployment.

Interoperability refers to the integration of heterogeneous SPEs [28]. A common scenario involves different engines run by different and autonomous entities that must work together but cannot resort to a homogeneous solution. We have encountered such a scenario in automatic financial compliance checking, where government authorities validate streams of transactions created by financial institutions. A similar scenario arises in the supply chain management, where different RFID and bar code technologies, pallet and container tracking systems, and book keeping and stock control software need to be coordinated even across large geographic distances. From these scenarios we derive

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 269–288, 2011.

the requirement to be able to host in a single platform different systems that communicate through well-defined interfaces, while maintaining the authoritative boundaries imposed by each organization.

In terms of deployment, there is an increasing need to deploy SPEs flexibly to run them as virtual entities across a cluster and even across a cloud computing facility. The scenarios and motivation for this requirement are identical to those for standard applications and relational databases: elasticity, cost reduction, and fast provisioning.

To address these challenges we explore the possibility of *virtualizing* any component of a streaming engine (operators and buffers as well as entire engines) so that they can be automatically deployed, managed, and composed in a flexible and dynamic manner. The aim is to build a generic middleware *platform* that allows to (1) encapsulate existing engines (either for embedding into applications or for composition); (2) support combinations of heterogeneous operators; and (3) provide the functionality needed in a distributed platform (e.g., support for operator migration, replacement, data routing).

Like other virtualization approaches (e.g., machine virtualization like Xen or VMWare, and managed language runtimes for bytecode like Java VMs or the .NET CLR), the main objective of our approach is to gain flexibility (e.g., location transparency), ease of management (e.g., push-button deployment of virtual machines), and potential for optimizations (e.g., replacement of performance-critical code with an optimized version at runtime). The architecture proposed is inspired by the concept of the exokernel for operating systems [12,19]. Like exokernel, it implements as few policies and makes as few assumptions as possible to support a wide range of different SPEs well.

Middleware has already proven to be useful in providing additional features for data processing systems like, e.g., TP monitors or message queueing systems for traditional databases. In fact, several engines have been extended with middleware platforms: IBM's System S [17] or Yahoo's S4 [21]. These systems are built as extensions to one particular SPE. Our approach is a pure middleware system that is engine-independent. In that way, we do not impose engine-specific semantics or a processing model, but cater to dynamic and distributed operation, deployment, and lifecycle management and provide interoperability between heterogeneous SPEs with potentially different semantics. From our own work, we have explored semantic aspects of integrating heterogeneous SPEs in MaxStream [6], and wide-area, stream-based processing of personal information in XTream [11], both through middleware-based solutions.

In this paper, we present the design, use, and implementation of *ExoP*, an architecture for stream processing. ExoP provides well-defined, extensible interfaces for encapsulating stream processing entities (operators, buffers) and building applications on top of these. It also supports dynamic composition, dynamic data routing, and component lifecycle management.

The results presented in the paper validate the potential for the ideas behind ExoP as they cannot be achieved with any other system we are aware of. For instance, we have ported two existing, different SPEs into ExoP. One of them is the MXQuery engine [7]. We use the implementation of the Linear Road benchmark [5] with MXQuery to show that ExoP has a negligible overhead ($0.7\,\%$; see Sect. 5.4), and yet, it provides the flexibility to the implementation that we claim: dynamic and distributed deployment and component lifecycle management. We show that we can replace at runtime part

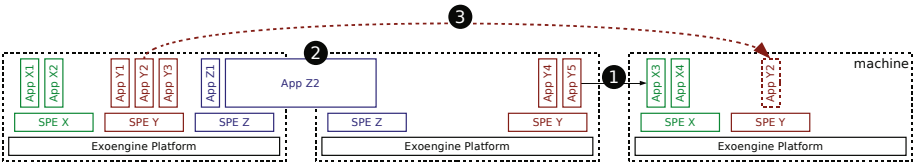


Fig. 1. Exoengine vision

of the system with a native implementation of the operators without interruption in service (see Sect. 5.5) and that we can turn the originally centralized implementation of the Linear Road benchmark into a distributed implementation (see Sect. 5.6) to achieve one order of magnitude improvement in performance; a load factor of 64, which is equal to the fastest published results [30] of a highly optimized, distributed implementation. The other engine ported to ExoP is the Stanford STREAM system [20]. In the paper, we show that we can deploy the engine on our platform, supply queries as part of our configuration mechanism, and federate STREAM and MXQuery (see Sect. 5.8) with a minimal development effort (see Sect. 5.7).

2 Exoengine Architecture

Figure 1 illustrates the vision of the exoengine architecture. It virtualizes stream processing and thus enables ❶ multiple applications written for different engines/query languages to exchange data across machine (platform) boundaries; ❷ single applications to run across multiple machines; and ❸ applications to be migrated to other machines.

2.1 Layers

The exoengine architecture considers stream processing applications at different layers of abstraction (Figure 2). On top, a high-level abstraction, e.g., a streaming query language, presents the *interface* to the system. This interface is provided by *application builders* (see Sect. 4.2). In the example shown, the interface is CQL [20], a language for continuous queries. Being able to expose arbitrary, high-level interfaces on top of the system enables *reuse* of existing applications developed against these interfaces.

The *data processing model* is the data management view onto the architecture and captures how data flows and is processed. It is a graph of entities that process data ("operators" as a first approximation), which we call *slets*, and entities that buffer and forward data, which we call *channels*. Section 2.2 provides more details. The data processing model is generic enough to fit different flavors of stream processing (e.g., push vs. pull driven engines) and thus enables *interoperability*.

The *implementation model* is the systems view onto the architecture. It specifies implementation details of slets and channels (e.g., interfaces) and adds a *connector* entity, which captures distribution in the model (see Sect. 2.3). The implementation model grasps elements as individually managed components, wires them using loose coupling, enables remote operation through connectors, and thus enables *flexible deployment*.

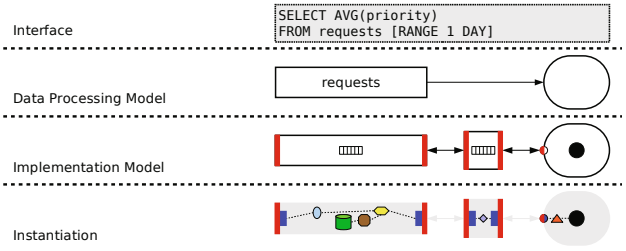


Fig. 2. Layers of abstraction

Ultimately, an *instantiation* of these entities is concretely implemented in some programming language. The generic parts of these entities, as well as the platform itself, are implemented by the platform provider and the specific parts of these entities (e.g., operator logic, custom buffer implementation) are either implemented manually as part of the application implementation or generated by the application (e.g., by a query compiler).

2.2 Data Processing Model

The two fundamental building blocks of stream processing are operators, which process data, and buffers, which forward and buffer data (e.g., to form finite windows over infinite streaming data). The data processing model of the exoengine architecture thus considers operators, which we call *slets*, and buffers, which we call *channels*. Figure 3 illustrates the model as a mesh of channels (rectangles) and slets (ovals). Slets have input and output ports, and each port can connect to one channel. In the figure, ports are implicitly illustrated as the places where arrows enter or leave slets. π*-slets* process data and behave like conventional operators. They can have any number of input and output ports. At the edges of the processing mesh, α*-slets* adapt data sources and ω*-slets* adapt data sinks, providing clean interfaces for exchanging data with the platform. α-slets have no input ports, they only receive data from external sources. Likewise, ω-slets have no output ports and only send data to external sinks. Sources and sinks can be any external device, application, or component that emits or consumes data, respectively.

Data is processed as discrete *items* (tuples) that flow from the sources on the left through the application mesh to the sinks on the right. Slets can transform data that arrives at an input port in any way, including dropping it, aggregating it into internal state, or creating and emitting new data. Channels can be seen as *views* over the upstream processing mesh. Similar to views in traditional databases, they contain the results obtained by processing source data (data sources in exoengine, data tables in databases) with the view definition. In databases, the view definition is a query and in the exoengine architecture it is the part of the mesh that is connected to the channel's input. Channel implementations can persist their contents, resembling materialized views in databases.

Processing can be push- or pull-driven to support all existing systems and to be able to combine them. Thus, the puristic, basic interfaces for data exchange define two methods:

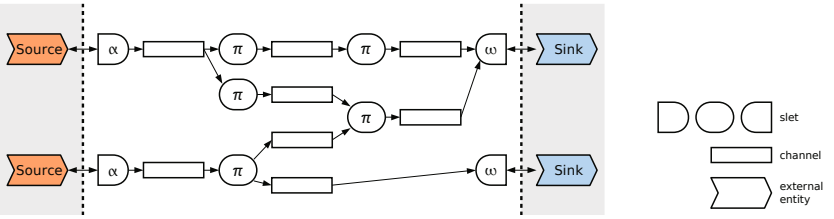


Fig. 3. Data processing model

push(item): push an item to the input of a channel or an slet
pull(): request items from the output of a channel or an slet, return a set of 0. . . n items.

Advanced functionality and optimizations can be implemented as extensions to the interfaces, including channels that provide individual windows for connected slets based on count, time, or explicit eviction from the buffer by the slet (semantic windows); sharing multiple windows and the materialized view from the same physical buffer; or allowing pull requests that are augmented by a query to push selectivity towards data sources. We use channels that materialize their contents or provide windows to slets for the Linear Road implementation presented in the evaluation (see Sect. 5).

2.3 Implementation Model

The implementation model captures implementation aspects, which include actual component interaction and distributed operation. It is based on service-oriented software design and the flexible and loose coupling between components, which facilitates dynamic changes to the processing mesh. It adds an additional type of component, *connectors*, as an indirection between slets and channels used to capture distributed operation in the model. Figure 4 illustrates the implementation model. It depicts all implementation details from slets emitting items on the left, to *input* connectors (*Conn.*), to a channel, to *output* connectors, to slets consuming items on the right. Ports, buffers, and the implementation of slets have been made explicit in the illustration.

Connectors are part of the virtualization strategy to enable distribution. They can be omitted as an optimization if channel and slet reside on the same instance of the platform. In the distributed case, where a channel residing on one instance of the platform is accessed by slets residing on another remote instance, *remote connectors* encapsulate communication between the instances of the platform. One half of the remote connector is installed on the platform instance of the channel and the other half is installed on the remote platform instance, where it represents (*proxies*) the channel. For every remote platform instance that accesses the channel, one remote connector is used to serve all slets on that instance. Section 4.3 with Fig. 6 illustrates distributed operation.

Using connectors as local proxies of channels can also improve performance and robustness. *Smart connectors* can, e.g., cache the content of the channel they are representing, serve requests from their own buffer, and thus reduce latency and save bandwidth. Similarly, when the connection between the smart connector and its counterpart

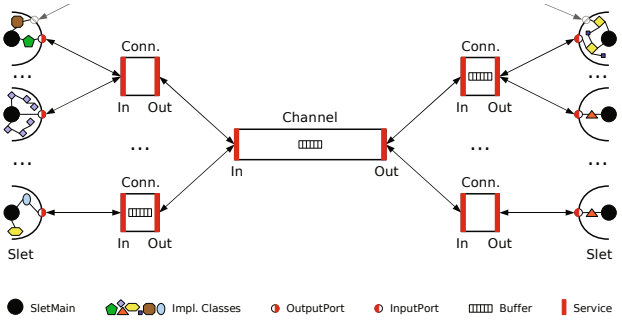


Fig. 4. Implementation model

on the remote platform is not available, the connector can autonomously work in offline mode. The wiring of slets to the connector can be left unchanged, because transitions between online and offline mode happen inside the connector, behind the interfaces.

In Fig. 4, arrows between components are bidirectional as they represent the component interaction in terms of service method invocations rather than in terms of data flow. Ellipses (. . .) between slets or connectors indicate that any number of instances thereof can exist and interact with one instance of a connector or channel, respectively. Service interfaces used in data exchange are depicted using thick bars. To exchange data, components call the *push(item)* method on the *InputPort* or *In* services, or the *pull()* method on the *OutputPort* or *Out* services.

The implementation model separates concerns of processing (slets), storage (channels), and communication (connectors) into separate entities. This separation facilitates capturing resource requirements and implementing respective optimizations.

2.4 Component Life Cycle Management

Every component running on top of an exoengine platform also provides a management service. At runtime, the platform interacts with a component through this service to perform common, generic management tasks like monitoring, suspending, and restarting it, or exchanging individual configuration data. The platform takes care of managing and persisting component configuration data (e.g., the particular query an engine in an slet is executing), component state (e.g., the counters of an aggregation operator wrapped as an individual slet), and applications (e.g., which instances of slets are connected to which instances of channels and thus form an application).

3 Stream Processing with the Exoengine Platform

One way to use our platform is to implement from scratch fully distributed, heterogeneous data streaming applications. This implies writing or synthesizing each operator and the additional components for our platform. As an example, we are in the process of implementing a wide-area, peer-to-peer stream processing infrastructure for exchanging personal data in a streaming manner across collections of devices and locations [11].

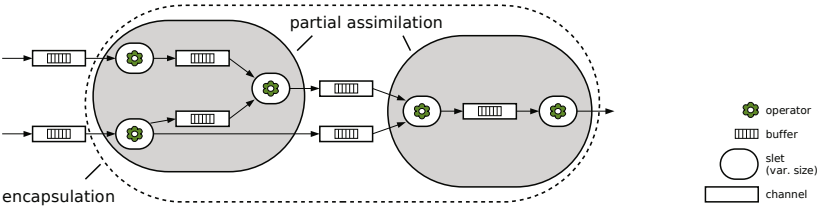


Fig. 5. Wrapping alternatives

3.1 Porting Existing Stream Processing Engines

In addition to implementing applications and their components from scratch, applications for existing SPEs can be reused by porting these SPEs to our platform. We distinguish three levels of granularity for porting existing SPEs: individual operators ("assimilation"), bare stream processing engines ("partial assimilation"), and complete applications ("encapsulation").

If a streaming engine has been implemented in a sufficiently structured way, our platform allows to wrap operators and buffers so that they become explicitly visible to the platform as slets and channels, respectively. This allows reusing operators and buffers without changing the semantics of the underlying engine, while opening up all three possibilities illustrated in Fig. 1. Assimilation is illustrated in Fig. 5 by individual operators and buffers being wrapped as slets and channels.

It is possible to wrap monolithic engines as an slet and use multiple instances thereof to compose an application, e.g., consisting of multiple queries. Each query is executed by one instance of the engine slet. This is the approach we use in the evaluation (Sect. 5) to wrap existing XQuery [7] and STREAM [20] engine implementations. Partial assimilation reduces the porting effort, while still providing access to intermediate results and allowing distributed deployment. It opens up all three possibilities illustrated in Fig. 1 but at a coarser granularity (e.g., at the level of individual queries) compared to full assimilation. In Sect. 5.6 we show the effectiveness of turning a centralized application composed of multiple instances of a partially-assimilated XQuery engine into a distributed application and, thereby, enabling it to handle higher load. Partial assimilation is illustrated in Fig. 5 by the two big, gray slets containing multiple operators and buffers/internal state, which are not explicitly visible to the exoengine platform.

In encapsulation, we wrap an entire application with all the engines and queries into a single slet. This allows to take already existing applications and make them available as a service, in the form of an slet. While limited in flexibility, this approach still facilitates the runtime management of the application and the combination of its ultimate inputs and outputs with other applications. Encapsulation is illustrated by the large, dashed slet in Fig. 5, which encapsulates everything.

3.2 Extensibility

The exoengine model generally matches stream processing applications. The functionality provided by the interfaces (*push(item)* and *pull()*), however, is only suitable for

basic data exchange and not sufficient to implement a full-fledged SPE. The interface between operators and storage/buffer instances in an SPE is typically richer and additionally supports, e.g., index-based access to data or bulk access to multiple tuples at once. The exoengine architecture supports any kind of interaction between slets and channels and thus allows to keep intrinsic implementation details of existing SPEs.

Slet and channel implementations can extend the service interfaces for data exchange (thick bars in Fig. 4). Thus, they can interact through additional methods as needed, without losing the property that the platform manages the dynamic binding between components' services. Connectors also need to support the methods of the extended service interfaces between slets and channels. In the local case, connectors are empty and simply omitted. In the distributed case, standard remote connectors only need to pass method calls through. Thus, they are created automatically by the platform by inspecting the extended interfaces of the components using, e.g., reflection. Distribution-aware implementations of SPEs for the exoengine architecture can provide implementations of smart connectors for enhanced remote operation.

In addition to extensions of interfaces on the data path, implementations of components can also extend their management interfaces to, e.g., allow an optimizer to replace parts of the processing mesh in a controlled manner (e.g., instruct buffers to pause and operators to persist internal state) or implement a richer configuration mechanism.

4 Platform Implementation

In this section, we discuss aspects of implementing an exoengine platform conforming to the data processing and implementation models presented in the previous sections. We also discuss how to implement applications running on top of it and present our prototype of an exoengine platform, ExoP. Though we have implemented the aspects discussed below in our prototype, they are applicable to any platform implementation.

4.1 Component Implementation

Components for the exoengine platform are implemented in a reusable manner, which allows the use of multiple instances of each component without any side effects (e.g., no global state, no singletons). Every instance of a component has a unique identifier in the platform. Every instance of a channel or connector provides one distinct service for its input and one for its output. Every port of an slet also provides a distinct service for data exchange. These services implement the basic interfaces defined by the architecture and potential extensions thereof. In addition, every component implements the respective management interface (slet, connector, or channel) and potential extensions to it.

An implementation of the exoengine platform provides the generic parts of slets, connectors, and channels as a library. These generic parts contain the necessary and recurring glue code that deals with registering a component's services with the underlying service framework, creating and destroying slet ports, exchanging configuration and state with the exoengine platform, and retrieving the connected components' input

(port) and output (port) service objects to invoke methods on them. The developer of a component concentrates on the component's actual functionality and writes the component against the API of the glue code library—thus generally without having contact with the details of the underlying service-based implementation of the exoengine platform.

4.2 Application Builders

Applications are created, modified, and removed by *application builders*. These provide the programming interface and abstraction to the system, typically through a high-level, declarative interface like a streaming query language or a graphical user interface.

Application builders register slet and channel implementations with the platform, instruct the platform to create instances thereof and how to wire them, interact with these instances through the management interface, and interact with remote platform instances. The controlling parts of a stream processing system (query compiler, optimizer, control API) become application builders. A mesh of slets and channels can have cycles and it is the responsibility of the application builder to ensure that this does not cause adverse effects.

Similar to the implementation of components, application builders do not need to know the details of the underlying service-based implementation of the platform. Instead, a management service provided by the platform is in charge of composition and management of all components in one instance of the platform. It provides methods to, e.g., create a new instance of a component, wire two components, or change the configuration of a component. As a response to these methods, the management service creates instances of slets, channels, and connectors; assigns a unique component identifier and configuration (e.g., the query that a particular slet is executing or whether a channel should persist data); and registers them under the corresponding service interfaces. The platform persists the configuration of every component and whether the component is active. This information is reused when an application or parts of it are restarted (e.g., migration). Then, components are recreated, receive the persisted configuration, and are started automatically, relieving application builders from these tasks.

4.3 Distributed Operation

Application builders access remote platforms through a remoting service provided by the platform. Similar to the management service, it abstracts from implementation details and provides high-level methods to connect to known remote platforms, discover channels of interest in the network vicinity (e.g., using multicast discovery), access the management service of remote platforms, migrate components between platform instances, and connect local slets to remote channels or vice versa.

Stateful components must implement methods for (de)serializing their state in order to enable their stateful migration (*memento* pattern). The platform calls these methods and handles the serialized state between suspending and resuming a component.

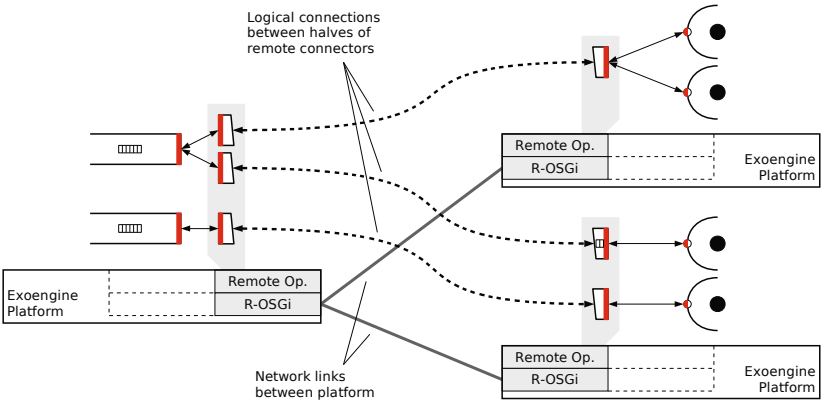


Fig. 6. Distributed operation using connectors

Every instance of the platform maintains one connection to every remote platform instance. Through this connection, all communication takes place, as is illustrated in Fig. 6. Remote connectors are provided by the *remote operations* component of ExoP. By default, network failures result in the removal of the remote connector service, which appears like any other dynamic change to the processing mesh. Smart connectors can override this behavior and remain registered and thus connected.

4.4 Prototype

We have implemented the ideas presented in this paper in *ExoP*, our prototype of an exoengine platform. ExoP is implemented as a componentized, service-oriented system using Java and OSGi. This section presents some of its implementation details.

OSGi. ExoP is based on the OSGi Service Platform [22]. OSGi is a widely used (e.g., Eclipse IDE, application servers) framework for module management and service composition for Java. Modules are called *bundles* and explicitly state code dependencies on other bundles. Bundles can be installed, uninstalled, updated, started, and stopped at runtime. The OSGi framework handles the dependencies that arise in the process.

Services are implemented as Java classes, which are registered with the OSGi framework's service registry under one or more interfaces. A service registration can further be augmented by a set of key/value properties. Service clients can look services up in the registry, including filters on properties. When fetching a service they receive a direct Java reference to the object registered as the service. OSGi provides loose coupling and dynamic service composition within a Java VM. The open source project R-OSGi [25] extends OSGi to support dynamic service composition across multiple Java VMs.

ExoP is implemented as a set of OSGi bundles and uses R-OSGi as the communication fabric to interact with remote platforms (see Fig. 6). ExoP is modular and dynamic itself and a subset of its bundles (e.g., management) can be (un)loaded at runtime.

Component Implementation. ExoP provides an API that facilitates the implementation of slets and channels. Using the example of slets, one class of an slet implementation must implement the interface *SletMain*, which defines methods that are called at initialization or state transitions. In Fig. 4, this class is represented as solid black disk.

Multiple instances of the same slet can exist and each is instantiated by creating a new instance of the class implementing *SletMain*. During initialization, an object of type *SletUtil* is passed to the slet. Through this, slets interact with the platform to create and destroy ports, and to update configuration and state.

ExoP's component model for slets, channels, and connectors extends OSGi's lifecycle management facilities. The API hides the details of the service-based design and allows developers to concentrate on the actual logic of the slet. Interactions with the OSGi platform, like registering services or persisting configuration, is implemented in ExoP and happens behind the API. For example, when an slet calls the API method to create an input port, a port object is instantiated, a unique identifier assigned, the object added to the slet's list of ports, and eventually registered with the OSGi service registry under the *InputPort* service interface and with the identifier as service property.

When an instance of an slet is created, a configuration object for that instance is created and the configuration is persisted with OSGi's Configuration Admin service, resulting in a callback to the particular instance of a Managed Service Factory implementation. The factory then creates an instance of a generic *SletImpl* and an instance of the specific *SletMain* slet implementation (supplying the instance of *SletImpl* as *SletUtil*), calls initialization and start methods on it, and eventually registers the *SletImpl* under the *Slet* service interface and with a set of service properties (including the unique Slet instance identifier) with the OSGi service registry. These steps are implemented in ExoP's management bundle and hidden behind its *ComponentManager* service interface.

Component Binding and Interaction. Components are bound to each other (i.e., a link is created in the mesh) by assigning the unique identifier of the service of the component with cardinality one to a specific "connected to" property of the service of the component with a higher cardinality. When a port is connected to a connector, the connector's unique identifier is saved in the port's "connected to" property. Likewise, for connectors and channels, the channel's identifier is saved in the connector's property.

When a component wants to interact with the component(s) it is connected to (i.e., call a method on their service interface), it fetches the matching components according to the specific "connected to" property and unique identifiers. For example, a channel fetches all connectors that have the channel's unique identifier in their specific "connected to" properties, while a connector fetches the channel with the unique identifier that is saved in the connector's specific property. Even though the setup of the mesh typically hardly changes, properties need to be matched for every interaction between components, which is a rather expensive operation. Therefore, we make heavy use of OSGi's service tracker, which pro-actively tracks and caches matching components, similar to a proxy. The details of setting up and persisting component bindings with unique identifiers, service properties, and service trackers are implemented in ExoP's management bundle and hidden behind its *WiringController* service interface, providing straightforward methods to connect and disconnect components.

5 Evaluation

We have ported the MXQuery engine and the Linear Road benchmark (LRB) implementation presented by Botan et al. [7] to ExoP using partial assimilation (see Sect. 3.1). The evaluation measures the overhead of the exoengine approach, demonstrates the features and benefits gained by porting MXQuery to ExoP (i.e., capability for dynamic modifications and extensions, distributed operation, federation with different SPEs), and describes the porting effort.

5.1 The Linear Road Benchmark

The Linear Road benchmark [5] is a well established benchmark for stream processing systems. It simulates variable tolling based on traffic conditions on a fictitious linear city, consisting of a number of straight, 100 mile long, parallel highways. The input to the system increases in rate during a full, three hour run of the benchmark, and consists of car position reports and requests for toll information and balance reports. The output of the system consists of accident alerts, tolls, and balance reports. A system running the Linear Road benchmark must emit an output tuple (e.g., balance report) within at most five seconds of when the last input tuple that causes the output to be generated (e.g., request for balance report) enters the system. The number of concurrent highways (in units of .5 for separate directions of highways) that a system can cope with constitutes its load factor L. Due to the coarse granularity of this load factor L, we will fix L across comparable experiments and examine average tuple latencies a as measure of performance impact.

5.2 Experiment Setup

Unless noted differently, the experiments were run on a machine with a single Core i5-750 CPU (quad-core, 2.66 GHz) and 8 GB RAM, running the 64bit version of FreeBSD 8.2 configured to use the CPU's TSC register as timecounter. We use OpenJDK 6b22 as Java runtime with maximum heap size set to 5 GB.

The numbers presented refer to the toll alerts output of the Linear Road benchmark. They average 4 repetitions of a full, 3-hour-long run with an input load of $L = 5.0$.

5.3 Porting MXQuery and Linear Road

Applications for the MXQuery system typically consist of multiple instances of the MXQuery engine and MXQuery's storage implementations. Glue code creates and links them to each other to form the final application. Every instance of the engine executes one specific XQuery query. A query is compiled into a query graph consisting of multiple operators that potentially have small, internal, implicit state and/or buffers between each other. The storage instances provide windowing or persistent storage, and serve as explicit buffers for (intermediate) results between instances of the engine.

The MXQuery engine was wrapped as a π-slet and the storage implementations as channels. The rich interface between engine and storage, which, for example, allows for index-based access to data in the storage, remains in use as an extension to the basic

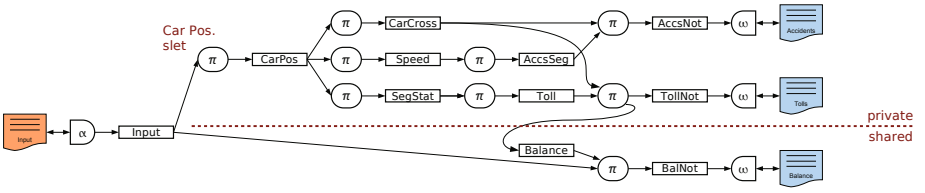


Fig. 7. Linear Road benchmark implementation on ExoP

interfaces of ports and channels. Furthermore, the code that loads the input file of the Linear Road benchmark and feeds it to the benchmark as well as the code that writes the result files was wrapped as α- and ω-slet, respectively. The code that sets up the Linear Road benchmark by creating instances of all involved components (data loader and writer, storage, MXQuery engine), assigning queries to the instances of the engine, and linking these components to each other with custom glue code was turned into an application builder. The application builder registers the slets (MXQuery, data loader, data writer) and the channel implementations with ExoP. It then instructs the platform to create respective instances thereof and to connect them. The queries that each instance of the engine has to execute as well as input and output file names are passed to the slets through ExoP's configuration mechanism. Once the application builder has completed setting up the Linear Road benchmark implementation in ExoP (as illustrated in Fig. 7), processing starts. With the exception of the daily expenditures query[1], the implementation is the same as the original one presented by Botan et al. [7] and consists of 9 instances of the MXQuery engine, each processing a different query.

5.4 Overhead of the Exoengine Architecture

Since every additional layer potentially adds overhead to a system, we first measure the overhead incurred by the modular and dynamic design employed by our platform. Figure 8 compares the original implementation of the Linear Road benchmark ("Without") with the ExoP version ("With"). Both implementations can handle the load well and the overhead added to the average processing time of tuples is negligible (147.90 ms vs. 148.92 ms). Table 1 provides additional details about the experiment runs. It counts output tuples grouped by processing time (bins of 1 second) in the 2^{nd} and 3^{rd} column.

5.5 Replacing an Slet at Runtime

The exoengine architecture encapsulates entities like operators and buffers and uses loose coupling between them, which enables dynamic changes to the processing mesh. We demonstrate this feature by replacing the car positions slet in the LRB workflow after 1 hour of the 3-hour-long run of the benchmark with a native implementation, while

[1] We have removed the daily expenditures query from both the original and the ExoP implementation due to its negligible impact on performance and the effort required to deploy the historical data.

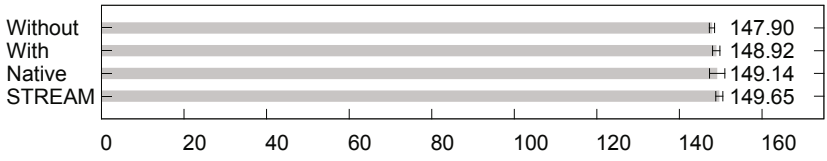


Fig. 8. Average tuple latencies [ms]

the benchmark is running. The query executed by the car positions slet filters car position reports from the input and forwards them to the upper part of the workflow shown in Fig. 7. Our native implementation performs the same functionality directly in Java instead of using the MXQuery engine and can be seamlessly plugged into the processing mesh. Replacing one instance of the MXQuery slet with a native implementation happens almost instantaneously and without adverse effects to the benchmark, as the average latencies "With" compared to "Native" in Fig. 8 show. Table 1 again provides additional details of the corresponding experiment runs in the 3^{rd} and 4^{th} column.

5.6 Distributed Deployment

The encapsulation of entities like operators and buffers behind well-defined interfaces abstracts from concrete implementations and, thus, allows to transparently introduce network communication between components. We demonstrate this feature by distributing the centralized MXQuery engine across multiple machines and scaling the load factor L of the Linear Road benchmark using data partitioning at the level of highways.

We use 16 cluster nodes on a switched gigabit ethernet. Each node has two Xeon L5520 CPUs (quad-core, 2.26 GHz) and 24 GB RAM. They run Ubuntu 10.04 64 bit and Oracle's JDK 6u22 with maximum heap size set to 5 GB. The maximum possible load for a single node is $L = 4.5$. However, applications designed for the exoengine architecture can easily be transformed into distributed systems by introducing remote invocations between components running on different machines. For the Linear Road benchmark, we chose the partitioning depicted by the dashed line in Fig. 7. Every node handles the traffic of 4.0 highways (upper part of the figure). The toll balance (lower part of the figure) runs only on one node (the master) and the other nodes (slaves) update the shared toll store through a transparent remote service invocation, implemented by a remote connector. We fixed $L = 4.0$, as this configuration spared enough capacity on the master node for the shared part of all experiment setups up to 16 nodes.

Figure 9 shows how we scale the aggregate load (left y-axis) linearly with the number of nodes (x-axis) and the effect on the mean latency of all tuples (right y-axis). For every node we add we can process another 4.0 highways, resulting in $L = 64.0$ being processed on 16 nodes. The overall tuple latency only increases significantly for the first few added nodes and then flattens out. The impact of the distributed setup on the latency is twofold. First, updates to the toll store on the master by the slaves are synchronous in the current implementation and, thus, block local processing on the slaves until the update has completed successfully. The impact of this constant overhead on the total average tuple latency is proportional to the number of slaves ($0, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \ldots$) and

Table 1. Tuple count grouped by processing time

Time	Without ExoP	With ExoP	Native after $1h$	STREAM
$[0, 1)$ s	11 329 044.00	11 333 460.50	11 333 349.00	11 324 352.00
$[1, 2)$ s	52 042.75	48 511.00	48 009.75	53 378.75
$[2, 3)$ s	14 830.25	14 367.75	14 373.00	17 216.75
$[3, 4)$ s	1 755.00	1 332.75	1 940.25	2 724.50
$[4, 5]$ s	0.00	0.00	0.00	0.00
$(5, \infty)$ s	0.00	0.00	0.00	0.00

results in the steep increase when adding the first slaves. Second, the load of processing updates to the toll store on the master node increases with every slave added. This results in slightly increased latency on both the master node's local traffic processing as well as responses to slaves' update requests to the toll store and thus their local traffic processing as well. The small, steady increase in latencies reflects this effect.

The experiment shows that we can use ExoP to scale out an application that was based on a centralized engine. We were able to linearly scale up the load of the Linear Road benchmark implementation on MXQuery with the number of nodes. The communication between nodes happens through ExoP's communication system, used by remote connectors which appear to the MXQuery engine slet like a connector to a local store. We chose a synchronous and straightforward implementation of the remote connector to capture all impacts of network communication. Depending on application semantics, asynchronous remote connectors, with queues, can be used to cut latency.

5.7 Developing with the Exoengine

Since it is not possible to provide universally valid, hard numbers on the effort that is needed to implement certain functionality in software, we provide at least an inkling of the overhead and savings when implementing using the exoengine architecture.

The native car positions slet consists of two classes. One class implements *SletMain* (see Sect. 4.4 for details) and the other class implements the actual filtering functionality. The main class consists of 55 lines of source code, out of which all but 9 lines have been generated from the *SletMain* interface. Packaging an slet implementation for ExoP only requires the addition of one attribute to the manifest of the JAR file.

SletMain of the MXQuery slet consists of 300 lines of code. It uses the original MXQuery codebase with a set of interfaces and small helper classes, which are again reused by the data loader and writer slets, the channels, and the native car positions slet.

The overhead of implementing an slet is moderate and limited to implementing the basic interfaces for management and data exchange. Implementing buffers as channels follows the same pattern and is equally simple. For the distributed experiment, we only had to change certain data types of the MXQuery engine to implement the *Serializable* interface so that we could ship instances to other machines. The remote invocations along the boundaries of OSGi services happen transparently with R-OSGi.

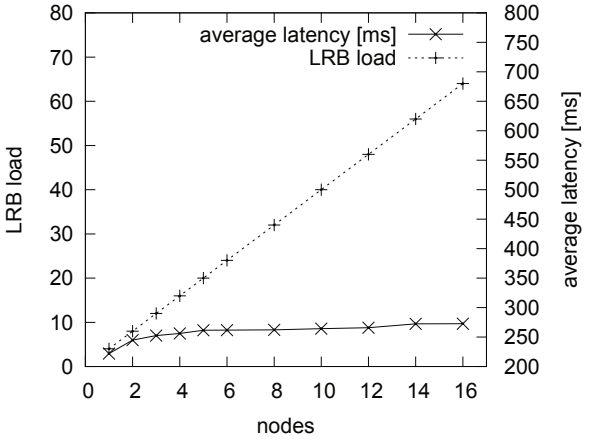


Fig. 9. Scale-out of MXQuery-based LRB implementation on ExoP

The MXQuery and the STREAM engines [20], storage implementations of MXQuery, data loaders and writers were ported by us and are reusable as a library. Additional applications for these engines can therefore be deployed right away.

5.8 Heterogeneity

ExoP enables the federation of heterogeneous stream processing entities on a common platform. We demonstrate this feature by combining MXQuery and STREAM into one application. Figure 10 illustrates the modified setup. We replaced the sink for the toll notifications with an slet that converts flat XML fragments into binary relational tuples (*X2R*). The STREAM engine (*SE*) processes the toll notifications according to its assigned query and emits the results to a sink that writes relational tuples to a file (*RS*).

The two engines used in this setup differ in terms of the query model (XQuery vs. CQL), data model (XML vs. relational tuples), implementation language (Java vs. C++), and processing model (purely pull-driven vs. thread-driven, pull-based input and push-based output). Each engine processes queries in its native format. No query transformation or translation takes place and the strengths of each engine and its specific query dialect are retained. Data is consumed and emitted by each engine in its native format as discrete items. Conversion slets, like slet *X2R* in Fig. 10, convert between different data formats. They can be built using existing conversion tools and libraries. Federation of applications written and running on different engines typically happens at few, well-defined interaction points. Therefore, the effort to deploy conversion slets or provide custom conversion slets for proprietary data formats is manageable.

We measure the overhead introduced by adding the STREAM engine to the MXQuery-based benchmark by running this modified setup of the benchmark and simply passing tuples through the STREAM engine using `select * from S` as query, where `S` corresponds to the *TollNotR* input channel. Connecting STREAM to the processing mesh and passing tuples through it adds only $0.5\,\%$ overhead to the average

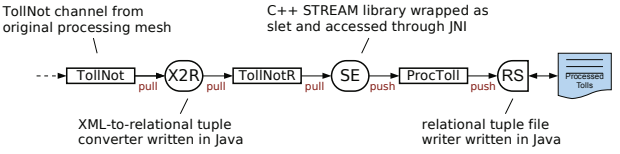


Fig. 10. STREAM engine attached to toll notifications

latencies (compare "With" and "STREAM" in Fig. 8) and the benchmark runs well within limits, as can be seen in the 5^{th} column in Table 1.

The seamless integration of the pull-driven MXQuery engine written in Java with the pull/push-driven STREAM engine written in C++ demonstrates the suitability of the architecture for supporting different processing models and the composition of heterogeneous engines into new applications.

6 Related Work

The fundamental difference between the research [2,20,9] and commercial products [27, 29, 4, 16] mentioned in the introduction and our exoengine architecture is that it is not yet another SPE. Rather, it provides a platform for facilitating application development, deployment, integration, and management of existing or new-to-be-built SPEs.

Related work on distributed SPEs focused on functionalities like load management (e.g., [1,24]), fault tolerance and high availability (e.g., [1]), integration with sensor networks (e.g., [14, 3]), and performance tuning (e.g., [17]). These systems mostly target fixed cluster-based settings, where the dynamic wide-area architectural requirements that we consider, such as loose coupling and heterogeneity of components, as well as flexibility of deployment and reconfiguration, were not considered as equally critical. Closer to our work, the XFlow Internet-scale distributed stream processing system proposes a loosely-coupled architecture for query deployment and optimization, focusing on an extensible cost model [23]. XFlow does not provide any abstract programming models or techniques for building, hosting, or porting various SPE components.

There are a few platforms proposed for facilitating the development of stream-based applications, such as System S, Auto-Pipe, MaxStream, or PIPES. System S [17] includes a distributed runtime platform that facilitates dynamic stream processing. The platform pursues similar goals in terms of deployment as exoengine does but is less extensible due to its focus on the ecosystem of System S, which includes a language and run-time framework (SPADE) and a semantic solver (MARIO). In constrast, exoengine is an independent, pure middleware approach, and as such, is usable for many different SPEs. Auto-Pipe [8] is a development environment for streaming applications executing on diverse computing platforms consisting of a hybrid of multicore processors, GPUs, FPGAs, etc. The authors propose a coordination language X and a compiler that maps X programs into the native languages of the underlying platforms so that parts of applications can be run on the platforms that will provide the highest performance for them. This work focuses on diverse hardware platforms, whereas we focus on diverse SPEs.

The MaxStream architecture [6], on the other hand, integrates heterogeneous SPEs and databases behind a common declarative query interface, but without considering the lower-level virtualization and flexible wide-area deployment issues that our exoengine architecture tackles. PIPES [18] is a flexible and extensible infrastructure that provides fundamental building blocks (including runtime components like a scheduler, memory manager, and query optimizer) to implement a stream processing system for the Continuous Query Language (CQL) and a specific operator algebra. In contrast, the exoengine approach proposes a generic model and platform to host and execute a variety of different and independent stream processing systems, which are not required to share a common query language, algebra, implementation language, or runtime components, but can still share them where appropriate.

The importance of elastic stream processing has also been recognized by related work recently [26]. This work focuses on elastically scaling the performance of individual streaming operators on multicore machines, whereas our work provides a more general architecture for distribution and a platform that can also serve as basis for elastic stream processing. Yahoo's S4 [21] provides an architecture and platform for processing streaming data similar to MapReduce [10] for stored data, and the similar key property of a specific, simple processing model that enables automatic parallelization and deployment on a large number of machines. StreamCloud [15] is a middleware layer that sits on top of streaming engines and focuses on how to parallelize continuous queries by splitting them into subqueries and distributing them to nodes. The exoengine approach provides a platform that hosts different streaming engines and could benefit from StreamCloud by integrating it as application builder. Lastly, in our XTream project [11], we explore how an exoengine-like platform and stream processing in general can facilitate personal information processing and dissemination at global scale.

Publish/subscribe systems also provide mechanisms and an infrastructure to disseminate and filter data from sources to sinks [13]. They decouple senders from receivers by topics, which can be modeled by channels in our architecture. However, they do not support sophisticated in-network data processing, distributed operation in a peer-to-peer manner, access to intermediate results, or in-network storage, as stream processing with the exoengine does.

7 Discussion and Outlook

In this paper, we have proposed a new architecture for implementing data stream processing applications by virtualizing components of stream processing systems and deploying them on a common middleware platform. While being radical in its puristic approach inspired by the exokernel architecture, the non obtrusive nature of the approach—it does not dictate a specific query language, algebra, operator implementation, or scheduling model—allows to leverage any existing stream processing system and its particular strengths. In contrast, yet another concrete implementation of a stream processing system would require a much more radical reimplementation of existing applications. The exoengine architecture defines the fundamental elements of stream processing (slets/operators, channels/buffers) using extensible interfaces to allow rich interaction between specific slet and channel implementations of a particular system, while retaining basic data exchange capabilities with other systems.

Depending on the granularity of the integration of streaming systems with the exoengine platform (see Sect. 3.1), the benefits range from the automatic management of deploying and executing an encapsulated application and its federation with applications for other engines (through its ultimate inputs and outputs) to the reuse of engines or individual operator implementations and the ability to replace them with a different implementation at runtime, as demonstrated in Sect. 5.5. The architecture transparently provides data transport in a distributed setup and allows to run centralized engines in a distributed setting, as demonstrated with the scale-out experiment in Sect. 5.6.

Finally, we have provided brief conceptual instructions for building an exoengine platform using SOA in Sect. 2.3 and discussed concrete implementation aspects as well as showed a concrete prototype implementation in Sect. 4. The prototype confirms that the dynamic nature and additional indirections (ports, connectors) can be implemented efficiently with negligible overhead, as validated in Sect. 5.4.

Future work includes automatic state capturing of slets for migration (currently slets need to implement serialization and deserialization of internal state to allow migration); deriving common, generic channels and slets (e.g., round robin distributor) and providing them as a base library (similar to the platform providing common functionality for deploying, running, and managing configuration and state); and extending the exoengine's area of application to elastic/cloud computing—the holy grail of dynamic operation, automated management, and distributed deployment.

Acknowledgments. Part of this work was funded by the Swiss National Science Foundation SNF ProDoc program and by the Microsoft ICES initiative.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: CIDR (2005)
2. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: A New Model and Architecture for Data Stream Management. The VLDB Journal 12(2), 120–139 (2003)
3. Aberer, K., Hauswirth, M., Salehi, A.: A Middleware for Fast and Flexible Sensor Network Deployment. In: VLDB (2006)
4. Ali, M.H., Gerea, C., Raman, B.S., Sezgin, B., Tarnavski, T., Verona, T., Wang, P., Zabback, P., Ananthanarayan, A., Kirilov, A., Lu, M., Raizman, A., Krishnan, R., Schindlauer, R., Grabs, T., Bjeletich, S., Chandramouli, B., Goldstein, J., Bhat, S., Li, Y., Di Nicola, V., Wang, X., Maier, D., Grell, S., Nano, O., Santos, I.: Microsoft CEP server and online behavioral targeting. Proc. VLDB Endow. 2, 1558–1561 (2009)
5. Arasu, A., Cherniack, M., Galvez, E.F., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear Road: A Stream Data Management Benchmark.. In: VLDB (2004)
6. Botan, I., Cho, Y., Derakhshan, R., Dindar, N., Haas, L., Kim, K., Lee, C., Mundada, G., Shan, M., Tatbul, N., Yan, Y., Yun, B., Zhang, J.: Design and Implementation of the MaxStream Federated Stream Processing Architecture. Tech. Rep. TR-632, ETH Zurich Department of Computer Science (2009)
7. Botan, I., Kossmann, D., Fischer, P.M., Kraska, T., Florescu, D., Tamosevicius, R.: Extending XQuery with Window Functions. In: VLDB (2007)

8. Chamberlain, R.D., Franklin, M.A., Tyson, E.J., Buckley, J.H., Buhler, J., Galloway, G., Gayen, S., Hall, M., Shands, E.B., Singla, N.: Auto-Pipe: Streaming Applications on Architecturally Diverse Systems. IEEE Computer Magazine 43(3), 42–49 (2010)
9. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In: CIDR (2003)
10. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI (2004)
11. Duller, M., Alonso, G.: A lightweight and extensible platform for processing personal information at global scale. Journal of Internet Services and Applications 1, 165–181 (2011)
12. Engler, D.R., Kaashoek, M.F., O'Toole Jr., J.: Exokernel: An Operating System Architecture for Application-Level Resource Management. In: SOSP (1995)
13. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The Many Faces of Publish/Subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
14. Franklin, M.J., Jeffery, S.R., Krishnamurthy, S., Reiss, F., Rizvi, S., Wu, E., Cooper, O., Edakkunni, A., Hong, W.: Design Considerations for High Fan-In Systems: The HiFi Approach. In: CIDR (2005)
15. Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Valduriez, P.: StreamCloud: A Large Scale Data Streaming System. In: ICDCS (2010)
16. IBM InfoSphere Streams, http://www-01.ibm.com/software/data/infosphere/streams/
17. Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., Venkatramani, C.: Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In: SIGMOD (2006)
18. Krämer, J., Seeger, B.: PIPES - A Public Infrastructure for Processing and Exploring Streams. In: SIGMOD (2004)
19. Leslie, I., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R., Hyden, E.: The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. IEEE Journal on Selected Areas in Communications 14(7), 1280–1297 (1996)
20. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., Varma, R.: Query Processing, Approximation, and Resource Management in a Data Stream Management System. In: CIDR (2003)
21. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed Stream Computing Platform. In: ICDMW (2010)
22. OSGi Service Platform, http://www.osgi.org/
23. Papaemmanouil, O., Çetintemel, U., Jannotti, J.: Supporting Generic Cost Models for Wide-Area Stream Processing. In: ICDE (2009)
24. Pietzuch, P.R., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.I.: Network-Aware Operator Placement for Stream-Processing Systems. In: ICDE (2006)
25. Rellermeyer, J.S., Alonso, G., Roscoe, T.: R-OSGi: Distributed Applications Through Software Modularization. In: Cerqueira, R., Pasquale, F. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 1–20. Springer, Heidelberg (2007)
26. Schneider, S., Andrade, H., Gedik, B., Biem, A., Wu, K.L.: Elastic Scaling of Data Parallel Operators in Stream Processing. In: IPDPS (2009)
27. StreamBase Systems, Inc., http://www.streambase.com/
28. Tatbul, N.: Streaming data integration: Challenges and opportunities. In: NTII (2010)
29. Truviso, Inc., http://www.truviso.com/
30. Zeitler, E., Risch, T.: Scalable Splitting of Massive Data Streams. In: Kitagawa, H., Ishikawa, Y., Li, Q., Watanabe, C. (eds.) DASFAA 2010. LNCS, vol. 5982, pp. 184–198. Springer, Heidelberg (2010)

Leader Election for Replicated Services Using Application Scores

Diogo Becker[1], Flavio Junqueira[2], and Marco Serafini[2]

[1] Technische Universität Dresden,
Computer Science Department,
Dresden, Germany
diogo@se.inf.tu-dresden.de
[2] Yahoo! Research, Barcelona, Spain
{fpj,serafini}@yahoo-inc.com

Abstract. Replicated services often rely on a leader to order client requests and broadcast state updates. In this work, we present POLE, a leader election algorithm that select leaders using application-specific scores. This flexibility given to the application enables the algorithm to tailor leader election according to metrics that are relevant in practical settings and that have been overlooked by existing approaches. Recovery time and request latency are examples of such metrics. To evaluate POLE, we use ZooKeeper, an open-source replicated service used for coordinating Web-scale applications. Our evaluation over realistic wide-area settings shows that application scores can have a significant impact on performance, and that just optimizing the latency of consensus does not translate into lower latency for clients. An important conclusion from our results is that obtaining a general strategy that satisfies a wide range of requirements is difficult, which implies that configurability is indispensable for practical leader election.

Keywords: leader election, replicated services, fault tolerance, performance.

1 Introduction

Leader election is a fundamental primitive often used in practical systems, such as ZooKeeper [12] and Chubby [4], which are stateful middleware services used for coordination tasks of Web-scale applications. Given the size and extent of such applications, it is critical to prevent faults from bringing them to a halt, so both services use replication for masking faults and rely upon a primary server, *i.e.*, a leader, to propose state updates and to disseminate them using an atomic broadcast protocol. Consequently, the ability to elect a leader for the replicated service is critical to ensure progress.

In replicated services like ZooKeeper and Chubby, the leader performs more work than other servers, since it processes more messages and generates state updates. Many existing leader election algorithms rely on the identifiers of servers

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 289–308, 2011.

to select a leader [1,9,11]. While designing ZooKeeper, however, one initial requirement regarding leader election was the ability to elect the server with the longest history of state updates among a quorum of servers. Such a server only has to push missing state updates to follower servers instead of pulling missing updates first. Because servers present comparable performance in data center deployments, such a leader not only enables faster recovery, but also provides the same performance while broadcasting as any other server would provide.

Over time, however, we encountered settings in which ZooKeeper servers either were running on heterogeneous hardware or presented different connectivity to other servers and application clients. Deployments spanning multiple data centers are important examples of such settings, where applications often present disaster-tolerance requirements. In some deployments, most clients are in one data center and electing a leader in the data center where most client requests arrive minimizes the request latency observed by clients. For some applications, such an optimization can be even more important than optimizing recovery time, as previously done by ZooKeeper, or the time required to terminate consensus, as with existing leader election algorithms such as the latency-aware algorithm of Santos *et al.* [17].

Although a number of leader election algorithms exist in the literature, there is no algorithm to our knowledge that can be easily adapted to specific constraints. This observation led us to reason about how to elect servers with properties other than the history length, and to use a generic *score* computed at runtime to classify servers according to *application-specific* properties.

In this work we propose an algorithm that takes a generic score as input to order servers during election, and we call it POLE (Performance-Oriented Leader Election). POLE is *configurable*, since it provides to an application the ability of selecting the desired properties of a leader server. To enable configurability, we implement application-specific scoring functions in an oracle module external to POLE. Before starting leader election, the local POLE module of a server queries its oracle to assign itself a score. Oracles compute scores using information either collected during regular operation or explicitly measured for estimating performance. Designing application-specific score oracles is simpler than implementing a new leader election that suits the application needs. Simplicity also comes from having each server assigning a score only to itself. After querying the oracle, servers share their scores, encapsulated in election messages, and try to elect the server with the highest score. POLE simply broadcasts scores and does not require reaching agreement on the ordering of servers before starting leader election, as required in the work of Sampaio and Brasileiro [16].

We show the flexibility and simplicity of our approach by implementing score oracles that optimize important metrics arising from real-world applications and were not considered by existing leader election algorithms: *mean request latency*, the mean time required for clients to complete a request; *worst-case request latency*, a metric often used to specific application requirements to the ZooKeeper service; and *recovery time*, the time it takes for a new leader to start operating again after the failure of the previous leader.

For evaluation, we implemented a prototype using the ZooKeeper code base[1] and emulated wide-area systems to investigate the performance of different application requirements. The results show that our request latency metrics can in some cases significantly diverge from consensus latency, which is the metric optimized by leader election algorithms such as the one of Santos *et al.* [17]. Depending on the particular setting, selecting an appropriate scoring function can elect leaders that provide up to 50% lower request latencies than arbitrarily elected leaders, while recovery-oriented oracles can elect leaders that provide minimal recovery times. Our results show however that obtaining a *general* strategy that fulfills multiple requirements is difficult.

The following list summarizes our contributions in this paper:

- We present POLE, the first leader election algorithm that is explicitly designed to enable application-specific performance configurability;
- We propose novel leader election oracles optimizing request latency and recovery time, metrics that cannot be directly optimized using the leader election algorithms proposed in the literature;
- We evaluate these oracles under a set of emulated wide-area settings and discuss trade-offs that operators deploying POLE might encounter.

Roadmap. The remainder of this work is structured as follows. Section 2 presents common application metrics and propose scores that approximate these metrics. Section 3 presents the algorithm and oracles using our scores. POLE and the oracles are evaluated in Section 4. We discuss further extensions in Section 5 and related work in Section 6. Finally, we conclude in Section 7.

2 Application Scores

In this section, we present several application-specific scoring functions. A scoring function $\theta(p)$ is a function that maps identifiers of servers (*id*s) to values called *scores*. We use the term θ score instead of scoring function $\theta(p)$ whenever the *id* of server p is clear from the context. Our scoring functions are based on two important metrics motivated by requirements that real-world applications impose on coordination services: recovery time and latency of (write) requests as perceived by clients. In particular, we focus on wide-area network (WAN) deployments, spanning multiple data centers. Such deployments are typical for applications with disaster-tolerance requirements. POLE, however, supports other scoring functions not explored in this work such as pre-defined preference lists. Table 1 summarizes our scoring functions.

2.1 Background: Replicated Coordination Services

A coordination service, such as Chubby and ZooKeeper, enables clients to interact through a shared data tree of simple small files; for instance, ZooKeeper

[1] `http://zookeeper.apache.org`

Table 1. Scores for server p

Score	Symbol	Description
maximum *zxid*	θ_z	length of the local history stored by p
consensus latency	θ_c	consensus latency if p becomes leader
request rate	θ_r	rate of client requests sent to p
mean request latency	θ_l	mean request latency if p becomes leader
worst-case request latency	θ_w	worst-case request latency if p becomes leader

by default does not allow files (znodes) larger than 1 MB. This data structure is replicated across servers of an ensemble to ensure availability and durability. To access the service, clients initiate sessions and manipulate these files through a file-system-like API. Although simple, this API is powerful enough to create complex synchronization mechanisms.

Coordination services are designed to be a consistent, reliable component of larger distributed applications. We consequently assume real deployments are properly provisioned (the service rarely saturates). Different from Chubby, ZooKeeper clients can connect to any server and this server processes locally read requests, thus avoiding the cost of running atomic broadcast. Deployments of coordination services consist typically of ensembles of 3 to 7 servers, depending on the application requirements. In deployments of these services, process and link failures are typically infrequent.

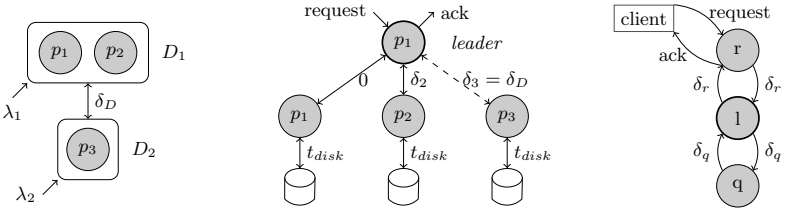
2.2 Request Latency

In wide-area settings, links often have different latencies, and the request latency perceived by the client application can be adversely affected by slow links. Consider the example with an ensemble of 3 servers deployed in two data centers D_1 and D_2 as depicted in Fig. 1a. Let δ_D be the latency of the links between any server in D_1 and any server in D_2. In a wide-area setting, data centers are geographically separated, so the link latency inside a data center, *e.g.*, between p_1 and p_2, is typically much lower than between data centers, *e.g.*, δ_D.

In leader-based atomic broadcasts protocols, the leader orders client requests, and in systems like ZooKeeper and Chubby, they also process them generating state updates that the leader broadcasts to followers (a leader server is also a follower). Following ZooKeeper terminology, each of these state updates is a *transaction*. When a server is elected leader, *e.g.*, server p_1, the links can be represented as in Fig. 1b, where δ_i denotes the latency of the link between server p_1 and server p_i. The leader can only acknowledge a request to the client after a *quorum* of servers accepted the respective transaction, by logging it to stable storage and replying. In this work, we consider a quorum to be a majority of servers in the set N of all servers of the system, ensuring that every two quorums have a common server.

The elapsed time between a broadcast by p_1 and the receipt of a quorum of replies is the *consensus latency* of p_1.[2] Let server l be a leader (or candidate)

[2] For our purposes, atomic broadcast and consensus are equivalent terms.



(a) 3 servers in 2 data centers, high link latency δ_D (b) Links between leader and followers with latencies (c) Total request latency

Fig. 1. Two data centers, leader-follower links, and latency of a request

and q be the server with *slowest* link to l in the quorum of servers with the *fastest* links to l, *i.e.*, q is the server with the $(\lfloor |N|/2 \rfloor + 1)^{\text{th}}$ slowest link to l. In the example, $l = p_1$, $q = p_2$, and $\delta_q = \delta_2$. The score θ_c of a server l is its consensus latency assuming it is the leader, *i.e.*, $\theta_c(l) = 2 \times \delta_q + t_{disk}$, where t_{disk} is the time it takes a server to store the transaction in stable storage. Modern disks have a write cache that improves significantly the performance of writes to disk, and enables short latency values. With such a buffer on, which is a typical choice in production, and persisting updates to disk, ZooKeeper is able to process operation in a few milliseconds including local area network latency. For our purposes, we can consequently ignore the disk latency. Additionally, we assume in the following discussion that latency of a link is roughly the same in both directions and relatively constant despite of spikes.

Electing the server with the fastest links to a quorum minimizes consensus latency, but not necessarily request latency. With ZooKeeper, as with other replicated services, the leader does not always directly receive all client requests. A follower r also receives them on behalf of the leader and forwards them to the leader l, keeping a session to the client (see Fig. 1c). Once the leader performed consensus on the transaction, the leader sends an acknowledgment to the follower that forwarded the request, which in turn sends an acknowledgment to the client. The *request latency* is thus the sum of the hop from follower r to l and back plus the consensus latency of leader l. In the example, if only the consensus latency is considered, then servers p_1 or p_2 are the best leaders because they have a fast link to a quorum, *i.e.*, to themselves and to another server in the same data center. However, the request latency actually observed by the client also needs to include the additional communication step from the replica to the leader, so the "slow" server p_3 could be the best leader if most of the requests arrive on it. Note that we assume clients first connect to replicas in their data centers, so the latency client-replica is negligible compared to WAN latencies.

We now define scoring functions (scores) that target short request latency. Let λ_T be the total request rate of the system. The request rate arriving in data center D_i or on server p_i is represented by λ_i. If not clear from the context, we explicitly indicate to which one λ_i refers. The scoring function $\theta_r(p)$ is the *mean request rate* received by p from clients, *i.e.*, an approximation of λ_i.

The scoring function $\theta_c(p)$ represents the consensus latency provided by p if it were the leader. The scoring function $\theta_l(p)$ represents the *mean request latency* provided by server p if it were the leader, *i.e.*, the consensus latency score θ_c plus the round-trip time between p and each follower r weighted by the follower's θ_r:

$$\theta_l(p) = \theta_c(p) + \frac{1}{\lambda_T} \sum_{r \in N'} \theta_r(r) \times \delta_{r,p} \tag{1}$$

The scoring function $\theta_w(p)$ represents the *worst-case request latency* provided by server p if it were the leader, *i.e.*, the consensus latency score θ_c plus the round-trip time to the follower r with the greatest $\delta_{r,p}$:

$$\theta_l(p) = \theta_c(p) + 2 \times max(\{\delta_{r,p} | r \in N'\}) \tag{2}$$

In both equations $N' \subseteq N$, N' contains no server which is suspected to be crashed; and, if the server follows a leader l, then $l \notin N'$. The latter restriction is because the score should represent the performance of the candidate in the case the leader crashes. Section 3 explains how these scores are used in the election.

2.3 Recovery Time

The second practical metric we use to evaluate a leader is the recovery time, since it directly influences the down time of a service. For the purposes of this work, we define *recovery time* of a replicated service to be the time the new leader takes to learn from a quorum of followers the chosen transactions it has not seen, if any, and propagating the transactions to servers that are missing them. After a quorum is synchronized by persisting transactions to stable storage and applying to the state, the leader can start processing new transactions.

In ZooKeeper, a transaction encompasses an idempotent state update generated by executing the client's request and an increasing transaction identifier called *zxid* [12]. The total order enforced by ZooKeeper is consistent with the *zxid* order. As in any practical replicated service implementation [4,12], the leader performs multiple consensus instances in parallel. Because servers do not proceed in lock-step, and the leader only requires a quorum of servers to guarantee progress, at any point in time different followers may have accepted a different number of transactions due to a number of factors. If a server is powered on after crashing or after being powered off, then its state can be arbitrarily behind. Resource contention and traffic spikes might also cause a particular server to lag behind. As a consequence, the amount of transactions a new leader has to learn to synchronize with other servers on recovery can be arbitrarily large. The recovery time directly depends on this amount of data and on how fast the links between leader and followers are. During the time the service is recovering, no new request is accepted, and consequently, minimizing this amount of time is critical for availability.

Using the last accepted *zxid* directly in the election is a sensible approach. In fact, this is the scoring function ZooKeeper election currently implements; the last accepted *zxid* of a server is its θ_z score. By electing the server with the

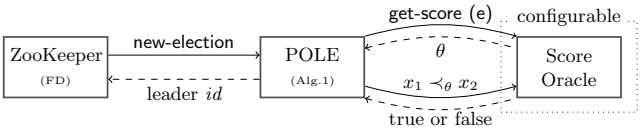


Fig. 2. Module interactions

highest θ_z, the leader does not have to copy missing transactions from other servers, thus making the time to recover negligible in local-area deployments, since it is essentially the time the leader takes to learn from a quorum of servers their last accepted *zxid*, but no transaction content. The server with the highest θ_z is called the *fattest* server.

3 The POLE Algorithm

POLE achieves configurability by allowing the operator to select on deployment a score oracle to order the processes[3] in an election round. A score oracle consists of a function get-score, which implements a scoring function $\theta(p)$ that assigns a score to the process; and a total order relation $\prec_\theta$ over the set of scores (a partial order relation is also sound with POLE, but the relations we use in this work are all total orders). For example, one possibility is to use the $\theta_z(p)$ scoring function, *i.e.*, to use *zxid*s as scores, and the total order over *zxid*s as the ordering relation. Figure 2 shows the interaction between the modules of our system.

Our POLE implementation in ZooKeeper breaks the leader election into two tasks: (unreliable) leader selection and (unreliable) failure detection. POLE, together with a score oracle, implements the first task, whereas ZooKeeper implements the second task. Both together implement an Ω leader election oracle [7]. When a ZooKeeper process detects that the current leader crashed, it invokes POLE. Next, POLE starts an election by invoking the oracle, which in turn assigns a θ score to the local process. POLE then shares the θ score with the other processes. In timely runs, where messages are not lost the process with the highest score, ordered with the $\prec_\theta$, becomes the new leader. POLE terminates by returning to ZooKeeper the *id* of the selected process.

3.1 Failure Detection

Failure detection is implemented in ZooKeeper because it is a byproduct of its operation. When a process is elected leader, it opens TCP connections to other processes. As long as a leader has open connections with a majority of processes, it remains able to commit the operations it proposes, and progress is guaranteed. This condition holds as long as the leader has timely links to a quorum, that is, it is f-connected [15]. If the leader process fails to open enough connections, or if too many connections are terminated due to faults or asynchrony, the process

[3] In this section, we use the term process instead of server to agree with the literature.

abandons its leadership by closing all its connections and executes POLE to perform a new election. Similarly, when a process notices that it has disconnected from the leader, it executes POLE. Our election algorithm only requires leader-follower failure detection information, so it never requires ZooKeeper to open more connections than those needed by the atomic broadcast protocol itself. Processes only accept a connection from the process they believe to be the leader. Hence, at any given time *at most* one leader has open connections with a *majority* of processes. Consequently, if two processes are leaders simultaneously, eventually one of them drops leadership.

3.2 Leader Selection Algorithm

We now describe in detail the leader selection algorithm. The pseudo-code is shown in Alg. 1. We assume messages can be delayed or lost, but not corrupted; processes can start at different points in time, can crash, and can recover.

The core idea of the algorithm is very simple. Once new-election is called by ZooKeeper, process i starts the election by requesting its current θ score from its oracle with get-score. It then invokes start-epoch and broadcasts a propose message $\langle P, \mathit{epoch}, \theta, i\rangle$. Epochs are used to identify election instances. Whenever process i receives a greater proposal tuple from some other process j, it overwrites its proposal and repeats the broadcast. Proposals are totally ordered with $\prec_{\mathrm{p}}$, since it encapsulates multiple ordering relations: first ordering by epoch, then by θ using $\prec_{\theta}$, and finally by id in case breaking ties is necessary. In a timely run, non-crashed processes eventually converge to the same leader. Note that get-score receives the current epoch as an argument, since some scoring functions depend on the epoch, such as the rotating scheme described below.

As in any practical implementation, the algorithm has to return a leader at some point. If process i received proposals from *all* processes, then process i can elect the highest proposal immediately. Due to delayed or lost messages and process crashes, process i might receive proposals only from a subset of processes. In general, leader-based broadcast protocols require at least a quorum to achieve progress. Therefore, if process i receives proposals from a quorum, it starts election-timer (Line 11), which terminates the election upon timeout. Whenever a proposal is delivered, the procedure start-timer? is called to check these conditions. Upon timeout (Line 12), leader is set to the id of the process with the highest proposal, and new-election returns (Line 11) to ZooKeeper. The returned leader might, however, be a crashed process if it crashes after broadcasting its proposal. In such cases, the failure detection implemented in ZooKeeper will eventually restart the election by calling new-election.

Because processes can be started at different points in time, and can be temporarily disconnected from each other, a process j can try to start an election when other processes already follow a leader. When process i believes to know who the leader is, it replies proposals from process j with a vote (V) message containing the value of leader (Line 12). If process j receives a quorum of vote messages (Line 54) indicating the same leader in the same epoch, then process j starts also believing in its leadership and sets its leader and epoch variables, thus

Algorithm 1. Selecting a leader in process i

```
1  import get-score, ≺_θ;
2  export new-election;
3  constant T;
4  timer election-timer, retry-timer;
5  init
6      leader ← ⊥;
7      epoch ← 0;
8      proposal-set ← vote-set ← ∅;
9      stop election-timer;
10 procedure start-epoch
11     leader ← ⊥;
12     proposal-set ← vote-set ← ∅;
13     send ⟨P, epoch, proposal⟩ to all;
14 function new-election
15     t ← T;
16     epoch ← epoch+1;
17     proposal ← (get-score(epoch), i);
18     invoke start-epoch;
19     while leader = ⊥ do
20         start retry-timer;
21         while leader = ⊥
22           ∧ retry-timer < t do  nop;
23         if leader = ⊥ then
24             send ⟨P, epoch, proposal⟩ to
               all;
25         t ← t × 2;
26     return leader;
27 procedure start-timer? argument j
28     proposal-set ← proposal-set ∪ {j};
29     if |proposal-set| = |N| then
30         start election-timer with T + 1;
31     else if |proposal-set| > ⌊|N|/2⌋ then
32         start election-timer;
33 upon receive ⟨P, e, p⟩ from j do
34     if leader ≠ ⊥ then
35         send ⟨V, epoch, leader⟩ to j;
36         return;
37     if p ≺_p proposal then
38         send ⟨P, epoch, proposal⟩ to j;
39     else if e = epoch then
40         if  proposal ≺_p p then
41             proposal ← p;
42             send ⟨P, epoch, proposal⟩ to
               all;
43     else
44         epoch ← e;
45         proposal ← (get-score(epoch), i);
46         if proposal ≺_p p then
47             proposal ← p;
48         invoke start-epoch;
49     invoke start-timer? with j;
50 upon election-timer > T do
51     (θ, l) ← proposal;
52     leader ← l;
53     stop election-timer;
54 upon receive ⟨V, e, l⟩ from j do
55     vote-set ← vote-set ∪ (e, l, j);
56     if ∃S ⊆ vote-set :
57       ∀ (e1, l1, j1), (e2, l2, j2) ∈ S :
58       e1 = e2 ∧ l1 = l2
59       ∧ |S| > ⌊|N|/2⌋
60       ∧ (∃ (e4, l4, j4) ∈ S : j4 = l1)  then
61         epoch ← e1;
62         leader ← l1;
63         stop election-timer;
64     else
65         invoke start-timer? with j;
```

guaranteeing leader stability [1]. Line 2 checks if the leader replied with a vote message as well, what is important to avoid reelecting a crashed leader.

Whenever the election does not converge, *e.g.*, too many messages were lost and no quorum sent proposals to process i, a second timeout (retry-timer) is triggered and process i repeats its broadcasts (Line 1), exponentially backing off on each retry. Note that the timeliness implied by waiting for a quorum is not the minimal to enable atomic broadcasts [2,15]. Nevertheless, we have not yet observed the need for weaker timeliness assumptions.

3.3 Oracles

An oracle is the implementation of a scoring function and an ordering relation over set of scores. We implemented one oracle for each scoring function in Sec. 2. With history oracle, POLE elects the process with the highest θ_z. With request oracle, it elects the process that receives the largest volume of requests. With consensus oracle, it elects the process with the shortest consensus latency. With latency oracle, the process with the lowest mean request latency. And with

worst-case oracle, it elects the process with the lowest worst-case request latency. For comparison purposes, we also implemented an application-*unaware* oracle which represents a rotating leader selection based on the process identifier, a typical approach of existing protocols. The leader of the current epoch (also called round's leader) assigns itself a score of 1, while all other processes assign themselves a score of 0. Note that differently from traditional rotating elections, our oracle uses the same message pattern as any POLE election (broadcasts).

Different scores require different information to be acquired and transmitted between the processes. The θ_z and θ_r scores are already built into ZooKeeper and can be easily accessed by an oracle by querying the right components. To compute them, no information has to be transmitted between processes. In contrast, θ_l and θ_w require communication. The election module in ZooKeeper builds a clique graph encompassing all processes. These connections are encapsulated in a component called ConnectionManager, which implements links between the processes exclusively for the election. This component can be used by any oracle to transmit oracle messages if needed.

To compute θ_c, θ_l, and θ_w, each process i keeps a vector RTT_i with the mean round-trip times between i and all other processes. It sends "ping-pong" heartbeats to all nodes periodically via the ConnectionManager. Once a process crashes, it does not respond to heartbeats and is removed from RTT_i. The heartbeat period can be configured on a per deployment basis. Fairly long periods such as 100ms to 1s are sufficient in most cases (we use the default value of 1s). To calculate θ_c or θ_w, let $\overline{RTT}_i$ be RTT_i sorted in ascending order of values. The score $\theta_c(i)$ is the $(\lfloor |N|/2 \rfloor + 1)^{\text{th}}$ value of $\overline{RTT}_i$, and $\theta_w(i)$ is $\theta_c(i)$ plus the highest value in $\overline{RTT}_i$ (see Eq. 2). Note that we exclude the current leader from RTT_i because the scores are needed once the leader becomes faulty.

Finally, to compute θ_l, each process i needs to additionally keep a vector F_i with the value of $\theta_r(j)$ for all $j \in \Pi$. The sum of all elements in F_i approximates λ_T. The values of the scoring function $\theta_r(j)$ are transmitted in the heartbeat messages via the ConnectionManager. The final value is calculated by weighting RTT_i with the frequencies in F_i as in Eq. 1.

4 Experimental Evaluation

In this section, we evaluate the oracles described in Sec. 3.3. For readability, we often say an oracle provides low request latency instead of saying the server elected with this oracle provides low request latency.

4.1 Experimental Setup and Methodology

We performed all of our experiments on a cluster of 10 workstations. Each of them has 2 quad-core 2.0 GHz Xeon processors, 8 GB of RAM, a Gigabit Ethernet interface, and a 7200 RPM disk attached via SATA2. The running operating system is Debian GNU/Linux 5.0 with kernel version 2.6.26.

Table 2. End-to-end latency examples

source	target	mean rtt (ms)	sd (ms)
slac	caltech	9.88	0.10
tud	cern	20.75	0.11
slac	fnal	53.26	0.14
caltech	fnal	77.06	0.26
slac	cern	172.47	0.69

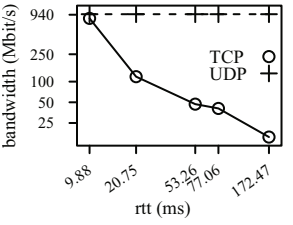


Fig. 3. Achievable bandwidth

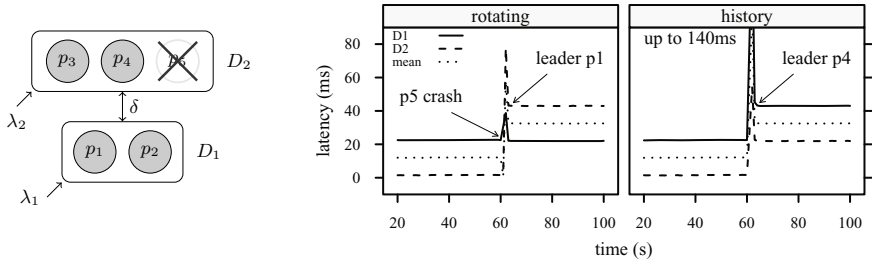
Implementation and Workload Generator. POLE and the oracles were implemented using ZooKeeper version 3.3.0. Our clients were implemented in Python using the asynchronous ZooKeeper API. They only perform *write* requests because, as explained previously, only these requests exercise the underlying consensus protocol. We use request sizes of 1 kB as in the work of Hunt [12].

We always start either a client or a server (*i.e.*, ZooKeeper server) per machine. We assume clients connect to the server with which they have the lowest round-trip time. Thus, we ignore the hop between client and server as in Fig. 1c.

Wide-area Setting Emulation. ZooKeeper ensembles are typically deployed in controlled environments. Consequently, instead of using testbeds such as PlanetLab,[4] we emulate a set of specific wide-area deployments using Netem, a network emulator available in the standard Linux kernel.[5] The deployments we define below use real round-trip times shown in Table 2, where the names refer to the following end-points: California Institute of Technology (`caltech.edu`), European Organization for Nuclear Research (`cern.ch`), Fermilab (`fnal.gov`), Stanford Linear Accelerator Center (`slac.stanford.edu`), and Technische Universität Dresden (tud, `tu-dresden.de`). All the round-trip values are aggregations of one month of data (November 2010) taken from the PingER project[6] except the values from tud to cern, which were performed by the authors (900 ping samples on the 2nd March 2011).

We use a Pareto distribution with mean given by the mean round-trip times in the table, and jitter of 2% of the mean. We use 2% jitter instead of the standard deviation to simplify the experiment setup. Figure 3 shows the achievable bandwidth using TCP and UDP for the given round-trip times measured in our cluster with Iperf.[7] Not shown in the figure is the bandwidth with round-trip time set to 0, which is about 940 Mbits/s as well. Note that, when the round-trip time is set to 0, the actual round-trip time is about 100 μs. In our experiments, we use round-trip times up to 77.06 ms, as higher round-trip times restrict the bandwidth excessively.

[4] `http://www.planet-lab.org`
[5] `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`
[6] `http://www-iepm.slac.stanford.edu/pinger`
[7] `http://iperf.sourceforge.net`



(a) 2 data centers topology (b) Request latency per data center

Fig. 4. Throughput and request latency with leader crash in two data centers

The communication between ZooKeeper servers uses TCP; consequently, we use only delay and jitter as emulation parameters because in TCP packet loss and reordering are perceived as jitter assuming the connections do not drop. The bandwidth of links can be limited by the emulator as well. Independent of emulator limits, note that TCP inherently limits the achievable bandwidth (see Fig.3). To simplify our setup, we rely only on the protocol-limited bandwidth. Although TCP can use more bandwidth with non-standard optimizations such as Jumbo frames, we focus on typical installations of both hardware and software.

Experiments. In the following, each experiment represents an emulated WAN deployment in two or three data centers. Two data centers (Fig.4a) enables clients close to a remote data center to perform *read* requests with lower latency. Three data centers (Fig.5) allows one data center to crash without affecting the availability of the service given that there is no data center with a quorum of servers. We assign λ and δ parameters to these topologies, where λ_i is the mean request rate (req/s) arriving at data center D_i (distributed uniformly at random across the servers in D_i); and δ is the link latency between any two servers in two different data centers. We use the mean round-trip times in Table2 to set δ, where $2 \cdot \delta = rtt$. Whenever appropriate, instead of giving δ parameters, we let the data centers be the end-points from Table 2, *e.g.*, slac, caltech, tud.

The experiments are executed for 120 seconds and always have as initial leader p_5. The settings were selected such that the leader p_5 is always initially the best leader. After 60 seconds, p_5 is killed. Measurements showing lines with no dots are single runs over time. Dots and bars represent mean aggregations over 5 runs. Latency aggregations refer to the steady state starting 10 seconds after the new leader is elected, *i.e.*, the last 50 seconds of a run. Recovery time aggregations refer to the interval from the time the new leader recognizes itself as such, up to when it processes the first request.

4.2 Request Latency and Random Request Distributions

We initially evaluate our oracles considering request latency when the request distribution arriving on each data center is arbitrary or unpredictable. In

particular, we evaluate oracles that are unaware of request distribution: history, rotating, worst-case, and consensus oracles. We divide our results in two data centers and three data center deployments.

Two Data Centers. We show that in deployments of two data centers, when the atomic broadcast has to cross the slow link (δ in Fig.4a), any oracle provides the same mean request latency after the crash of the initial leader p_5. Let D_1 and D_2 be respectively tud and cern, *i.e.*, $2 \cdot \delta = 20.75$ ms. Because we evaluate oracles using latency, we use a low request rate (λ_T) of 1000 req/s (8 Mbits/s). A low request rate does not overload the system, and consequently correct servers are synchronized when the leader crashes. The recovery time in such cases is roughly the same independent of which server is elected as leader. Requests arrive randomly on both data centers: $\lambda_1 = \lambda_2 = \frac{\lambda_T}{2} = 500$ req/s.

Figure4b shows the request latency on each data center before and after the crash of p_5 for two oracles: rotating and history. Because servers have the same θ_z (they are synchronized), history elects the server with highest *id*, *i.e.*, server p_4. The rotating oracle elects the server with the next *id* modulo the number of servers, *i.e.*, server p_1. Server p_4 is in the same data center as the crashed leader, while p_3 in another. Note that the form of the spike at 60 s depends rather on the run than on the oracle. With both oracles, the mean request latency over all requests is initially around 10 ms and after the crash around 30 ms. This is so because there is a complete quorum in the data center of the leader p_5 before the crash, but not after. The new leader – despite of its location – has to send every transaction over the slow link to form a quorum. Therefore, *any* oracle elects an equally fast leader, and all servers can be considered to be the fastest.

Three Data Centers. We show that in three data center deployments (1) history fails to elect the fastest leader, and (2) worst-case can elect faster leaders than consensus with respect to mean request latency. To that end, we assign to the data centers D_1, D_2, and D_3 in Fig.5 the end-points slac, caltech, and fnal in different combinations (see Table3). These end-points have the following round-trip times: 9.88 ms for slac-caltech, 53.26 ms for slac-fnal, and 77.06 ms for caltech-fnal. Let $\lambda_1 = \lambda_2 = \lambda_3 = \frac{\lambda_T}{3} = 333.33$ req/s.

Figure6 shows the results for the deployments described in Table3. Each group of bars corresponds to experiments with different leaders (indicated below the figure) which are reported together with the data center in which they are

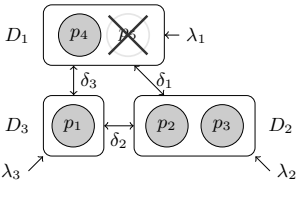


Fig. 5. Topology of 3 data centers

Table 3. End-to-end latency examples

Deployment	D_1	D_2	D_3
1	caltech	slac	fnal
2	slac	caltech	fnal
3	slac	fnal	caltech

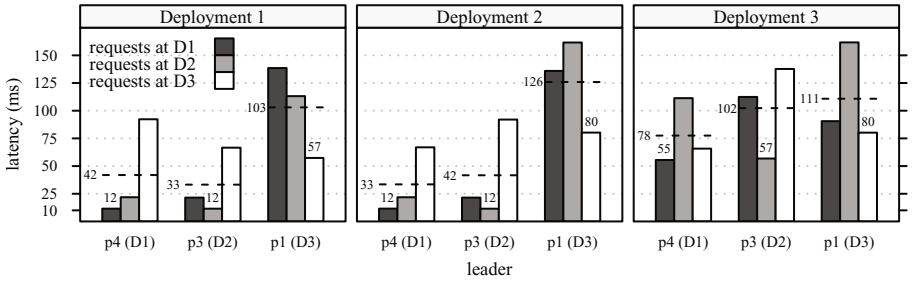


Fig. 6. Mean request latency per data center for different new leaders

located. Leaders provide different request latencies for requests arriving on the different data centers. For example, in Deployment 1, when server p_4 is elected, requests arriving on D_1, D_2, and D_3 are complete in roughly 12 ms, 22 ms, and 92 ms, respectively. The numbers over the bars represent the consensus latency of the leader (approximated by θ_c, see Sec. 2.2), while the dashed lines represent the mean request latency (approximated by θ_l), both rounded for readability. Leader p_4 in Deployment 1 has the consensus latency and mean request latency of about 12 ms and 42 ms, respectively.

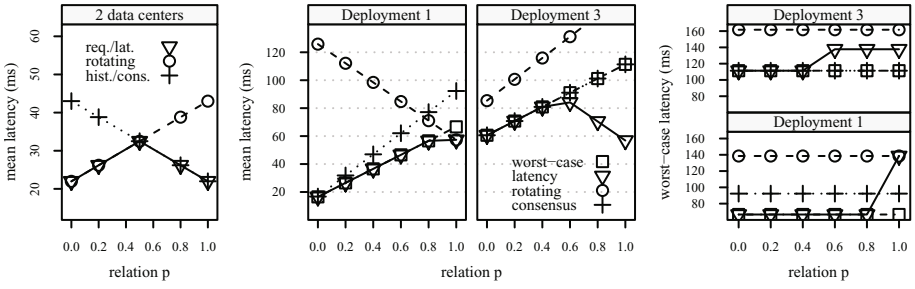
Due to how the server *id*s are assigned, history elects server p_4 in all deployments, while rotating elects p_1. By exchanging the location of p_1 and p_4, history and rotating behave in opposite ways. Therefore, no guarantees can be given on the performance of servers elected with them.

The worst-case oracle elects p_3 in Deployment 1 and p_4 in Deployment 2, minimizing both worst-case (highest bar of each leader) and mean request latency (dashed line). In contrast, consensus might elect either p_3 or p_4 depending on fluctuations of the consensus latency. In these deployments, worst-case might provide 20% lower mean request latency than consensus (33 versus 42 ms).

4.3 Request Latency and Uneven Request Distributions

We evaluate the request latency we obtain with our oracles when the request distribution arriving on each data center is uneven. In particular, we compare the performance of oracles that are aware of the request distribution (request and latency) with the remaining ones: history, rotating, worst-case, and consensus. We divide again our results in two data centers and three data center deployments.

Two Data Centers. We show that in two data centers request and latency oracles succeed to elect the *fastest* leader, while the remaining oracles do not. Figure 7a shows the same settings of the previous experiment in two data centers except that $\lambda_1 = (1 - p) \cdot \lambda_T$ and $\lambda_2 = p \cdot \lambda_T$. The y-axis represents the mean request latency after the crash with values p. Because after the crash all servers have the same θ_z and θ_c, history and consensus elect p_4. Similarly, worst-case (not depicted) calculates roughly the same θ_w for all servers (about 43 ms in Fig. 4b), and, therefore, elects the server with highest *id*, *i.e.*, p_4. The rotating oracle



(a) Two data centers (b) Mean, 3 data centers (c) Worst-case, 3 dc

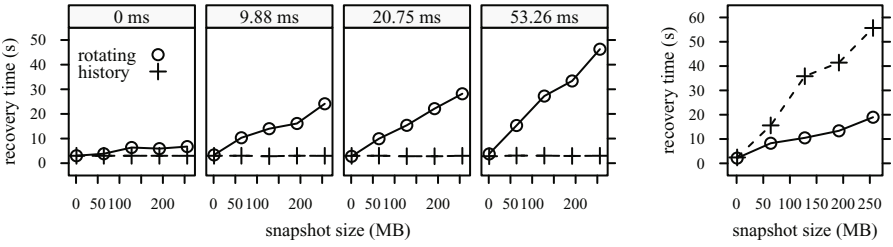
Fig. 7. Mean and worst-case request latency for different relations p

elects server p_1. In contrast, request and latency oracles elect the fastest leader for different relations p. In this example, the latency with them is half of the latency provided with the other oracles when requests arrive only on one data center. To understand the reason, suppose $p = 0$. If p_1 is the leader, all requests arriving on D_1 have to travel only once over slow link. If p_4 is the leader, all requests are first sent to the leader, then atomic broadcast is performed, and finally a confirmation is sent back (see Fig. 1c). As long as we are only concerned with mean request latency, and requests are unevenly distributed among the data centers, request and latency elect the *fastest* leader in two data center deployments.

Three Data Centers. In three data center deployments, the request oracle does not elect the server with the shortest mean request latency. We give a simple counter example. Consider the experiments in Fig. 6. Assume all requests arrive at D_3, *i.e.*, $\lambda_3 = \lambda_T$, and $\lambda_1 = \lambda_2 = 0$. In Deployment 1, the fastest leader is server p_1, while in Deployment 2, server p_4. The request oracle elects however in both cases server p_1 because its θ_r is the highest.

We next discuss some insights on Deployments 1 and 3 from above except that the request distributions are now uneven. In Deployment 1, $\lambda_1 = \frac{1-p}{2}\lambda_T$, $\lambda_2 = \frac{1-p}{2}\lambda_T$, and $\lambda_3 = p \cdot \lambda_T$. And, in Deployment 3, $\lambda_1 = \frac{1-p}{2}\lambda_T$, $\lambda_2 = p \cdot \lambda_T$, and $\lambda_3 = \frac{1-p}{2}\lambda_T$. Figures 7b and 7c respectively show the mean request latency and the worst-case request latency. We can observe that:

1. Oracles that are unaware of request distribution elect arbitrarily slow leaders. Consider Deployment 1 with $p = 0$ (Fig. 7b). The rotating oracle provides mean request latencies more than 7 times higher than latency (125.8 ms versus 16.5 ms). Consider Deployment 3 with $p = 1$. The consensus oracle has mean request latency of 111.3 ms, while the latency 57.4 ms.
2. The consensus oracle also fails to elect the server with the shortest worst-case request latency (Fig. 7c). In Deployment 1, the consensus oracle provides worst-case request latency about 20 ms higher than the worst-case oracle.



(a) 2 data centers varying δ_D (b) 3 data centers

Fig. 8. Recovery time varying snapshot size using history and rotating oracles

3. worst-case provides 10 ms worse mean latency than latency (66.6 ms versus 57.4 ms) in Deployment 1, and more than 50 ms worse in Deployment 3.
4. Finally, latency does not provide minimal worst-case request latency when p increases in both deployments, being from 30 to 60 ms worse than worst-case.

4.4 Recovery Time

Electing a leader that provides the minimal *recovery time* in a LAN deployment is the default election oracle in ZooKeeper. The history oracle elects the *fattest* leader to avoid transferring state from followers to the leader (see Sec. 2.3). We start experimenting whether the history oracle provides the same property in WAN deployments and then we consider using the latency oracle for that end.

History Oracle. We show that the *fattest* leader provides the minimal recovery time with two data centers, but not necessarily with three. Figure 8a shows the recovery time for 4 deployments with 2 data centers (Fig. 4a) and different round-trip times ($2 \cdot \delta$). We force p_1 to be outdated when the leader crashes, so that, becoming leader or follower, server p_1 has to first receive a snapshot of the current state from another server. To outdate a server, we kill the server and restart it when the leader is about to crash. The size of snapshots is application dependent; we used from about 0 up to 250 MB. The history oracle elects server p_4, and the rotating oracle elects server p_1. A round-trip time of 0 ms is equivalent to a LAN deployment, where all servers are in the same data center. When the outdated server p_1 is elected, the recovery time directly depends on the size of the snapshot to be transferred. By partitioning the servers in two data centers, the recovery time when electing p_1 is "amplified" with the increasing round-trip time, while is minimal when electing p_4.

The history oracle does not provides the minimal recovery time in 3 data centers. Figure 8b shows such deployment of the 3 data centers (Fig. 5) with the following parameters: $2 \cdot \delta_1 = 77.06$ ms, $2 \cdot \delta_2 = 20.75$ ms, and $2 \cdot \delta_3 = 9.88$ ms. We force both servers in data center D_2 to be outdated with respect to the leader. After leader p_5 crashes, servers p_1 and p_4 have the same θ_z and need to bring at least another server in sync to start processing requests. Because the latency between D_1 and D_2 is shorter compared to the one between D_3 and D_2, server p_1 presents a shorter recovery compared to p_4.

Latency Oracle. It is easy to see that oracles such as latency cannot minimize the recovery time because it is guided by the request distribution. Consider the previous experiment with 3 data centers. Assume $\lambda_1 = \lambda_T$, and $\lambda_2 = \lambda_3 = 0$. The *fastest* is server p_4, while server p_1 provides minimal recovery time.

5 Extensions

We now briefly discuss some aspects not addressed in the paper: combination of scoring functions, how to perform the first election, and how to cope with workload changes and score miscalculations.

Co-optimizing Multiple Metrics. In our experiments, we considered oracles optimizing a single metric. Co-optimizing several oracles is not always trivial. For example, we have shown that worst-case cannot optimize the mean request latency, while latency cannot optimize the recovery time. However, a simple way to achieve co-optimization is the following. The application defines a priority order $\theta_1, \ldots, \theta_n$ among the scores of interest. Each server corresponds to a vector of scores in which we can easily identify an ordering function. To enable the comparison of vectors, it is possible to define equivalence ranges for some scores, such that all servers whose score is in the same class are considered equally good.

Consider the example where an application wants to elect a leader that optimizes the mean latency and, as a secondary goal, that minimizes recovery time. The application can combine $\theta_z(p)$ and a modified $\theta'_l(p)$, which truncates $\theta_l(p)$ in latency classes: 10 ms, 20 ms, etc. An oracle implementing such a combination of scoring functions can elect the server in the lowest latency class $\theta'_l(p)$ and with the highest $\theta_z(p)$, *i.e.*, the fattest among the fastest leaders.

Initial Election and Resignation. All experiments we performed already started with a leader, which later on crashed. For the first election, the get-score function simply blocks until enough information is collected to generate a score.

An application may also want to include *resignations*, which happen when the actual leader is not the actual best leader. There is a number of reasons for this to happen. For example, when a crashed server recovers and rejoins the ensemble, it might be a better leader than the actual leader is. Another reason can be that some score, as for example the request distribution, may change over time making the actual leader a worse leader than some follower.

POLE can also easily be extended to consider resignations as follows: The leader periodically sends messages to other servers querying their actual θ scores. Based on some threshold, the leader decides if some servers has a better score than itself, and resigns its leadership. In contrast to "after-crash" scores, as we have presented, "before-resignation" scores have to include the actual leader in RTT_i and F_i vectors because it is supposed to be alive during the next election.

6 Related Work

Consensus is the primitive underlying the atomic broadcast protocol [5]. To solve consensus, it is necessary and sufficient to elect a leader [5,7]. The problem of

leader election has been broadly investigated from many viewpoints, from minimizing the synchrony and reliability requirements imposed on links [2,15], to optimizing the Quality of Service of failure detection [8], to electing the leader that can minimize the latency of solving consensus [17]. The ZooKeeper atomic broadcast protocol (Zab) implements a variant of the atomic broadcast primitive called *primary order atomic broadcast* [13]. Similar to many consensus implementations, Zab uses a leader to suggest a total order of transactions.

Most leader election algorithms are, however, oblivious to the application needs and elect a leader based on its *id*, *e.g.*, the highest *id* in the alive-set of each process [9], or the next *id* when incrementing a counter [1,5]. In fact, leader election has been proposed as a *service* for multiple applications at the same time [9,18]. In this work, we have shown that the *service* approach is not sufficient when applications use different criteria to select a leader.

Selecting leaders based on performance has been previously proposed by Singh and Kurose [19,20]. They do not present the distributed algorithm to spread votes, but focus on different voting schemes taken from social sciences to elect the best leader when some votes might be wrong (*e.g.*, indicate a bad leader). Furthermore, they assume that processes that vote do not crash, and that messages cannot be lost.

Santos *et al.* [17] presented a rotating leader election that finds an optimal leader after a series of intermediate elections. In systems like ZooKeeper, this solution can be very costly because each reelection incurs additional recovery costs, even when servers have the same history, *e.g.*, snapshot reading, connection establishment, and the first phase of the consensus protocol. Furthermore, their approach focuses exclusively on *consensus latency*. As we have shown, it is often more important to minimize other metrics such as the *request latency*.

Sampaio and Brasileiro [16] propose a configurable solution for *id*-based elections by using a process-ordering oracle. This additional component changes the *a priori* order of processes at run-time based on application metrics (but the authors only evaluate *consensus latency*). Nevertheless, their solution is not of great practical interest because it requires the processes to use each consensus instance to agree on the *a priori* process ordering of the next consensus instance. Paxos [6,14] and ZooKeeper's atomic broadcast [13], however, promote the simultaneous execution of instances of consensus to achieve high performance in replicated services; Instead of electing leaders based on an previously agreed-upon order, POLE enables processes to locally assign scores to themselves and share their scores. Computing scores accurately is therefore critical for selecting an appropriate leader, and score deviations introduced, for example, due to sampling may lead to different processes being selected. Such deviations, however, do not preclude liveness.

The idea of broadcasting *ids* and electing the process with the highest one is the core of the Bully algorithm, which was proposed by Garcia-Molina [11] in the context of synchronous systems. Bully and its partially-synchronous counterpart, the Invitation algorithm [11], do not enable configurability and target generic master-worker applications. In contrast, POLE has been specifically designed

for replicated systems, and it provides mechanisms such as electing a process only once there is a quorum of processes running. Using scores as opposed to identifiers is also a key difference.

The separation of concerns between failure detection and consensus has been proposed previously by Chandra and Toueg [5]. While being conceptually fundamental, this strict separation of concerns does not necessarily result in more efficient implementations. Using application-level failure detection information, we can avoid sending failure detection messages when a stable leader exists [10]. Note that although some scoring function might require additional messages to calculate their θ scores, that is not the concern of the election algorithm itself.

Bakr and Keidar study the running time of a "communication round" of distributed algorithms using TCP over the Internet [3]. The chain replica-leader-quorum in this work relates to their *secondary leader* communication pattern.

7 Conclusion

In replicated services, leaders have a great impact on metrics of interest to client applications, such as recovery time and mean request latency. In this work, we have presented POLE, a general and flexible leader election algorithm for practical replicated systems. POLE uses an application-defined scoring function to assign scores to servers when selecting a leader. A variant of POLE that uses *zxid*s as scores has been deployed as part of ZooKeeper, and in this work we have proposed additional scoring functions that satisfy different application requirements. Our goal, however, was not to explore exhaustively the space of oracles, but instead to argue for the importance of providing such flexibility to applications deployed in heterogeneous settings, such as the ones encompassing multiple data centers.

Our experimental results illustrate trade-offs that designers face when designing oracles for POLE. When client requests are unevenly distributed, the consensus oracle described in the literature performs poorly compared to mean *and* worst-case request latencies. There are a few available options when deploying POLE. For minimizing the worst-case request latency, the worst-case oracle can be used. A simple request oracle can minimize the mean request latency for settings with two data centers. With three data centers, a more complex oracle such as the latency oracle is however necessary. Leaders elected with these oracles can be at least 50% faster compared to arbitrarily elected ones. In contrast, although history is able to minimize recovery time in two data centers, none of the presented oracles can minimize it in three data centers. The design of a generic recovery-time oracle is still an open problem; in general exploring further the space of oracles is subject of future work.

Acknowledgments. This work has been partially supported by the SRT-15 project (ICT-257843), funded by the European Commission. We also would like to thank Benjamin Reed and Christof Fetzer for useful discussions on this work.

References

1. Aguilera, M., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Stable Leader Election. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 108–122. Springer, Heidelberg (2001)
2. Aguilera, M., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: On implementing Omega in systems with weak reliability and synchrony assumptions. Distributed Computing 21, 285–314 (2008)
3. Bakr, O., Keidar, I.: Evaluating the running time of a communication round over the Internet. In: PODC 2002: Proceedings of the 21st Annual Symposium on Principles of Distributed Computing, pp. 243–252. ACM, New York (2002)
4. Burrows, M.: The Chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI 2006, pp. 335–350. USENIX Association, Berkeley (2006)
5. Chandra, T., Toueg, S.: Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM 43(2), 225–267 (1996)
6. Chandra, T.D., Griesemer, R., Redstone, J.: Paxos made live: an engineering perspective. In: Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, pp. 398–407. ACM, New York (2007)
7. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM 43(4), 685–722 (1996)
8. Chen, W., Toueg, S., Aguilera, M.: On the Quality of Service of Failure Detectors. IEEE Transactions on Computers 51, 561–580 (2002)
9. Fetzer, C., Cristian, F.: A Highly Available Local Leader Election Service. IEEE Trans. Softw. Eng. 25, 603–618 (1999)
10. Fetzer, C., Raynal, M., Tronel, F.: An adaptive failure detection protocol. In: Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing, PRDC 2001, pp. 146–153. IEEE Computer Society, Washington, DC (2001)
11. Garcia-Molina, H.: Elections in a Distributed Computing System. IEEE Trans. Comput. 31(1), 48–59 (1982)
12. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: ZooKeeper: Wait-free coordination for Internet-scale systems. In: Proceedings of the 2010 USENIX Annual Technical Conference, USENIXATC 2010. USENIX Association, Berkeley (2010)
13. Junqueira, F., Reed, B., Serafini, M.: Zab: High-performance broadcast for primary-backup systems. In: Proc. of the IEEE Int'l Conf. on Dependable Systems and Networks, DSN-DCCS (2011)
14. Lamport, L.: Paxos made simple. ACM SIGACT News 32(4), 18–25 (2001)
15. Malkhi, D., Oprea, F., Zhou, L.: Ω Meets Paxos: Leader Election and Stability without Eventual Timely Links. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 199–213. Springer, Heidelberg (2005)
16. Sampaio, L., Brasileiro, F.: Adaptive Indulgent Consensus. In: International Conference on Dependable Systems and Networks, pp. 422–431 (2005)
17. Santos, N., Hutle, M., Schiper, A.: Latency-aware leader election. In: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC 2009, pp. 1056–1061. ACM, New York (2009)
18. Schiper, N., Toueg, S.: A Robust and Lightweight Stable Leader Election Service for Dynamic Systems. Tech. Rep. 2008/01, University of Lugano (March 2008)
19. Singh, S., Kurose, J.: Electing leaders based upon performance: the delay model. In: 11th Int'l Conference on Distributed Computing Systems, pp. 464–471 (1991)
20. Singh, S., Kurose, J.: Electing "Good" Leaders. Journal of Parallel and Distributed Computing 21(2), 184–201 (1994)

PolyCert: Polymorphic Self-optimizing Replication for In-Memory Transactional Grids*

Maria Couceiro, Paolo Romano, and Luis Rodrigues

INESC-ID
Instituto Superior Tecnico, Technical University of Lisbon
{maria.couceiro,ler}@ist.utl.pt, romano@inesc-id.pt

Abstract. In-memory NoSQL transactional data grids are emerging as an attractive alternative to conventional relational distributed databases. In these platforms, replication plays a role of paramount importance, as it represents the key mechanism to ensure data durability. In this work we focus on Atomic Broadcast (AB) based certification replication schemes, which have recently emerged as a much more scalable alternative to classical replication protocols based on active replication or atomic commit protocols. We first show that, among the existing AB-based certification protocols, no "one-fits-all" solution exists that achieves optimal performance in presence of heterogeneous workloads. Next, we present PolyCert, a polymorphic certification protocol that allows for the concurrent coexistence of different certification protocols, relying on machine-learning techniques to determine the optimal certification scheme on a per transaction basis. We design and evaluate two alternative oracles, based on parameter-free machine learning techniques that rely both on off-line and on-line training approaches. Our experimental results demonstrate the effectiveness of the proposed approach, highlighting that PolyCert is capable of achieving a performance extremely close to that of an optimal non-adaptive certification protocol in presence of non heterogeneous workloads, and significantly outperform any non-adaptive protocol when used with realistic, complex applications that generate heterogeneous workloads.

Keywords: Transactional systems, replication, autonomic computing, machine learning, atomic broadcast.

1 Introduction

In-memory NoSQL transactional data grids are emerging as an attractive alternative to conventional relational distributed databases. By employing alternative data models, such as plain key/value stores, and relying on replication rather than on persistence to stable storage to ensure data durability, in-memory transactional data grids have shown to achieve higher performance, scalability, and elasticity when compared to classical SQL-based database management systems [32, 28].

* This work was partially supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds and the Aristos project (PTDC/EIA-EIA/102496/2008), and by the European Commission through the Cloud-TM project (FP7-257784).

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 309–328, 2011.

Replication clearly plays a role of paramount importance in these in-memory data platforms, as it represents the key mechanism to ensure data durability in face of unavoidable node failures. Unsurprisingly, replication algorithms employed in these in-memory platforms take inspiration from the vast literature on replication of transactional systems (traditionally, database systems [24,18,23], but also, more recently, transactional memory systems [8,7]).

Among the plethora of transactional replication mechanisms published in literature, over the last years, a wide body of research has highlighted that schemes based on Atomic Broadcast (AB) and certification [24,18,23] tend to outperform classic eager replication schemes based on distributed locking and atomic commit, which suffer from large communication overheads and are prone to thrashing due to distributed deadlocks [12]. Conversely, certification based schemes avoid any onerous replica coordination during the execution phase, running transactions locally in an optimistic fashion. The consistency of replicas (typically, 1-copy serializability [3]) is ensured at commit- time, via a distributed certification phase that uses a single AB to enforce agreement on a common transaction serialization order, avoiding distributed deadlocks, and providing non-blocking guarantees in the presence of failures. Furthermore, unlike active-replication approaches that require the full execution of update transactions at all replicas [30], only one replica executes an update transaction, whereas the remaining replicas are only required to validate the transaction and to apply the resulting updates. This allows to achieve high scalability levels even in the presence of write-dominated workloads, as long as the transaction conflict rate remains moderate [24].

In the design space of 1-copy serializable certification replication protocols, which represents the focus of this work, a decision that can have a dramatic impact on the actual efficiency and robustness of the system is related to how to address the trade-off between the size of the messages sent via the AB primitive and the number of communication steps required during the transaction commit phase. Depending on how this trade-off is addressed, existing certification-based replication algorithms can be classified into three main categories:

- Solutions that disseminate the whole transaction's read-set to all replicas, called *Non-voting* schemes, allow each replica to certify transactions locally, by sending both the read-set and write-set via an AB primitive. This makes these protocols optimal in terms of communication steps, but also makes them prone to generate very large messages and to overload the Group Communication System.
- *Voting* schemes, which avoid broadcasting the read-set of transactions by sending (via AB) only the write-set, thus drastically reducing the network bandwidth consumption. On the other hand, they incur into the costs of an additional coordination phase along the critical path of the transaction commit , which can hamper significantly performance.
- Approaches relying on the space efficient encoding of Bloom Filters [4] to implement a variant of the non-voting certification mechanism, called *Bloom Filter Certification* (BFC) [8] . Unlike voting mechanisms, BFC determines the outcome of transactions using a single AB, generating smaller messages when compared to

non-voting protocols. The probabilistic nature of the Bloom filter encoding, however, induces false positives in the certification phase, increasing the transaction abort rate.

The above protocols are designed to ensure optimal performance in different workload scenarios and, as we will show in the following, they can exhibit up to 10x differences in terms of maximum throughput. Our goal is to alleviate the developers/administrators from the hard and time-consuming task of profiling the application and selecting the most suitable replication protocol for each deployment. Furthermore, a static configuration may lead to largely suboptimal configurations in presence of heterogeneous workloads. In these contexts, the employment of a single, statically chosen, replication mechanism, optimized for a specific workload type, will lead to suboptimal performance when processing the transactions that have different characteristics.

The solution presented in this work, which we named *Polymorphic Self-Optimizing Certification* (PolyCert), supports the simultaneous use of the three aforementioned classes of protocols, and relies on machine-learning techniques to determine, on a per transaction basis, the certification strategy to be adopted. PolyCert relies on a modular design, which encapsulates the logic associated with the on-line choice of the replication strategy into a generic oracle. We design and evaluate two alternative mechanisms to implement this oracle, based on two different parameter-free statistical learning techniques.

- An off-line technique based on regression decision trees [26], that requires a preliminary, computational intensive, feature selection and training phase, but that was shown (in our previous work [9]) to achieve high accuracy in forecasting the performance of Atomic Broadcast algorithms in presence of heterogeneous workloads.
- An on-line reinforcement learning technique, that uses an innovative, parameter-free variant of a very lightweight, but theoretically optimal solution [2] to face the exploration versus exploitation dilemma, i.e. the search for a balance between exploring the environment to find profitable actions while taking the empirically best action as often as possible.

Via an extensive experimental evaluation, based on a fully fledged system prototype and a range of heterogeneous benchmarks, we assess the effectiveness of the proposed approaches in terms of performance benefits and learning time. We show that PolyCert can achieve a significant speed-up with respect to static solutions and enhance the robustness of the system to unexpected fluctuations of the workload.

The remainder of the paper is structured as follows. Section 2 reports the results of a performance evaluation study highlighting the relevance of the addressed problem. The system architecture is presented in Section 3. Section 4 describes the functioning of PolyCert and the mechanisms employed to determine at run-time which certification strategy to use. The results of the experimental evaluation study are reported in Section 5. Related work is analysed in Section 6. Section 7 concludes the paper.

2 Motivations

As already mentioned in the Introduction section, existing certification-based solutions can be classified into three main categories:

- **Non-Voting Certification (NVC):** These solutions [24,7,23] disseminate the whole read-set and write-set using the AB service, allowing every replica to determine, upon delivery of the corresponding message, the outcome (commit/abort) of the transaction, by running the certification procedure locally. These schemes are optimal in terms of communication steps, delivering excellent performance when used in workloads characterized by small transaction read-sets. On the other hand, they exhibit very poor performance in presence of transactions reading a significant number of data items. Even worse, in these scenarios, the large network traffic generated by this protocol can saturate and disrupt the proper functioning of the Group Communication Service, leading to network partitions and false failure suspicions.
- **Voting Certification (VC):** These solutions [17] disseminate exclusively via AB the transaction write-set, thus avoiding the issues incurred in by Non-voting schemes with large transaction read-sets. On the down side, the transaction can only be certified at the site in which it was originated. This implies the need for an additional communication phase, executed using a Uniform Reliable Broadcast (URB) [14] (a lighter communication primitive when compared to AB), which is triggered by the replica where the transaction was originated in order to inform the remaining replicas of the final outcome of the transaction. This extra communication phase, which requires at least two communication steps, has a negative impact on the latency of the commit phase, which represents by far the dominating cost for small transactions. By introducing additional latency in the critical path of the commit phase, which needs to be run sequentially for conflicting transactions, these schemes can adversely affect the maximum throughput achievable by the platform [29].
- **Bloom Filter Certication (BFC):** An alternative approach, denoted as Bloom Filter Certification (BFC) [8], consists in encoding the read-set of the transaction in a Bloom filter [4], a space-efficient data structure that allows compressing the messages disseminated via the AB service, while still allowing every replica in the system to deterministically certify the transactions. Unlike Voting schemes, BFC avoids additional communication steps during the commit phase. In terms of generated network traffic, even though BFC generates larger messages than voting protocols, it typically reduces significantly the size of the messages exchanged via the AB service when compared to non-voting schemes. On the down side, BFC can suffer from false positives due to the probabilistic nature of Bloom filter-based encoding, which ultimately leads to an additional rate of aborted transactions.

From the above discussion, the performance of each of these three alternative certification mechanisms is strongly dependent on the actual distribution of the size of the read-sets generated by the transactional application. Unfortunately, realistic transactional applications can exhibit very heterogeneous workloads encompassing read-sets whose sizes range from less than ten to hundreds of thousands of objects. We have experimentally observed this phenomena, as illustrated in Figure 1, which depicts the distribution of the read-set size for a widely used benchmarking application for in-memory transactional systems (in particular Transactional Memories), namely the STMBench7 [13] benchmark.

In Figure 2 we show the results of a sensitivity analysis aimed at assessing the actual impact of the read-set size distribution on the performance of the three certification

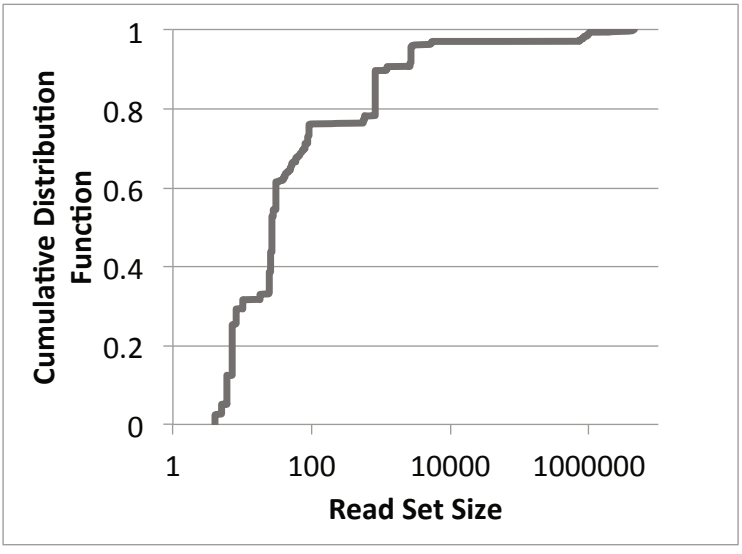


Fig. 1. Distribution of transaction read-set size in the STMBench7 Benchmark

schemes described above. The results were obtained using a simple synthetic benchmark adapted from the Bank Benchmark originally used in [8]. This benchmark simulates the concurrent transfer of funds from different bank accounts (modelled as a simple array of doubles), and was altered to vary the number of items read within a transaction in the range [1,100'000]. Further, to focus only on the effects due to variations of the read-set size, which represents the goal of this sensitivity analysis, we configured the benchmark to never generate conflicts among transactions. The only aborts experienced in the system are therefore those determined by false positives with the BFC scheme (which was configured to have an additional abort-rate of 1%, as in [8]). These results were obtained running on a cluster of eight nodes, each one equipped with two Intel Quad-Core XEON at 2.0 GHz, 8 GB of RAM, running Linux 2.6.32-26-server and interconnected via a private Gigabit Ethernet (which represents the reference experimental platform used in the remainder of the paper). The in-memory transactional data grid and the certification protocols were implemented in JAVA. The system uses two main components: i) a state of the art Software Transactional Memory (STM), namely JVSTM [6], used to manage local concurrency, and ii) a replicated key/value store, used to maintain associations between unique object identifiers and object instances. Further details on the system architecture will be provided in Section 3.

Our experimental results highlight that no-one-fits-all solution exists that maximizes the throughput across all the considered workloads. On the contrary, NVC provides the best performance in the scenario with small read-sets, BFC in the scenario with 1000 items in the read-set, and VC is by far the best performing protocol with large read-sets. Further, the relative difference in the performance between the best and worst performing protocol for each scenario ranges from a factor 2.5x (BFC vs VC, read-set size equal to 1000) to 10x (VC vs NVC, read-set size equal to 100'000).

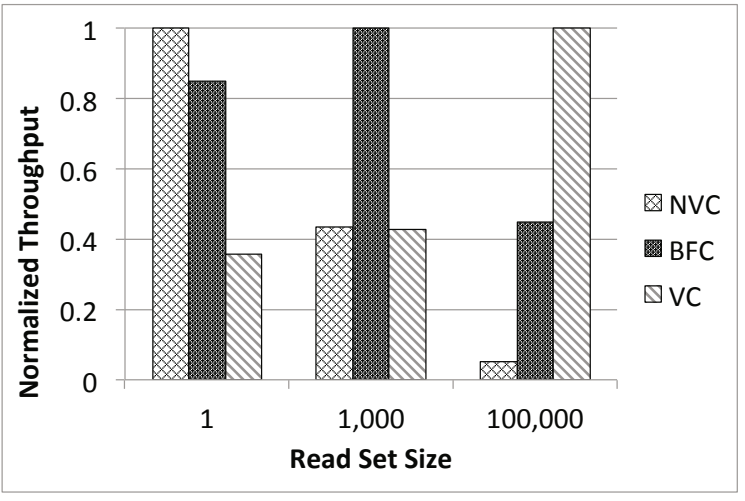


Fig. 2. Throughput of three certification strategies with different read-set sizes

3 System Architecture

The system architecture is depicted in the diagram shown in Figure 3. At the topmost layer, it exposes the API of an object-oriented STM, which is however fully replicated across a number of distributed nodes. The API provided to applications is a transparent extension of JVSTM's API, a state-of-the-art STM relying on an efficient Multi Version Concurrency Control (MVCC) algorithm [3]; a strong advantage of JVSTM is that read-only transactions are never required to block. The JVSTM programming paradigm requires that the programmer encapsulates any shared mutable state within VBoxes, which are then managed by JVSTM's MVCC to ensure transactional atomicity and isolation. This allows separating the transactional and non-transactional state of the application, ensuring strong atomicity [19] at no additional costs. We modularly extended JVSTM by augmenting it with what we have named a Polymorphic Replication Manager (PRM).

The PRM is in charge of triggering the execution of a certification protocol for each of the locally executed transactions, and to participate in the certification of transactions that have been executed at remote nodes. A unique feature of PRM, with regard to existing replication managers, is that it is able to determine, for each locally executing transaction, which certification algorithm is more appropriate, given the characterization of the transaction. The logic for determining the certification scheme is encapsulated by the abstraction of a Replication Protocol Selector Oracle (RPSO), whose interface exports two main functionalities:

– Given a local transaction, it selects the most appropriate certification protocol to be executed by the PRM. In Section 4.1 we will present and evaluate two performance forecasting methods which are based on alternative machine learning techniques.

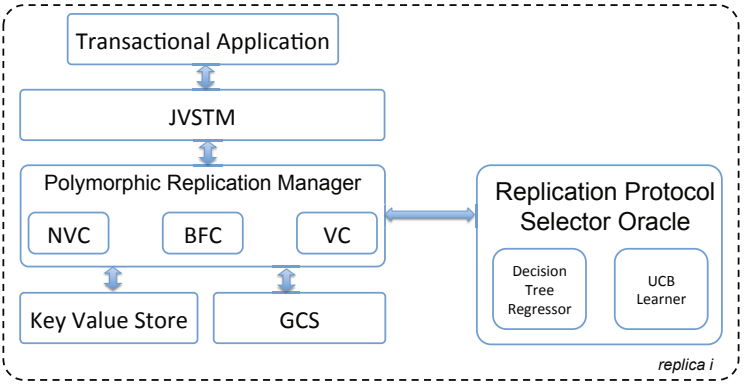


Fig. 3. Architectural Overview (Single Node Perspective)

– Collecting historical data about the execution of past transaction, to improve the selection process. For this, it exports an interface that allows the PRM to register the following information: i) the commit time experienced by a transaction, and ii) the certification protocol that it used. This allows the RPSO to gather, store and analyse (either on-line or off-line) statistical data on the distribution of the commit time of transactions.

The PRM also interacts with two other components, the Group Communication System (GCS) and a local Key Value Store (KVS).

The GCS, Appia [21] in our implementation, provides a number of communication abstractions required by the PRM: view synchronous membership, AB and URB [14].

Finally, the Key Value Store is a weak hash map, used to maintain the mapping between application level transactified objects (namely, containing JVSTM VBoxes) and replica-wide unique object identifiers, which are generated automatically by our framework upon creation of a new transactional object. More in detail, an entry of the key/value store contains the unique object identifier, as its key, and a *weak reference* to the local transactional object as its value. This information is used by the PRM, upon reception of a commit request for a remote transaction T, to retrieve in the local JVSTM instance the objects that were read/updated by T during its remote execution. The usage of a weak hash map ensures that the Java garbage collector is not prevented from discarding the object referenced by a hash map entry whenever all application level references to the object have been removed, thus avoiding any interference with the local JVM garbage collection mechanism.

4 The PolyCert Replication Protocol

PolyCert is designed to allow the simultaneous use of all the three AB-based certification protocols introduced before, namely NVC, BFC and VC. Specifically, at commit time (when the size of the transaction read-set and write-set is already known), the PRM

invokes the RPSO to determine which certification protocol to use in order to finalize the transaction's execution. This clearly implies that concurrent transactions may use different protocols, which need to coexist without endangering the correctness of the system. The three protocols lend themselves quite naturally to coexist, given that all of them rely on a first common phase during which they establish, via the AB primitive, the global serialization order for a committing transaction (even though each protocol piggybacks different information). Naturally, certification messages need to be tagged with a label that specifies which of the protocols is being used for each transaction.

Upon delivery of an AB message, the PRM performs the following steps:

- The message is inserted in a queue containing the transactions to be certified.
- If the transaction is being certified using NVC or BFC, no further processing is done until the message reaches the head of the queue.
- If the transaction is being certified using VC, and the node was the originator of the transaction, the following procedure is executed immediately. Upon the enqueuing of a transaction, say T, if and only if T does not conflict with any of the transactions already present in the certification queue (i.e. the read-set of T does not intersect with the write-sets of the transactions already present in queue), T is immediately validated, by verifying whether the snapshot that it read is still fresh, the URB convoying the decision outcome is triggered. This optimization, originally proposed in [29], allows to overlap the URB dissemination phases of (non-conflicting) transactions with the time spent by transactions in the certification queue, reducing the risk of convoy effects which are otherwise known (and confirmed by our experience) to significantly hamper performance of VC schemes.

Subsequently, messages are removed from the head of the queue one by one and the corresponding transactions validated in a sequential manner. More specifically:

- If the message that is extracted from the head of the queue corresponds to a transaction that is being certified using NVC or BFC, each node applies locally the corresponding certification algorithm. Essentially, it verifies if the read-set accessed by the transaction is still valid, and commits or aborts the transaction accordingly.
- If the message that is extracted from the head of the queue corresponds to a transaction that is being certified using VC, the node checks if a vote has already been received or not. If a vote has been received and the decision was to commit the transaction, the write set is applied. Otherwise, if the vote has been received and the decision was to abort the transaction, the write-set is discarded (actually, the transaction can be immediately removed from the queue as soon as an abort vote is received). On the other hand, if a vote has not been received and the transaction is remote, the certification is suspended until the corresponding vote is received. Finally, if a vote has not been received and the transaction is local, the node validates the transaction as described above, and issues the vote (this corresponds to the scenarios where the optimization described before cannot be applied).

As a final remark, note that, since all replicas deliver the certification messages in the same order due to the use of the AB primitive, and decide deterministically which certification protocol to use, consistency is guaranteed even in presence of concurrent transactions being processed using different certification schemes.

4.1 Replication Protocol Selection Oracle

As already mentioned, the Replication Protocol Selector Oracle abstraction (RPSO) is a convenient form of encapsulating different performance forecasting techniques. In this paper, we present two implementations of the RPSO, using two different machine learning techniques, namely a regressor based on decision trees [26], and a reinforcement learning technique, namely UCB [2], as described below.

Oracle based on regressor decision trees. In order to forecast the time necessary for committing a transaction with each of the three considered certification strategies, we start by forecasting the size, m_p, of the AB message that would be generated by each of the certification protocols $p \in \{NVC, BFC, VC\}$. This corresponds to the number of bytes required to transmit the transaction read-set and write-set with NVC, the transaction write-set with VC, and the write-set and the Bloom filter based encoding of the transaction read-set with BFC.

Next, we forecast the time for marshalling and validating a transaction with each of the considered certification schemes. To this end, we maintain, for each certification strategy, a moving average of the average marshalling time per byte, denoted as T_p^m, and of the validation time, denoted as T_p^v, for all $p \in \{NVC, BFC, VC\}$. Further, for BFC, we maintain moving averages of the time required to build a Bloom filter that encodes the read-set (normalized by the read-set's size), denoted as T^{BF}. Finally, for VC, we maintain also the moving averages of the self-delivery latency of the URB convoying the vote from the transaction's initiator, denoted as T^{URB}. Note that the choice of measuring self-delivery latencies allows us to avoid distributed clock-synchronization mechanisms, which in our preliminary experiments revealed not to be sufficiently accurate for our purposes.

Using the metrics above, the commit latency T_p for a transaction using certification protocol p is then forecast as follows:

$$T_{NVC} = T^m_{NVC} \cdot m_{NVC} + T^v_{NVC} + T^{AB}(m_{NVC}) \quad (1)$$

$$T_{BFC} = T^m_{NVC} \cdot m_{BFC} + T^v_{VC} + T^{BFC} \cdot rsSize + T^{AB}(m_{BFC}) \quad (2)$$

$$T_{VC} = T^m_{VC} \cdot m_{VC} + T^v_{VC} + T^{AB}(m_{VC}) + T^{URB} \quad (3)$$

where $T^{AB}(m)$ is the forecast latency for self-delivering a message of size m using the AB primitive. To this end, we exploit our recent results in [9], where we presented and evaluated a series of (off-line) machine learning techniques to forecast AB's performance, including neural-networks [16] and support vector machines [31]. In the light of the results achieved in [9], we integrated in our system a regression technique relying on the Cubist© [25] decision-tree regression algorithm, which proved to be the most accurate and robust predictor among those evaluated.

Analogously to classic decision tree based classifiers, such as C4.5 and ID3 [26], Cubist© builds decision trees choosing the branching attribute such that the resulting split maximizes the normalized information gain (namely the difference in entropy). However, unlike C4.5 and ID3, which contain an element in a finite discrete domain (i.e. the predicted class) as leafs of the decision tree, Cubist© places a multivariate linear model at each leaf, which we use to predict the AB self-delivery latency (expressed in microseconds).

Table 1. List of metrics (features) collected by the Monitoring Layer

Metric	Description
freeMem	Free memory in the Java Virtual Machine
tLGC	The time since the last garbage collection
pLGC	% of time since the last GC cycle w.r.t. the time between the last 2 GC cycles
undelivMsgs	#TO Broadcast msgs and not yet self-delivered
bytesUp_x	#Bytes received over a x msec. time window
bytesDown_x	#Bytes sent over a x msec. time window
TOBUp_x	#TOB deliver events over a x msec. time window
TOBDown_x	#TOB broadcast events over a x msec. time window
totCPU_x	% total CPU utilization over a x msec. time window
esCPU_x	% CPU utilization by ES thread over a x msec. time window
TCPqueue	Outgoing messages queued at the Transport Layer

In order to generate the training data for the decision tree regressor we ran the synthetic benchmark described in Section 2, for each of the three considered certification protocols, varying every 3 minutes the read-set size of the generated transactions in the set {10,100, 1'000, 100'000}. Overall the training data set gathered by each replica is constituted, on average, by approximatively 25'000 samples, reporting the self-delivery latency for each AB message along with the message size, and a total of 53 different metrics (i.e. context information), also referred to as features, including averages on multiple time scales and time series of a plethora of metrics (listed in Table 1) concerning the utilization of various system resources (CPU, RAM and Network). The choice of this synthetic benchmark to generate the training data set has the following rationale: since this benchmark generates transactions with very heterogeneous read-set sizes, it allows gathering a good a-priori knowledge on the performance of a wide range of possible workload scenarios that the system may face when running more complex, realistic applications.

To minimize the effects of overfitting, which are likely to occur given the high dimensionality of the feature space, we run a greedy feature selection algorithm (Forward Selection [15]) aimed at discarding loosely correlated features and boosting the predictor's accuracy. Feature selection is by far the most time consuming phase of the off-line training, taking on average 45 minutes (per replica) when run on a PC equipped with Intel Core 2 CPU with a 2.2GHz clock-rate and 2GB of RAM. On the other hand, feature selection allows to achieve a significant improvement in the accuracy of the predictions, as highlighted by the results shown in Table 2, which report the correlation factor and mean absolute error using 10-fold cross-validation before and after performing feature selection.

Oracle based on the UCB online learner. The second implementation of the Oracle Layer employs an on-line learning technique. Therefore, it does not require an a-priori

Table 2. Prediction accuracy of the decision tree regressor before and after feature selection

Metric	Before FS	After FS
Relative Absolute Error	0.81	0.30
Correlation Coefficient	0.17	0.76

computational intensive off-line training. Instead, it relies on a lightweight reinforcement learning (RL) technique that updates the knowledge of the oracle as the system is running.

This oracle relies on a customized, self-tuning version of a state of the art RL algorithm, called UCB (Upper Confidence Bounds), which solves (in a theoretically optimal manner) a classical on-line learning problem, known in literature as the *multi-armed bandit* [27]. In this problem, a gambling agent is faced with a bandit (a slot machine) with k arms, each associated with an unknown reward distribution. The gambler iteratively plays one arm per round and observes the associated reward, adapting its strategy in order to maximize the average reward. Formally, each arm i of the bandit, for $0 \leq i \leq k$, is associated with a sequence of random variables $X_{i,n}$ representing the reward of the arm i, where n is the number of times the lever has been used. The goal of the agent is to learn which arm i maximizes the criterion:

$$\mu_i = \sum_{n=1}^{\infty} \frac{1}{n} X_{i,n}$$

that is, achieves maximum average reward. To this purpose, the learning algorithm needs to try different arms in order to estimate their average reward. On the other hand, each suboptimal choice of an arm i costs, on average, $\mu^* - \mu_i$, where μ^* is the average obtained by the optimal lever. Several algorithms have been studied that minimize the *regret*, defined as

$$\mu^* n - \mu_i \sum_{i=1}^{K} E[T_i(n)]$$

where $T_i(n)$ is the number of times arm i has been chosen. Building on the idea of *confidence bounds*, the UCB algorithm creates an overestimation of the reward of each possible decision, and lowers it as more samples are drawn. Implementing the principle of *optimism in the face of uncertainty*, the algorithm picks the option with the highest current bound. Interestingly, this allows UCB to achieve a logarithmic bound on the regret value not only asymptotically, but also for any finite sequence of trials [2].

More in detail, UCB assumes that rewards are distributed in the [0,1] interval, and associates each arm with a value:

$$\overline{\mu_i} = \overline{x_i} + \sqrt{2\frac{\log n}{n_i}} \tag{4}$$

where $\overline{x_i}$ is the current estimated reward for arm i, n is the number of the current trial, n_i is the number of times the level i has been tried. The right-hand part of the sum in Eq. 4 is an *upper confidence bound* that decreases as more information on each option is

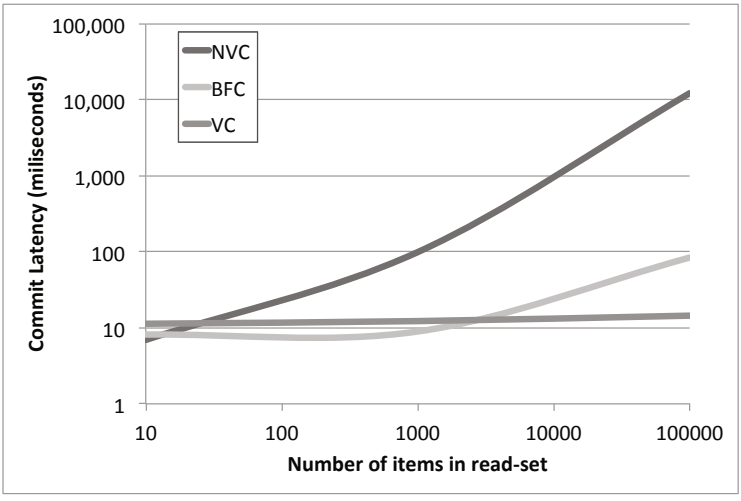


Fig. 4. Commit latency as a function of the read-set size

acquired. By choosing, at any time, the option with maximum $\overline{\mu_i}$, the algorithm searches for the option with the highest reward, while minimizing the *regret* along the way.

In order to apply the UCB technique to our problem, we had two face two main issues, which we discuss in the following paragraphs.

State space discretization. As we have illustrated in Section 2, the performance of certification depends on the workload characterization. Thus, using a single UCB instance, having as arms the three alternative protocols for all possible scenarios is clearly not a viable solution. This observation raises the problem of discretizing the workload state space into a set of distinct, representative, classes of workload scenarios. This allows, in fact, to associate a different instance of a UCB learner with each discretized interval of the workload's parameter space, and to train each instance to choose among the 3 considered protocols under specific workload conditions.

The discretization process involves a delicate trade-off: a finer (i.e. denser) discretization can lead, eventually, to more accurate predictions across the entire state space, but requires the training of a larger number of UCB instances, which can lead to a considerable increase of the learning time. In order to determine an appropriate discretization strategy, we analysed the average commit latency of each of the three protocols as a function of the read-set size using the synthetic benchmark introduced in Section 2. The results, reported in the log-log plot of Figure 4, highlight that, for NVC and BFC, the read-set size and commit latency exhibit a heavy-tail relationship. At the light of this observation, we opted to use the read-set size as the reference variable to discriminate different workload situations,and we discretized it using exponentially increasing intervals, where each sampling interval is defined by the range $[10^i, 10^i + 1]$ with $i \in \{1 \ldots 6\}$). This choice allowed us to partition the state space into a small number of intervals, thus reducing learning time, while associating each discretized interval with fluctuations of approximately the same relative amplitude in the commit latency,

even for the case of the NVC, whose commit latency is the most sensible to variations of the read-set size.

Definition of the reward function. UCB is based on the assumption that rewards are distributed in the [0,1] interval, whereas, as we have seen in Figure 4, the commit latencies are distributed over a very large domain. This required defining a mapping function, denoted as $R(t)$, taking as input a commit latency, t, and outputting a value (the reward) distributed in the [0,1] interval. In order to preserve the relative distance among samples before and after applying the mapping function we employed the following linear transformation:

$$R(t) = \frac{maxLatency - min\{maxLatency, t\}}{maxLatency}$$

which relies on the parameter $maxLatency$, defining a threshold for the commit latency, above which the reward is mapped to the value 0. Based on our preliminary experiments, we observed that the correct definition of the $maxLatency$ parameter value has a fundamental impact on the effectiveness of UCB: excessively low or high values would in fact lead to saturating the reward function, preventing UCB to distinguish sensibly the performance of the various protocols. Also, the manual tuning of this parameter is an extremely time-consuming task, given that the setting of $maxLatency$ was found to depend strongly on the characteristics of the user level application. For instance, we noted that, when testing this approach with the STMBench7 benchmark, we had to increase the value of $maxLatency$ by a factor approximately 27x larger than when using the synthetic benchmark described in Section 2.

This led us to define a self-tuning mechanism to define the value of the $maxLatency$ parameter. This mechanism is based on the observation that the (average) commit latency when using VC is i) largely unaffected by the read-set size (given that it does not disseminate the read-set), and ii) lower than that of *both* NVC and BFC for sizes of the read-set larger than some threshold (this threshold being unknown and dependant on the application and deployment scenario). In other words, VC's commit latency represents a consistent upper bound for NVC's and BFC's commit latencies below a given read-set's size threshold, in which the two protocols typically exhibit alternate performances. On the other hand, it represents a lower bound for NVC's and BFC's commit latency for high read-set's size, a scenario in which it is actually unnecessary to be able to accurately predict their performance, given that VC outperforms them significantly.

This makes the VC's average commit latency, denoted as T_{VC}, a good reference point for UCB's $maxLatency$ parameter value. This insight led us to define the following rule:

$$maxLatency = \overline{T_{VC}} \cdot (1 + \sigma(T_{VC}))$$

where $\sigma(T_{VC})$ denotes the standard deviation (more precisely the squared root of the sampling variance) of T_{VC}. In order to instantiate this formula, upon boot-strapping of the system, we execute transactions using the VC scheme until the following stopping condition is reached:

$$\sigma(T_{VC}) < 2 \cdot \overline{T_{VC}}$$

which in our experiments typically implied a few tens of transactions (and that was however upper bounded to 100 transactions to ensure robustness in the presence of highly disperse sampling data). To minimize the impact of this (typically quite short) bootstrapping phase on the learning time, we provide the observed sampling data to the corresponding UCB's instances also during this phase (in which UCB's instances are not being queried to choose the replication protocol), thus allowing them to gather statistical information concerning the reward of the arm associated with the VC protocol.

A further optimization that we designed in order to minimize learning time is to have the replicas periodically exchange and merge the locally gathered statistical information concerning the reward distributions of UCB's arms. This allows the replicas to mutually benefit from the statistical knowledge that they have gathered so far, narrowing the upper confidence bounds of the UCB's instances and accelerating their convergence. To minimize the overhead, we piggyback periodically (e.g. each 10 seconds in our experiments) the state of the 6 UCB's instances maintained at each replica (encoded by the tuple $< \overline{x_i}, n_i, n >$ for each of its three arms $i \in \{NVC, BFC, VC\}$, and globally accounting to around 100 bytes) to the AB messages generated by the PRM. As soon as updated statistical information from a different replica is received, the information concerning the local UCB instances is updated by setting, for each arm i:

- the value of $\overline{x_i}$ to the average of the local and remote values of $\overline{x_i}$, weighted proportionally to the number of times i was played locally and remotely, namely:

$$\overline{x_i} = w_i^{loc}\overline{x_i} + w_i^{rem}\overline{x_i^{rem}}, \text{ where } w_i^{loc} = \frac{n_i}{n_i + n_i^{rem}} \text{ and } w_i^{rem} = 1 - w_i^{loc}$$

- the value of n_i and n to the sum of their, respectively, local and remote values, namely $n_i = n_i + n_i^{rem}$ and $n = n + n^{rem}$.

5 Experimental Evaluation

In this section we report the results of an experimental study aimed at assessing the performance gains achievable by PolyCert, and the adequacy of the proposed machine-learning based self-optimizing mechanisms.

We start by considering the synthetic benchmark already used in Section 2 that, thanks to its simplicity and predictability, allows us to analyse the performance of PolyCert in precisely identifiable workload scenarios. We then evaluate a widely used benchmark for Transactional Memories, namely STMBench7, already mentioned in Section 2. STMBench7 is a complex benchmark that features a number of operations with different levels of complexity over an object-graph with millions of objects, generating a very intense and heterogeneous workload for the GCS. All the throughput results reported in the following were obtained averaging over a number of runs sufficient to ensure that the width of the 90% confidence intervals for the throughput was less than 10% of the corresponding average value.

The bar plot in Figure 5 reports the normalized throughput (with respect to the optimal non-adaptive workload) for each of the workloads generated by the Bank benchmark, including the versions of PolyCert when employing the oracles based on regressor decision trees and UCB (respectively dubbed as DT and DistUCB in the plot). These

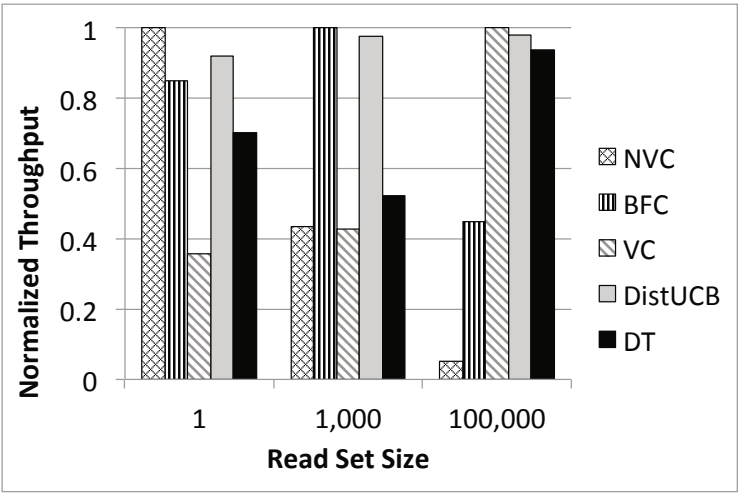


Fig. 5. Normalized throughput of the adaptive and non-adaptive protocols (Bank benchmark)

experiments were run by switching the workload every three minutes, thus the reported performance incorporates also data gathered during the initial phases during which we bootstrap the statistical information of UCB.

Our experimental data shows that the on-line learning oracle using UCB (with the optimization for periodically exchanging statistical information among replicas enabled) achieves a performance very close to the corresponding optimal protocol for each scenario, namely on average around 5% less than the optimal solution and in the worst case, the scenario where the transaction's read-set size is set equal to 1, less than 10% from the optimum. In this scenario, UCB alternates between BFC and NVC, whose performances are quite close (differing by around 15%); in several runs some replicas eventually converged towards the choice of BFC. In all the remaining scenarios, after a short bootstrapping phase, the replicas converged consistently towards the choice of the optimal certification protocols, which explains why they achieved performance almost indistinguishable from those of an optimally tuned non-adaptive protocol.

On the other hand, the performance achieved by the oracle based on regressor decision trees was significantly worse. When using DT, the performance of PolyCert was approximately 25% worse than that of the corresponding optimal non-adaptive scheme (across the three workloads). Note that, DT was still able to outperform the second best non-adaptive protocol (but not the optimal choice). A main source of inefficiency in the implementation of the DT oracle is the following: it relies on the Java Native Interface (JNI) to query the decision tree-based model generated by Cubist, implemented in C. The overheads due to JNI are negligible in the scenario with read-set size equal to 100K, whose transactions have a local execution time in the order of a few tens of milliseconds. On the other hand, JNI's overheads have a negative impact on performance in the scenarios with smaller read-set sizes, in which transactions have a local execution time on the order of just a few tens of microseconds. The performance of DT is lower in the

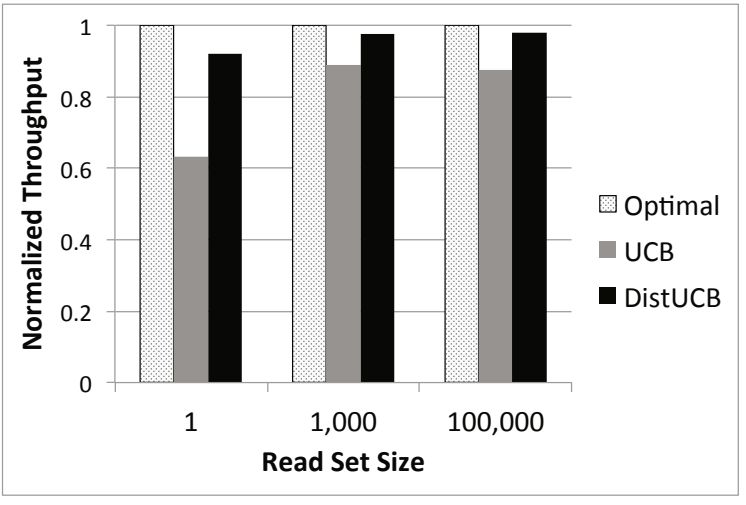


Fig. 6. Normalized throughput of UCB and DistUCB over a three minute run (Bank benchmark)

scenario with a read-set size of 1000, as in this case the DT oracle had a lower accuracy in forecasting the AB self-delivery time, and erroneously biased its decisions towards the voting protocol (which is chosen in approximately 30% of the cases on average).

In Figure 6 we contrast the performance of the UCB oracle (again in terms of normalized throughput vs the optimal non-adaptive protocol), over a three minute run, with and without enabling the optimization of exchanging periodically (each 10 seconds) statistical information among replicas to improve learning. The data clearly shows the effectiveness of this optimization, with speed-ups larger than 25% due to the fastest convergence towards the optimal non-adaptive solution. Figure 7 provides more detailed insights on the speed of convergence of UCB and DistUCB versus the optimal solution, reporting the average throughput over 10 seconds time windows, achieved by the two protocols. The plots clearly highlight the positive effects, in terms of learning time reduction, due to the exchange of statistical information occurring, in particular, at the time instants 10, 20 and 30 (seconds), that nearly halves the time required to converge to the optimal choice.

Finally, we assess the performance of PolyCert with STMBench7, plotting the corresponding results in Figure 8. The benchmark was configured to use the write-dominated workload with long traversals, which generates approximately 90% of update transactions, thus allowing us to focus on the performance of the transactions that require the activation of a commit-time certification phase. As shown in Figure 1, around 5% of transactions (namely the so-called *long traversal* transactions) in this benchmark have read-set sizes larger than 500K items. As a consequence, when using either NVC or BFC, this benchmark generates a very high traffic volume that, in all our runs, eventually determined the saturation and the collapse of the GCS. This is the reason why in Figure 8 we only report the throughput of VC, DT, UCB and DistUCB (normalized with respect to the throughput of the optimal non-adaptive protocol, namely VC). In this

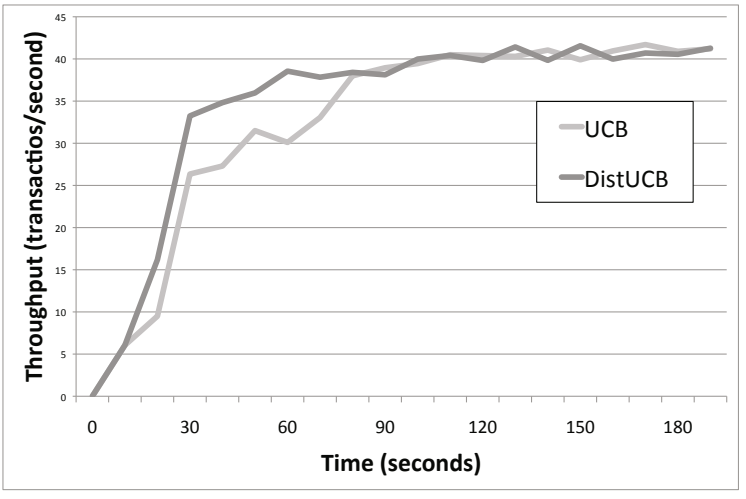


Fig. 7. Evolution of throughput over time with UCB and DistUCB (Bank benchmark - 100K read-set size scenario)

scenario, the adaptive protocols clearly outperform the non-adaptive VC scheme, thanks to their ability to use the more efficient NVC and BFC protocols to handle transactions with smaller read-set's size. The speed-up of PolyCert when using the three alternative oracles ranges from 25% to 35%, with the best performance also in this case achieved by DistUCB.

Overall, our experimental data demonstrated the effectiveness and viability of the proposed self-tuning polymorphic replication technique. The reported results highlight in particular the efficiency of the DistUCB oracle, which, not needing any time-consuming off-line training phases, and being totally parameter-free, results as extremely convenient for deployment in real-life practical scenarios. Interestingly, PolyCert does not only provide benefits in terms of performance, but also in terms of robustness, avoiding to saturate the GCS in presence of transactions with extremely large read-sets, a main source of instability for BFC and, in particular, NVC.

6 Related Work

Our work is clearly related to the vast literature on replication of transactional systems, and in particular to the more recent works relying on AB to achieve a replica-wide agreement on the transaction serialization order [18,24,17,23]. All these protocols adopt a single static strategy, unlike PolyCert which, not only allows for the simultaneous coexistence of multiple certification strategies, but autonomically determines, on a per-transaction basis, the most adequate replication protocol to employ using machine-learning techniques.

Machine learning techniques have already been used to predict the performance of computer systems in several contexts. These include works aiming at forecasting the

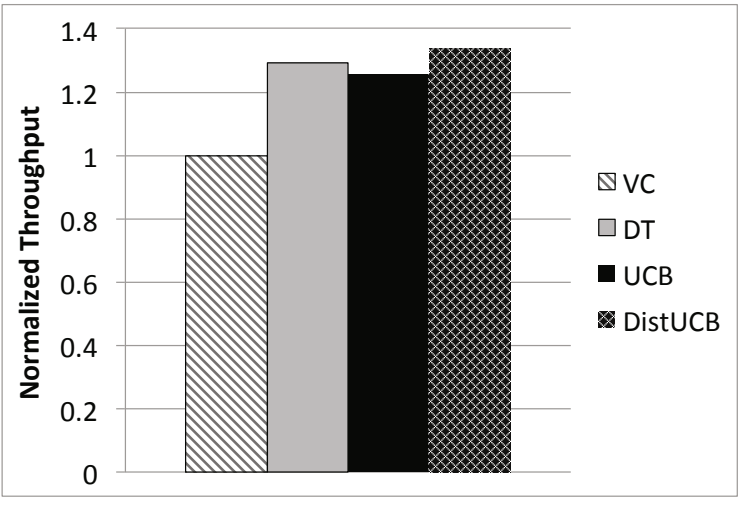


Fig. 8. Normalized throughput of the adaptive and VC protocols. NVC and BFC not reported as they caused the collapse of the GCS layer. (STMBench7 benchmark).

throughput of TCP flows [22] and Pub-Sub systems [11], solutions aimed at automatically classifying traffic based on semi-supervised learning techniques [10], at automatizing the allocation of resources in cloud-computing infrastructures [33], or generating software aging models to be used in the context of rejuvenation frameworks [1]. Also, as noted in the text, the regressor decision tree oracle exploits our previous results in the area of machine-learning performance prediction of AB protocols, recently published in [9].

Our work is clearly related to the body of research on autonomic computing, and in particular to the field of self-optimizing databases. In this context, several approaches have been proposed based on the idea to automatically analyse the incoming workload, e.g. [20], to automatically identify the optimal database physical design or self-tune some of the DBMS inner management schemes, e.g. [5]. However, none of these approaches investigated the issues related to autonomically adapt the replication scheme. We argue that this is mainly due to the fact that current DBMSs, because of the high complexity of their architecture, lack the flexibility required to dynamically adapt such low level mechanisms.

7 Conclusions

Replication is of uttermost importance for in-memory NoSQL data platforms, which are emerging as an attractive alternative to conventional relational distributed databases. However, since the parameter space defining the workload of transactional applications is extremely wide, it is extremely challenging to devise universal transactional replication solutions capable of guaranteeing optimal performance in any possible scenario. In this paper we proposed, to the best of our knowledge for the first time in literature, a

self-tuning adaptive scheme, which we named PolyCert, that allows for the simultaneous coexistence of multiple AB-based certification schemes. PolyCert uses parameter-free machine learning techniques to determine the optimal replication strategy to use on a per-transaction basis. The self-tuning strategy of PolyCert allows to achieve significant speed-ups when compared with non-adaptive certification protocols. Furthermore, it also improves the robustness of the replicated data platform, avoiding to saturate the GCS in the presence of transactions with extremely large read-sets, a main source of instability for several certification protocols.

References

1. Andrzejak, A., Silva, L.: Using machine learning for non-intrusive modeling and prediction of software aging. In: Proc. of the Network Operations and Management Symposium (NOMS), pp. 25–32. IEEE, Salvador de Bahia, Brazil (2008)
2. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning 47, 235–256 (2002)
3. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley Longman Publishing Co., Inc., Boston (1986)
4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
5. Bruno, N., Chaudhuri, S.: An online approach to physical design tuning. In: Proc. of the International Conference on Data Engineering (ICDE), pp. 826–835 (2007)
6. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Science Computer Programming 63(2), 172–185 (2006)
7. Carvalho, N., Romano, P., Rodrigues, L.: Asynchronous Lease-Based Replication of Software Transactional Memory. In: Gupta, I., Mascolo, C. (eds.) Middleware 2010. LNCS, vol. 6452, pp. 376–396. Springer, Heidelberg (2010)
8. Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D2STM: Dependable distributed software transactional memory. In: Proc. of the Pacific Rim International Symposium on Dependable Computing (PRDC), Shanghai, China, pp. 307–313 (2009)
9. Couceiro, M., Romano, P., Rodrigues, L.: A machine learning approach to performance prediction of total order broadcast protocols. In: Proc. of the International Conference on Self-Adaptive and Self-Organizing Systems (SASO), Budapest, Hungary, pp. 184–193 (2010)
10. Erman, J., Mahanti, A., Arlitt, M., Cohen, I., Williamson, C.: Offline/realtime traffic classification using semi-supervised learning. Performance Evaluation 64(9-12), 1194–1213 (2007)
11. Garces-Erice, L.: Admission control for distributed complex responsive systems. In: Proc. of the International Symposium on Parallel and Distributed Computing (ISPDC), pp. 226–233. IEEE Computer Society, Washington, DC (2009)
12. Gray, J., Helland, P., O'Neil, P., Shasha, D.: The dangers of replication and a solution. In: Proc. of the International Conference on Management of Data (SIGMOD), pp. 173–182. ACM, New York (1996)
13. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: a benchmark for software transactional memory. SIGOPS Operating Systems Review 41(3), 315–324 (2007)
14. Guerraoui, R., Rodrigues, L.: Introduction to Reliable Distributed Programming. Springer, Heidelberg (2006)
15. Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. The Journal of Machine Learning Research 3, 1157–1182 (2003)
16. Haykin, S.: Neural Networks: A Comprehensive Foundation. Prentice Hall PTR, Upper Saddle River (1994)

17. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In: Proc. of the Very Large Data Base Conference (VLDB), pp. 134–143. ACM, Cairo (2000)
18. Kemme, B., Alonso, G.: A suite of database replication protocols based on group communication primitives. In: Proc. of the International Conference on Distributed Computing Systems (ICDCS), p. 156. IEEE Computer Society (1998)
19. Martin, M., Blundell, C., Lewis, E.: Subtleties of transactional memory atomicity semantics. IEEE Computer Architecture Letters 5(2), 17 (2006)
20. Martin, P., Elnaffar, S., Wasserman, T.: Workload models for autonomic database management systems. In: Proc. of the International Conference on Autonomic and Autonomous Systems (ICAS), p. 10. IEEE Computer Society, Washington, DC (2006)
21. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a flexible protocol kernel supporting multiple coordinated channels. In: Proc. of the International Conference on Distributed Computing Systems (ICDCS), pp. 707–710. IEEE, Phoenix (2001)
22. Mirza, M., Sommers, J., Barford, P., Zhu, X.: A machine learning approach to TCP throughput prediction. In: Proc. of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), pp. 97–108. ACM, New York (2007)
23. Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B., Alonso, G.: Scalable Replication in Database Clusters. In: Herlihy, M.P. (ed.) DISC 2000. LNCS, vol. 1914, pp. 315–329. Springer, Heidelberg (2000)
24. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Distributed and Parallel Databases 14(1), 71–98 (2003)
25. Quinlan, J.R.: Cubist, http://www.rulequest.com/cubist-info.html
26. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc., San Francisco (1993)
27. Robbins, H.: Some aspects of the sequential design of experiments. Bulletin of the American Mathematical Society 58(5), 527–535 (1952)
28. Romano, P., Rodrigues, L., Carvalho, N., Cachopo, J.: Cloud-TM: Harnessing the cloud with distributed transactional memories. SIGOPS Operating Systems Review 44, 1–6 (2010)
29. Schiper, N., Sutra, P., Pedone, F.: P-Store: Genuine partial replication in wide area networks. In: Proc. of the Symposium on Reliable Distributed Systems (SRDS), pp. 214–224. IEEE Computer Society, Washington, DC (2010)
30. Schneider, F.B.: Replication management using the state-machine approach. ACM Press/Addison-Wesley Publishing Co. (1993)
31. Shevade, S.K., Keerthi, S.S., Bhattacharyya, C., Murthy, K.R.K.: Improvements to the SMO algorithm for SVM regression. IEEE Transactions on Neural Networks 11(5), 1188–1193 (2000)
32. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era: (it's time for a complete rewrite). In: Proc. of the International Conference on Very large Data Bases (VLDB), pp. 1150–1160. VLDB Endowment (2007)
33. Xu, J., Zhao, M., Fortes, J., Carpenter, R., Yousif, M.: Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. Cluster Computing 11(3), 213–227 (2008)

A Trigger-Based Middleware Cache for ORMs

Priya Gupta*, Nickolai Zeldovich, and Samuel Madden

MIT CSAIL

Abstract. Caching is an important technique in scaling storage for high-traffic web applications. Usually, building caching mechanisms involves significant effort from the application developer to maintain and invalidate data in the cache. In this work we present CacheGenie, a caching middleware which makes it easy for web application developers to use caching mechanisms in their applications. CacheGenie provides high-level caching abstractions for common query patterns in web applications based on Object-Relational Mapping (ORM) frameworks. Using these abstractions, the developer does not have to worry about managing the cache (e.g., insertion and deletion) or maintaining consistency (e.g., invalidation or updates) when writing application code.

We design and implement CacheGenie in the popular Django web application framework, with PostgreSQL as the database backend and `memcached` as the caching layer. To automatically invalidate or update cached data, we use triggers inside the database. CacheGenie requires no modifications to PostgreSQL or `memcached`. To evaluate our prototype, we port several Pinax web applications to use our caching abstractions. Our results show that it takes little effort for application developers to use CacheGenie, and that CacheGenie improves throughput by 2–2.5× for read-mostly workloads in Pinax.

1 Introduction

Developers of popular web applications often struggle with scaling their application to handle many users, even if the application has access to many server machines. For stateless servers (such as HTTP front-ends or application servers), it is easy to spread the overall load across many machines. However, it is more difficult to add database servers, since partitioning the data over multiple machines is non-trivial, and executing queries across multiple machines can be expensive in itself. Database replication solves this problem well in read extensive workloads, but does not work well in write-heavy loads such as social networks, which we concentrate on in this work. Web application developers typically solve this problem by adding caching middleware in front of the database to cache the results of time-consuming queries, such as queries that span multiple servers, complicated aggregate queries, or small but frequent queries. Thus, caching forms an important part of storage systems of many web applications today; for example, many websites use `memcached` [8] as a distributed in-memory caching system.

However, popular caching solutions such as `memcached` offer only a key-value interface, and leave the application developer responsible for explicitly managing the cache. Most importantly, the developer must manually maintain cache consistency by invalidating cached data when the database changes. This has several disadvantages. First, developers have to write a significant amount of code to manage the application's caching

* Google

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 329–349, 2011.

layer. Second, this code is typically spread all over the application, making the application difficult to extend and maintain. It is also a common source of programming errors; for example, a recent outage of Facebook was caused by an error in application code that tried to keep `memcached` and MySQL consistent [13]. Finally, the developers of each application independently build these caching mechanisms and cannot re-use other developers' work, due to the lack of common high-level caching abstractions.

This work aims to address these issues with *CacheGenie*, a system that provides higher level caching abstractions for automatic cache management in web applications. These abstractions provide a declarative way of caching, where the developers only specify what they want to cache and the desired consistency requirements, and the underlying system takes care of maintaining the cache.

The first goal of CacheGenie is to relieve programmers of the burden of cache management. CacheGenie does three things to achieve this. First, it generates database queries based on object specifications from the developer; these queries are used to get the result from the underlying database, which is then cached. Second, whenever possible, CacheGenie transparently uses the cached object instead of executing the query on the database. Finally, whenever underlying data a cached object depends on is changed, it transparently invalidates or updates the cached object. This is done by executing database triggers when the underlying data changes.

The second goal of CacheGenie is to avoid making any modifications to the database or cache, and to use existing primitives present in modern databases (i.e., triggers) in order to ensure cache consistency. This makes it easy for developers to start using CacheGenie and to reuse existing components.

The third and final goal of CacheGenie is to minimize the performance overhead of maintaining a consistent cache. Any database-cache synchronization approach will incur a combination of two overheads: updating/invalidating the cache in response to database writes and sending read requests to the database if lookups in the cache fail. In CacheGenie, we minimize the first overhead by incrementally updating the cache from within the database, since the database has all the information regarding which keys to update and how. The second overhead is minimized through the use of an *update* approach where, once data is fetched into the cache, it is kept fresh by propagating incremental updates from the database (unless evicted due to lack of space). In contrast, in an *invalidation* approach, data is removed from the cache due to writes to database and must be completely re-fetched from the database. In our experiments we show that in CacheGenie, the overhead of maintaining a consistent cache is less than 30% (versus an approach in which no effort is made to synchronize the cache and database).

Our implementation of CacheGenie is done in the context of Object-Relational Mapping (ORM) systems. In recent years, ORM systems like Hibernate [12] (for Java), Django [7] (for Python), Rails [11] (for Ruby) and others have become popular for web application development. ORM systems allow applications to access the database through (persisted) programming language objects, and provide access to common database operations, like key lookups, joins, and aggregates through methods on those objects. ORM systems help programmers avoid both the "impedance mismatch" of translating database results into language objects, and the need for directly writing SQL. Because most accesses to the database in an ORM are through a small number of

interfaces, we focused on supporting these interfaces. While CacheGenie does not cache non-ORM operations, all updates to the database (through ORM and otherwise) are propagated to the cache through database triggers, ensuring cache consistency. Our current implementation of CacheGenie propagates updates to the cache non-transactionally (i.e., readers can see dirty data, but not stale data), and serializes all writes through the database to avoid write-write conflicts.[1] CacheGenie provides caching abstractions for common query patterns generated by the ORM.

We have implemented a prototype of CacheGenie for Django, PostgreSQL, and `memcached`. To evaluate CacheGenie, we ported several applications from Pinax [26] (a suite of social networking applications coded for Django) to use CacheGenie. Our changes required modifying only 20 lines of code; CacheGenie automatically generated 1720 lines of trigger code to manage the cache. Our experiments show that using CacheGenie's caching abstractions leads to a 2–2.5× throughput improvement compared to a system with no cache.

This paper makes four contributions. First, we describe CacheGenie, a novel and practical system for automatic cache management in ORMs, which works with an unmodified database and `memcached`. Second, we introduce new *caching abstractions* that help programmers declare the data they wish to cache. Third, we use a trigger-based approach to keep the cache synchronized with the database. Finally, we evaluate CacheGenie on a real-world web application and show the benefits of incremental updates and invalidations using our approach.

The rest of this paper is organized as follows: §2 gives a background of current caching strategies and discusses some important related work in this area. §3 explains the concept of caching abstractions, how we support them, and discusses the consistency guarantees offered by CacheGenie. §4 describes our implementation using Django, Postgres and `memcached`. §5 discusses our experience with porting Pinax applications to use CacheGenie, and describes our experimental results. Finally, §6 concludes.

2 Background and Related Work

Web applications employ several caching strategies to improve their performance and reduce the load on the underlying data store. These strategies can be divided into two main categories: *application caching* and *database caching*.

The first category refers to application-level caching of entire HTML pages, page fragments or computed results. This scenario is illustrated by Figure 1a. In this scenario, the web application is responsible for cache management, and typically uses a key-value store, such as `memcached`, as the cache. Cache management includes (i) choosing the granularity of cache objects, (ii) translating between database queries and cache objects, so that they can be stored in a key-value store, and (iii) maintaining cache consistency.

With application-level caching, the cache and the underlying database are not aware of each other, and cache management is the application developer's burden. The advantage of application-level caching is that it allows for caching at a granularity best suited to the application. The disadvantage is that application developers have to manually implement cache management themselves via three possible options. The first option

[1] We include a description of how to extend CacheGenie with full serializability in §3.3.

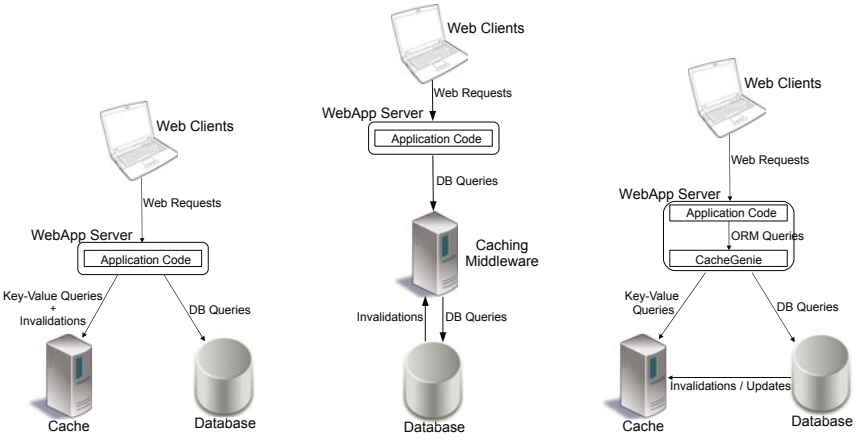


Fig. 1. Different approaches to caching in web applications

is to **expire** cached data after a certain interval of time. Although this mechanism is easy to use, it is insufficient for highly dynamic websites, and coming up with the *right* expiration time is often difficult. The second option is **manual invalidation**, where the programmer has to keep track of all possible writes to the underlying database and determine which updates could affect what data in the cache. This can be cumbersome and error-prone. The third option is a **write-through** cache. In this case, every time the programmer writes code to update data in the database, she must also write code to update the cache. Since the data in the cache is not invalidated but updated in place, this can increase the number of cache hits. However, sometimes the application might not have enough information to determine which entries from the cache should be updated; this might lead to additional queries to the database, making the updates slower.

The second category, database caching, is illustrated in Figure 1b. In this model, a middleware layer caches partial or full data from the database near the application servers to reduce the load on the database server. In some cases, the cached data can be partial rows returned from the database against which further queries are executed [1, 3, 18, 20]. In this case, the middleware layer is responsible for deciding what to cache, how to satisfy the application requests based on what is in the cache, and maintaining cache consistency with the underlying data. Though this model frees the developer from managing the cache, it can result in sub-optimal caching behavior since cached objects are typically database rows and not application-level objects.

A simple version of database caching is to cache results of exact queries, and return the same results for identical future queries, as in GlobeCBC [25]. To maintain cache consistency, template-based invalidation schemes are typically used (templates are used because the problem of determining whether two queries touch the same data is hard [14]). Update queries are executed at the database server, and when a cache server stores the results of a query, it subscribes to receive invalidations based on conflicting query templates. There are two limitations with this model. First, the programmer must specify *a priori* all pairs of conflicting query and update templates; this can be time-

consuming and error-prone. Second, if one update can potentially affect another query, all cached results belonging to the corresponding query template are invalidated. This can lead to poor cache hit ratios, and thereby increase server load.

None of the above approaches fully solve the problem of caching in web applications. CacheGenie combines the best parts of these approaches into a system that is most beneficial for the programmer. CacheGenie provides high-level caching abstractions that programmers can use without making substantial changes to the application, database system, or caching layer. CacheGenie caches query results and automatically stores and updates those results, as opposed to providing a key-value store that the programmer must manually manage. The caching abstractions determine the granularity of caching, and automate translation between the data in the cached objects and the data stored in the underlying database. Unlike a template-based system, CacheGenie only invalidates cached data that is affected by writes to the database (see §3.2). This leads to fewer invalidations and higher cache hit ratios. The high-level architecture of CacheGenie is illustrated in Figure 1c. CacheGenie operates as a layer underneath the application, modifying the queries issued by the ORM system to the database, redirecting them to the cache when possible.

There are several other systems that provide automatic cache management. Labrinidis et al. [17] present a survey of the state-of-the-art in caching and materialization in web application databases. Cache invalidation as a strategy to maintain strong cache consistency has been explored in numerous prior works [19, 21]. Challenger [5] proposed using a dependency graph between cached objects and underlying data to regenerate or invalidate relevant HTML pages or fragments in the cache. However, the query workload that they consider was mostly reads, with few writes; a higher write fraction in an invalidation based system will lead to poor cache-hit ratio. Degenaro et al. [6] explored a similar approach in the context of generic query result caching. Ferdinand [9] provides a disk-based cache of query results, which uses a publish-subscribe model to achieve consistency in a scalable distributed manner. Using a pub-sub model in CacheGenie could improve scalability of propagating cache updates. A key difference between CacheGenie and these systems is that CacheGenie updates cached data in-place.

Several systems leverage snapshot isolation for caching. TxCache [24] provides a transactional cache, and ensures that any data seen within a transaction, whether it comes from the cache or the database, reflects a slightly stale but consistent database snapshot. TxCache lets programmers designate specific functions as cacheable; it automatically caches their results, and invalidates the cached data as the underlying database changes. Unlike TxCache, CacheGenie performs in-place updates instead of invalidation, but can cache and update only pre-determined functions (caching abstractions) instead of arbitrary functions. CacheGenie also relaxes transactional guarantees to allow the application to access fresh data, as many web applications do not require or use strong transactional guarantees. SI-cache [22, 23] similarly extends snapshot isolation to caching in J2EE applications. The key difference in CacheGenie, in addition to the above, is that CacheGenie maintains a single logical cache across many cache servers. In SI-cache, each application server maintains its own cache, and if many servers cache the same data, the total effective cache capacity is greatly reduced.

TimesTen [27] allows for caching partial tables from the database in their in-memory store, providing a SQL-like syntax to the developer to specify which partial tables to cache. It allows for updates to flow from backend database to the cache using database triggers. However, the updates flow to the cache only periodically (the period of refresh specified by the application) and hence the application might see stale data. This is unlike CacheGenie, which ensures synchronous propagation of updates. Further, TimesTen caches raw table fragments, and hence requires extra computation at query time. AutoWebCache [4] uses aspect-oriented techniques to implement a caching middleware for dynamic content for J2EE server-side applications. Unlike CacheGenie, AutoWebCache caches entire web pages, and uses template-based invalidation for cache consistency.

There has been a lot of work exploring materialized views in databases and algorithms to incrementally update them. Materialized views are also useful in pre-computing and thus providing fast access to complex query results. The problem of incremental view maintenance is similar to the problem of maintaining up-to-date cached query results. However, unlike materialized views in the database, CacheGenie is geared towards maintaining views in a more distributed scenario. Moreover, CacheGenie scales better because it transfers the load from the database to a distributed cache. CacheGenie employs techniques similar to view maintenance [10] and materialization of entire web pages [16], except that in CacheGenie, "materialization" happens in a third caching layer, and CacheGenie caches more granular data than whole web pages.

A recent system called TAO by Facebook works in a way similar to ours, by letting programmers think in terms of high-level abstractions rather than SQL, and automatically managing the cache for them. An important difference however is that they perform write-through caching, whereas we propagate the updates through the database. CacheMoney [15] is a library for Ruby on Rails [11], which enables write-through and read-through caching to `memcached`; unlike CacheGenie, it does not support joins.

Table 1 summarizes the relation of CacheGenie to several representative systems.

Table 1. Comparison of CacheGenie with representative related systems

System	Cache granularity	Source code modifications	Stale data	Cache coherence
`memcached`	Arbitrary	Every read	Yes	None
`memcached`	Arbitrary	Every read + write	No	Manual invalidation
TxCache	Functions	None	Yes (SI)	Invalidation / timeout
TimesTen	Partial DB Tables	None	Yes	Incremental update-in-place
GlobeCBC	SQL queries	None	No	Template-based inv.
AutoWebCache	Entire webpage	None	No	Template-based inv.
CacheGenie	Caching abstractions	None	No	Incremental update-in-place

3 Design

In this section, we describe the programmer's interface to CacheGenie's caching abstractions and then describe how they are implemented internally using triggers.

3.1 Caching Abstractions

Rather than trying to provide a generic query caching interface, CacheGenie's goal is to cache common query patterns generated by ORMs like Django. The workloads may also have infrequent queries that are not handled by the patterns CacheGenie supports (e.g., because Django allows applications to directly write SQL queries). However, to improve performance, it typically suffices to improve these commonly occurring queries, and CacheGenie's approach does not require that all queries be mediated by the caching layer. Working within these frameworks also ensures that the programmer does not have to change the programming model she is using.

ORM-based web applications generate database queries using objects (such as an object representing an entire table, called models in Django) and functions (such as filtering objects based on the certain clauses). The programmer issues queries by calling functions on objects; the ORM framework issues SQL queries to the database in response to these function calls. CacheGenie provides caching abstractions called *cache classes* for common query patterns generated by the ORM. Each query pattern is represented by one cache class. To cache data pertaining to different entities but following the same pattern, the programmer defines multiple instances of the corresponding cache class, and each instance is called a *cached object*. Once a cached object is defined, the programmer can simply use existing object code, and CacheGenie takes care of fetching the right data from the cache.

We explain the concept of cache classes with an example from the social networking domain. Imagine that a developer wants to fetch the profile data of users in the application. To get this data from the database in Django, the developer would write the following code (assuming models for `User`, which contains administrative information about users, and `Profile`, which contains detailed information entered by the user, have already been created. The `Profile` is related to model `User` by the `user_id` field):

```
profile = Profile.objects.get(user_id=42)
```

To cache this data in Django, the developer has to manually put/get the profile data into the cache wherever needed in the application, and manually invalidate the cached data wherever profile of a user is modified, using statements such as the following:

```
from django.core.cache import cache
cache.set('profile:42', user_profile, 30) # Cache user 42's profile, 30s expiry
cache.get('profile:42')                   # Get data from cache
cache.delete('profile:42')                # Invalidate cached copy
```

In CacheGenie, this query falls under the FeatureQuery cache class (described below). To cache this data, the developer only needs to add a cached object definition once:

```
cached_user_profile = cacheable(cache_class_type = 'FeatureQuery',
    main_model = 'Profile',                 # Main Model to cache
    where_fields = ['user_id'],             # Indexing column
    update_strategy = 'update-in-place',    # optional arguments
    use_transparently = True)               # optional arguments
```

Once this definition is made, CacheGenie automatically and transparently manages profile data in cache. The application code to access the cached data remains exactly same

as before (i.e., it is the same as getting the object from the database), and the developer does not have to write any additional code to manage the cache. As we do not provide strict transactional consistency in CacheGenie (see §3.3), the programmer can set `use_transparently` to false for objects which might need such guarantees. In that case, CacheGenie will not transparently fetch the cached data. The programmer can manually call `evaluate` on the returned cached object `cached_user_profile` with the `id` of the desired user to get the cached data manually:

```
profile = Profile.objects.get(user_id=42)           # get from cache or db
profile = cached_user_profile.evaluate(user_id=42)  # explicit cache lookup
```

CacheGenie supports the following cache classes:

1. **Feature Query** involves reading some or all features associated with an entity. In relational database terms, it means reading a (partial or full) row from a table satisfying some clause—typically one or more `WHERE` clauses. For example, in a social networking application which stores profile information in a single table, the query to get the profile information of a user, identified by a `user_id`, is a Feature Query. Since these queries make up a large percentage of many workloads, caching them is often beneficial.
2. **Link Query** involves traversing relationships between entities. In relational database terms, these queries involve traversing foreign key relationships between different tables. Since they involve joins, Link Queries are typically slow; caching frequently executed Link Queries is often beneficial. An example of a frequent join query in a social networking app is to look up information about the interest groups to which a user belongs. This query involves a join between the `groups_membership` table and the `groups` table. To create an instance of the `LinkQuery` cache class, the application must specify the chain of relationships to be followed.
3. **Count Query** caches the count of rows matching some predicate. A typical web application's page displays many types of counts, for example, a user's Facebook page displays counts of her friends, messages in the inbox, unread notifications and pending friend requests. Count queries are good candidates for caching, as they take up little memory in cache but can be slow to execute in the database.
4. **Top-K Query** caches the top K elements matching some predicate. In database terms, a top-K query involves sorting a table (or a join result) and returning the top K elements. Top-K queries are often expensive, and their results should be cached when possible. One important property of Top-K queries is that the cached results can be incrementally updated as updates happen to the database, and don't need to be recomputed from scratch, and CacheGenie exploits this property. An example of a Top-K query is fetching latest 20 status updates of a user's friends. Additional parameters for a Top-K query are the sort column, the order of sorting, and the value K.

In addition to class specific parameters, the programmer can also specify the cache consistency strategy, which can be (i) invalidate the cached object, or (ii) update it in-place (default). Since cache objects are defined based on existing queries in the application, it should be possible to derive these definitions automatically from the application, either by looking at SQL queries generated, or analyzing the ORM queries. The developer would still be able to choose the characteristics such as cache consistency requirements. We would like to explore this in future work.

Although CacheGenie implements cache classes for only a few query patterns, it is easy to extend the same concepts to other types of queries. Note, however, that cache classes correspond to queries common to multiple applications, and hence are reusable. Each developer does not have to write their cache classes, unless they want to cache a query pattern very specific to their application. To give an idea of what it takes to write a new cache class, we list here the functions it must perform:

1. **Query generation** uses the models and fields in the cached object to derive the underlying query template to get that object from the database. Note that we cache the raw results of queries and not Django model objects constructed from them.
2. **Trigger generation** involves determining which database tables and operations need triggers to keep the cached object consistent with the database. It also includes generation of the necessary code for the triggers, as described in §3.2.
3. **Query evaluation** involves fetching the appropriate data from the cache and transforming the returned value into the form required by the Django application, when the application requests it. If the key is not present in the cache, the cache class must query the database with the query generated during the generation step, add the result to cache, and return the appropriate transformed values to the application.

As an example, the definition of LinkQuery Class is as follows, with each function executing one task from the above.

```
class LinkQuery(CacheClass):
    def __init__(self, *args, **kwargs): # Initialize, Implement Step 1.
    def get_trigger_info(self):          # Implements Step 2.
    def evaluate(self, *args, **kwargs): # Implements Step 3.
    def make_key(self, *args, **kwargs): # Returns the corresponding key.
```

3.2 Database Triggers

Writes in CacheGenie are sent directly to the database, where it uses database triggers to automatically sync the cached data with these changes. In CacheGenie, for each cached object, there are three triggers—for insertion, deletion and update—generated on each of the tables underlying the cached object. These triggers are automatically generated from the cached object specifications. The programmer does not need to manually write the triggers, or specify *a priori* the cached objects that may be affected by each write query. When fired, the trigger code determines which cached entries, if any, can be affected by the modified data. It then modifies or invalidates these entries appropriately.

If the programmer chooses to invalidate cached objects, the trigger code invalidates *only* those entries of the cached object which are affected by this change. For example, imagine that the profile information of users with `user_id` 42 and 43 is currently in the cache. A query that updates the profile information of user 42 causes only the cached entry for user 42 to be invalidated, and the cached entry for user 43 remains unchanged. Note that this is different from template-based cache consistency mechanisms, which invalidate both the user profiles since they both match the same template.

Invalidation makes the trigger code simple, but invalidating frequently used items can lead to a poor cache-hit ratio. A better solution may be to update the cached data in response to the update. In this approach, the trigger code determines which entries in the cache could be affected by the data change in the table, figures out how to update

the cached data incrementally, and finally updates the relevant cached objects. Continuing with the previous example, if an `UPDATE` query updates the profile information of user 42, the cached entry for user 42 is updated with the new profile information and is available to any future request from the application. The problem of figuring out how to update a cached object is similar to the problem of incrementally updating a materialized view. This problem has been previously studied, and is hard to solve in general. However, because CacheGenie supports a few fixed types of query patterns, it becomes less computationally intensive compared to solving it for a general view. View maintenance techniques [10] can be applied to incrementally update cached objects other than the ones supported by CacheGenie.

The generated trigger operates in four steps. First, the trigger gets the modified rows from the database as input. Second, based on the modified rows (depending on the cached object for which this trigger is responsible), it figures out which keys in the cache can be affected by these rows. Third, if the cache consistency strategy is to update the cache, it queries the cache for the affected keys and calculates new values for them. Fourth, again depending on the cache strategy, it either deletes the affected keys from the cache or updates them in place with the new computed values.

To illustrate how triggers are generated in more detail, we provide a detailed example of how a trigger for a Top-K Query cache class is generated. To cache a `TopKQuery` object, CacheGenie caches an ordered list of results in `memcached` for the underlying query. The list contains K elements, as specified by the programmer when defining the cached object, plus a few more, to allow for incremental deletes. Consider the example of a Facebook-like social networking application where each user has a `wall` where any friend can post a note for this user. Let the `wall` table schema be:

```
wall (post_id int, user_id int, content text, sender_id int, date_posted date)
```

Suppose the developer wants to create a Top-K cached object for the latest 20 posts on a user's wall. The cached object definition that the developer writes for this would be:

```
latest_wall_posts = cacheable(cache_class_type = 'TopKQuery',
    main_model = 'Wall', where_fields = ['user_id'],
    sort_field = 'date_posted', sort_order = 'descending', k = 20)
```

For this cached object, CacheGenie automatically generates three triggers on the `wall` table (one each for `INSERT`, `DELETE` and `UPDATE`). When a new post gets inserted into the table, the corresponding trigger runs and gets the new post as input. From this inserted row, CacheGenie's trigger finds the `user_id` whose wall the post belongs to (say, 42), and then determines the key in the cache that will be affected, say `LatestWallPostsOfUser:42` (we use this key prefix for the sake of illustration; in practice a system-generated unique prefix is used). Next, assuming the update strategy, CacheGenie's trigger queries `memcached` for this key. If not present, the trigger quits. Otherwise, it finds the correct position of the new post in the cached list of posts according to `date_posted` and modifies the list accordingly. Finally, it puts the key back in the cache with the new modified value. The actual generated Python code for this trigger is:

```
cache       = memcache.Client(['host:port'])
table       = 'wall'
key_column  = 'user_id'
sort_column = 'date_posted'
```

```
new_row   = trigger_data['new']
cache_key = 'LatestWallPostsOfUser:' + new_row[key_column]
(cached_rows, cas_token) = cache.gets(cache_key)

if cached_rows is not None: # if present, update
  new_sort_val = new_row[sort_column]
  insert_pos = 0
  for row in cached_rows:
    if new_sort_val > row[sort_column]:
      break
    insert_pos += 1
  if insert_pos < len(cached_rows):  # update
    cached_rows.remove(len(cached_rows) - 1)
    cached_rows.insert(insert_pos, new_row)
    cache.cas(cache_key, cached_rows, cas_token)
    # not shown is retry when CAS fails
```

The trigger for `DELETE` similarly determines whether the deleted row exists in the list, and if so, deletes it. For top-K queries, CacheGenie fetches a few additional rows beyond the top K to support deletes without immediate re-computation. When this reserve is exhausted, CacheGenie have to recompute the entire list for caching. `UPDATE` triggers simply update the corresponding post if it finds it in the cached list.

More details of how triggers for other cache classes are automatically generated are left out for the sake of brevity. Next, we discuss the consistency guarantees offered by CacheGenie and contrast it with those provided by existing caching systems.

3.3 Consistency Guarantees

We have already described the basic mechanisms provided in CacheGenie to enable cache consistency. In this section we discuss the consistency guarantees CacheGenie provides with these mechanisms. First, CacheGenie performs atomic cache invalidations/updates for any single database write. Second, CacheGenie provides immediate visibility of a transaction's own updates. All cached keys affected by a write query are updated as a part of that statement, and hence the user sees the effects of her own writes immediately after the query is executed. This is a highly desirable property even for web applications since users expect to see their own updates immediately.

Currently, CacheGenie does not extend database transactions to the caching layer, because there were few conflicts in our workload (we have omitted the experiment which measures this for lack of space). The implication of this is that a transaction may read the results of an uncommitted update of another transaction from the cache (because writes are still done in the database, write-write conflicts are prevented). Another reason we chose not to add transactional support to the cache is that doing so would require changes to `memcached`, which we tried to avoid. We note that other database caches, like DBCache [3] and DBProxy [1] also provide relaxed transactional semantics; similarly, the application caching systems we know of also provide a weak consistency model.

However, for completeness, we describe a method for supporting full transactional consistency in CacheGenie. The key observation is that the database cannot determine whether a pair of concurrent transactions conflict because it does not see all the read operations. Thus, CacheGenie would track the readers and writers of keys in `memcached` and block transactions from completing their reads/writes to `memcached` according to the rules of two-phase locking.

In our design, when a transaction begins, the application and database decide on a transaction id, *tid*. Whenever a database trigger issues any updates/invalidations to `memcached` as a part of a transaction, it includes *tid*. For each key k, `memcached` would keep track of $readers_k$ (a list of the *tid*s of all uncommitted transactions that read k), and $writer_k$ (the id of the uncommitted writer of k, if any). Note that a database write to a row r can affect multiple keys in the cache, and a given key k may be a cached result corresponding to many rows in the database, so the keys associated with these readers and writers do not have an exact correspondence with rows in the database.

When using invalidations, we need to keep the readers and writers for invalidated keys, even if the keys themselves have been removed from the cache. Similarly, when a transaction T does a lookup of a key k, we need to add T to $readers_k$, even if k has not yet been added to the cache.

Our modified `memcached` blocks reads and writes if they conflict. Thus, a transaction T reading key k will be blocked if $(writer_k \neq \text{None} \wedge writer_k \neq T)$, and a transaction T writing key k will be blocked if $(writer_k \neq \text{None} \wedge writer_k \neq T \wedge readers_k - \{T\} \neq \{\})$.

When Django is ready to commit a read/write transaction T, it sends a commit message to the database. If T commits at the database, CacheGenie removes T from the readers and writers of all keys in `memcached`, and allows any transaction blocked on one of those keys to resume executing, by adding itself to the appropriate readers and writers (latches must be used to prevent concurrent modifications of a given readers or writers list). If T aborts, CacheGenie has to remove T from the readers list of all keys it read, and remove all keys it wrote from the cache (so that subsequent reads will go to the database). For read-only transactions, Django does not need to contact the database for either aborts or commits, because read-set tracking is done entirely in the cache. Django can issue single-statement (autocommit) transactions for read queries that are not satisfied by the cache; if Django later needs to perform a write as a part of one of these transactions, these initial reads will not affect correctness of the serialization protocol.

Note that deadlocks can occur in the above protocol; because keys can be distributed across several `memcached` servers, we propose using timeout-based deadlock detection (as in most distributed databases). When a deadlock occurs, Django will need to abort one of the transactions on the database server, using the above abort scheme. Note that we cannot abort a transaction that is in the process of committing at the database server. One concern with the performance of this approach is the overhead of tracking each read/write operation by `memcached`. However, we think it would be insignificant as compared to (a) network latency, and (b) actual operation as size of *tid* is much smaller than the actual key/value size. Database overhead is also minimal as it only needs to maintain a list of `memcached` servers it contacted for each transaction.

Although we haven't implemented this full-consistency approach, we do provide a simple mechanism for the programmer to opt for a strict consistency on a case-by-case basis, if she so desires. If the programmer is aware that some cached object needs strict consistency in certain scenarios, she can opt out of automatic fetching from cache for that particular cached object. Then the programmer manually uses the cached object when she requires weak consistency and does not use it in case she requires strict consistency. The query in the latter case goes directly to the database and fetches the fresh results.

4 Implementation

We implemented a prototype of CacheGenie by extending the popular Django web application framework for Python. One advantage of using Django is that there are several open-source web applications implemented on top of Django, which we can use to test CacheGenie's performance and usability. In particular, we use Pinax, which is a suite of reusable Django applications geared towards online social networking.

Applications in Django interact with the database via models. A Django model is a description of the data in the database, represented as Python code. A programmer defines her data schema in the form of models and Django creates corresponding tables in the database. Further, Django automatically provides a high-level Python API to retrieve objects from the database using functions on model objects, such as `filter` and `limit`. Django also provides ways to define many-to-one, many-to-many and one-to-one relationships between database tables, using model attributes referring to other models.

We implemented the cache classes described in §3.1 as special classes in Django. A programmer can cache frequently accessed queries that fit our abstractions by defining instances (called cached objects) of the appropriate cache class. The cache class performs three functions—(i) it uses the models and fields in the cached object to derive the underlying query template to get that object from the database, (ii) it generates and installs the associated triggers on the database, and (iii) it intercepts regular Django queries to return cached results transparently, if present, and otherwise populates the cache with the desired data from the database. Our prototype supports invalidation, update-in-place, and expiry intervals for cache consistency.

We use unmodified `memcached` for caching. The default least-recently used (LRU) eviction policy works well for a web application workload. However, keys get bumped to the front of LRU when touched by the triggers, even though they are not really being "used" by the application. One can modify `memcached` to support a modified LRU policy where certain actions can be specified not to affect LRU.

For the database, we use unmodified Postgres, which is supported by Django natively. We exploit Postgres triggers (written in Python) to manage the cache on writes to the database. Note that even though we picked Django, `memcached`, and Postgres for our prototype implementation, it should be easy to apply our design to other web application frameworks, caches, and databases.

5 Evaluation

The first aspect of our evaluation is ease of use. To evaluate CacheGenie's ease of use, we ported Pinax, a reusable suite of Django applications geared towards online social networking, to use CacheGenie. Our results show that CacheGenie's abstractions require few changes to existing applications (changing about 20 lines of code for Pinax).

The second aspect of our evaluation is performance. For this, we compare three systems: (i) `NoCache`—a system with no caching, where all requests are being served from the database, (ii) `Invalidate`—CacheGenie prototype in which cache consistency is maintained by invalidating cached data when necessary, and (iii) `Update`—CacheGenie prototype in which consistency is maintained by updating cached data in-place. We evaluate performance using the Pinax applications ported to CacheGenie. Although it would

be instructive to compare CacheGenie with other automated cache management approaches, our system is inherently tied to the ORM model, whereas most of the previous systems are not, making a direct comparison difficult.

We ran several experiments to evaluate CacheGenie's performance. The results of these experiments show that using CacheGenie improves request throughput by a factor of 2–2.5 over `NoCache` for a mixed read-write workload. We also see that the `Update` scenario has up to 25% throughput improvement over `Invalidate`. Increasing the percentage of reads in the workload improves caching benefits of CacheGenie: for a read-only workload, CacheGenie improves throughput by a factor of 8.

We also performed microbenchmarks to understand the performance characteristics of CacheGenie. These microbenchmarks show that using `memcached` instead of a database can improve throughput by a factor of 10 to 150 for simple queries. Further, a database trigger can induce overhead ranging from 3% to 400%.

The rest of this section describes these experiments in more detail.

5.1 Experimental Setup

Pinax is an open-source platform for rapidly developing websites and is built on top of Django. We modified Pinax to use CacheGenie, and focused on three applications from the social networking component of Pinax—profiles, friends and bookmarks. We deal with only four particular actions by the user: (i) LookupBM: lookup a list of her own bookmarks, (ii) LookupFBM: lookup a list of bookmarks created by her friends, (iii) CreateBM: add a new bookmark, and (iv) AcceptFR: accept a friend invitation from another user.

We created cached objects for the frequent and/or expensive queries involved in loading the pages corresponding to these actions. For instance, we added cached objects for getting a list of a user's bookmarks, a count of saved instances of a unique bookmark, a list of bookmarks of a user's friends and so on. Once these cached objects have been defined, the application code automatically obtains the corresponding cached data.

For performance evaluation, as illustrated in Figure 1c, our experimental setup comprises of three main components: (i) the application layer (web clients, web server, and application), (ii) the cache layer, and (iii) the database layer. Since the aim of our evaluation is to measure the performance of the cache and database (the data backend), we have combined the web clients, web server and application server into a single entity called the 'application layer', which simulates a realistic social-network style workload and generates the corresponding queries to the data backends.

Our experimental workload consists of users logging into the site, performing the four actions (**Page Load**) described above according to some distribution and logging out. The default ratio of these actions in our workload is $\langle$LookupBM : LookupFBM : CreateBM : AcceptFR$\rangle$ = $\langle 50 : 30 : 10 : 10 \rangle$. We can also look at it as the ratio of read pages (LookupBM + LookupFBM) to write pages (CreateBM + AcceptFR). The default ratio is then 80% reads and 20% writes. We believe that this ratio is a good approximation of the workload of a social networking type application where users read content most of the time and only sometimes create content (Benevenuto et al [2] found that browsing activities comprised of about 92% of all requests). Note that 20%

writes does not mean 20% of queries are write queries, but only reflects the percentage of write *pages*. In practice, as in real web application workloads, a write page also has several read queries in addition to write queries.

The set of actions from a user's login until her logout is referred to as one **Session**. We refer to each action/page as a **Page Load**. Further, any request for data issued by the client to either the database or the cache is referred to as a **Query**. For most of the experiments each client runs through 100 sessions. The distribution of users across sessions is according to a *zipf* distribution with the zipf parameter set to 2.0 (as in Benevenuto et al [2]). Each session in turn comprises of 10 page loads, in the ratio specified above. Each page load consists of a variable number of queries, depending on the application's logic, which averages about 80.

For the final measurements, we only replay the queries generated during actual workload runs. As such, we need only one client machine to saturate our database. The client machine is an Intel Core i7 950 with 12 GB of RAM running Ubuntu 9.10. We use Python 2.6, Django 1.2 and Pinax development version 0.9a1. The client machine, database machine, and the `memcached` machine communicate via Gigabit ethernet. The database machine is an Intel Xeon CPU 3.06 GHz with 2 GB of RAM, running Debian Squeeze with Postgres 8.3. The database is initialized with 1 million users and their profiles, 1000 unique bookmarks with a random number of bookmark instances (between 1 and 20) per user. Further, each user has 1–50 friends to begin with, and 1–100 pending friendship invitations. The total database size is about 10 GB. The tables are indexed and clustered appropriately. The caching layer consists of `memcached` 1.4.5 running on a Intel Pentium 2.80 GHz with 1 GB of RAM. The size of the cache depended on the experiment, but for most experiments it was 512 MB. Note that this is only an upper limit on the amount of memory it *can* use, if needed.

5.2 Programmer Effort

As described in §3.1, the programmer needs to add a cached object definition for each query pattern instance she wants CacheGenie to cache for her. To port the Pinax applications described earlier in this section, we added 14 cached objects. Adding each cached object is just a call to the function `cacheable` with the correct parameters.

Once the cached object has been created, caching is automatically enabled for corresponding queries. In the absence of CacheGenie, the programmer has to manually write code to get and put data in the cache wherever a query is being made. In our sample application, we counted 22 explicit locations in the application code where such modifications are necessary. However, there are many more instances where the query is being made implicitly, such as from the Django framework. In such cases, the programmer will have to modify the internals of Django in order to cache these queries. We believe developers may find this undesirable.

To manage cache invalidations and updates, CacheGenie automatically generates triggers corresponding to the defined cached objects. For the 14 cached objects, CacheGenie generates 48 triggers, comprising of about 1720 lines of Python code. Without CacheGenie, the programmer will have to manually invalidate any cached data that might be affected by any write query to the database, and without an automatic cache management scheme, the programmer will have to write about the same number of lines

of code as our generated triggers, i.e. 1720 lines of code. Large applications may have many more cached objects and hence many more lines of code for cache management.

5.3 Microbenchmarks

We used microbenchmarks to quantify performance characteristics of the cache, database and database triggers. We ran a variety of experiments on a small database (fits in RAM) and a similarly sized `memcached` instance to measure the time cost of database vs. cache lookup queries, and the time cost of running a trigger.

For database vs. cache lookups, we found that simple B+Tree lookup on the database takes 10–25× longer on the database, suggesting there is significant benefit in caching.

We also looked at the time to run a database trigger as a part of an INSERT operation, relative to the cost to perform a plain INSERT. We found that a plain INSERT takes about 6.3 ms, while an INSERT with a no-op trigger takes about 6.5 ms. Opening a remote `memcached` connection, however, doubles the INSERT latency to about 11.9 ms. Each `memcached` operation done from within the trigger takes an additional 0.2 ms, which is the same amount of time taken by a normal client to perform a `memcached` operation. Hence, the main overhead in triggers comes from opening remote connections; if we could reuse the connection for subsequent triggers, it would make the write operations much faster. Exploring this is a part of future work.

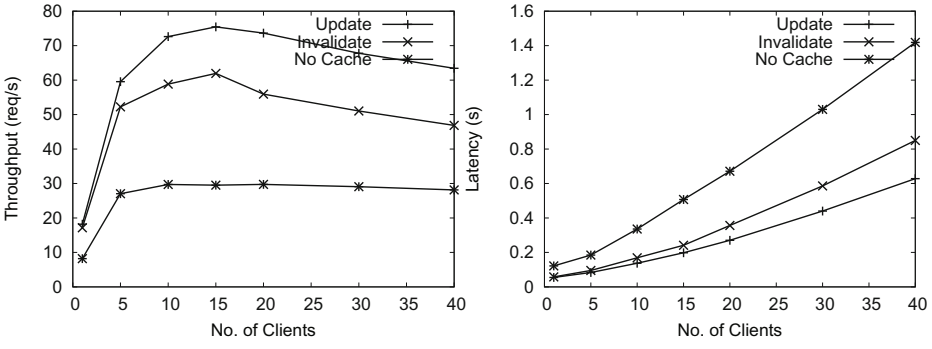
5.4 Social Networking Workload

In this section, we describe our performance experiments with the Pinax applications, present these results, and discuss our conclusions from these experiments. Each experiment has the following parameters: number of clients, number of sessions for each client, workload ratio, zipf parameter, and cache size. The default values for these parameters are 15 clients, 100 sessions, 20% write pages, 2.0, and 512 MB respectively. In each experiment, we measure throughput and latency, and compute averages for the time intervals during which all the clients were simultaneously running. We also warm up the system by running 40 parallel clients for 100 sessions before the start of each experiment.

Experiment 1: Throughput and Latency Measurement. In this experiment, we compare the three caching strategies—`NoCache`, `Invalidate` and `Update`—in terms of the maximum load they can support. The results in Figure 2 show the page load throughput and page load latency as the number of parallel clients increases.

From Figure 2a we can see that CacheGenie systems—`Invalidate` and `Update` provide a 2–2.5× throughput improvement over the `NoCache` system. This improvement is due to a significant number of queries being satisfied from the cache, thereby reducing the load on the database. Note that the advantage we get from our system is much less than the throughput benefit that `memcached` has over a database in our microbenchmarks. This is because we do not cache all types of queries; the uncached queries access the database and make it the bottleneck.

In all three systems, the database is the bottleneck and limits the overall throughput of the system. In the `NoCache` case, the CPU of the database machine is saturated,



(a) Page load throughput. (b) Page load latency.

Fig. 2. Experiment 1—Performance against varying clients

while in the two cached cases, disk I/O is the bottleneck. This is because queries hitting the database in `NoCache` are repeated, and hence a significant amount of time in the database is spent computing query results for in-memory data. For the cached cases, the bulk of the queries are either writes or not repeated (since the system caches most of the repeated queries). Hence, the database becomes bottlenecked by disk. This also explains why the throughput in cached cases drops after a certain point.

Note that the throughput is greater in the `Update` case than in the `Invalidate` case. On one hand, updating leads to slower writes, because triggers have to do more computation. On the other hand, updates lead to faster reads, because there are more cache hits. Figure 2a illustrates that the overhead of recomputing from the database after invalidation is more than the overhead of updating the relevant cached entries.

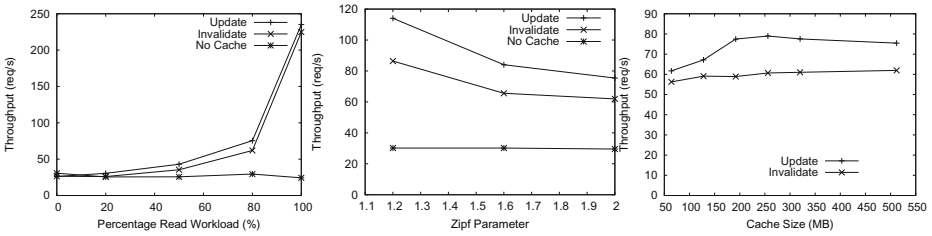
From Figure 2b we see that `Update` has the least latency of all the three scenarios, followed by `Invalidate` and `NoCache`. Also, the latency in all three cases rises more steeply as we increase the number of clients beyond 15, corroborating the fact that throughput drops slightly after this point. Table 2 shows the average latency for various types of page loads for the three systems in this experiment. The increased latency of CreateBM and AcceptFR is due to the overhead of updating the cache during writes to the database. In effect, the update operations become slower in order to ensure that subsequent reads are faster.

Table 2. Average latency by page type in Experiment 1 (with 15 clients)

Page Type	`Update`	`Inval.`	`NoCache`
Login	0.29 s	0.34 s	0.11 s
Logout	0.10 s	0.11 s	0.05 s
LookupBM	0.05 s	0.05 s	0.22 s
LookupFBM	0.06 s	0.16 s	1.25 s
CreateBM	0.55 s	0.53 s	0.09 s
AcceptFR	1.03 s	1.24 s	1.01 s

For all of the following experiments, in order to achieve the maximum throughput for all systems, we run 15 parallel clients, unless otherwise specified.

Experiment 2: Effect of Varying Workload. In this experiment, we vary the ratio of read pages to write pages in the workload, and measure how it affects the performance of the three caching strategies. The default ratio, as mentioned before, is 80% read pages and 20% write pages. The results of these experiments are shown in Figure 3a.



(a) Varying workload. (b) Varying user distribution. (c) Varying cache size.

Fig. 3. Throughput results for Experiments 2, 3, and 4

From the figure, we see that for a workload with 0% reads, caching does not provide any benefit. In fact, it makes the performance slightly worse. This is because database writes are slower in the cached system due to the overhead of triggers. As the percentage of reads in the workload increases, however, the performance of cached cases improves. In the extreme case of 100% reads, the cached case throughput is about 8× the throughput of `NoCache`. Also note that the workload variation does not significantly affect `NoCache` since it is already CPU bound because of reads, which hit the database buffer pool. However, workload variation affects the cached cases, since they are disk-bound, and the disk is accessed less as the number of writes goes down.

The gap in throughput between `Update` and `Invalidate` increases as the number of reads increases from 0% because as the fraction of reads increases, the advantage of better cache hit ratio overcomes the disadvantage of slower triggers in `Update`. The gap reduces when we have 100% reads because nothing in the cache is being invalidated or updated, and so both cases are equivalent. From this experiment, we conclude that caching shows much more benefit in a read-heavy workload than a write heavy one.

Experiment 3: Effect of Varying User Distribution. The formula for zipf distribution is $p(x) = \frac{x^{-a}}{\zeta(a)}$ where ζ is the Reimann zeta function and a is the zipfian parameter. In our experiments, $p(x)$ is the probability that a user has x number of sessions, i.e. logs in x number of times. $p(x)$ is high for low values of x and low for high values of x. In other words, most users log in infrequently, and a few users log in frequently. Also, a low value of the zipfian parameter a means the workload is more skewed whereas a high value means that users login with a more uniform distribution.

The value of zipf parameter affects both performance of the database and the cache. In the cache, if there are certain users who login frequently, then the data accessed by them remains fresh in the cache and the infrequent users' data gets evicted. This means that, over a period of time, frequent users will find most of their data in cache, and hence the number of cache hits goes up, improving the system's performance. It also means we need a cache big enough to hold only the frequent user's data, which is much smaller than the total number of users in the system. It matters for the database performance as well, but only within short intervals, since the buffer pool of database gets churned much faster than the cache. Thus, database performance improves when users log in repeatedly in a short time span.

In this experiment we vary the parameter a of the zipf distribution and see how it affects the performance of the three systems. Figure 3b shows the results from this experiment. From the graph, we can see the cached cases have a 1.5× higher throughput with $a = 1.2$ as compared to $a = 2.0$. The `NoCache` case, however, fails to show any improvement with changing values of a. The performance benefit in the cached cases comes from the database, which is disk-bound. With a lower zipf value, the database is hit with more repeated queries and reduces disk activity, thereby improving the performance for those cases. However, the `NoCache` case is already CPU-bound, and since Postgres does not have a query result cache, it still has to compute the results for the repeated queries from cached pages in the buffer pool.

Experiment 4: Effect of Varying Cache Size. In all our experiments so far, the cache was big enough (512 MB) and there were no evictions. This means the only misses in the cache would be for keys which have never been put in the cache in the first place, or which have been invalidated. In a realistic system we may not have a cache that is big enough to hold everything that can ever be cached. The purpose of this experiment is to analyze the performance effect of evictions due to a smaller cache (in the cached cases).

The results from this experiment are shown in Figure 3c. The graph shows that the throughput of `Update` plateaus at about 192 MB, whereas for `Invalidate` it does so at about 128 MB. This is because `Update` never invalidates the data in cache, and thus requires more space. We can see that even with only 64 MB of cache space, `Update` and `Invalidate` have at least twice the throughput than `NoCache`. In practice the cache size needed depends on frequency of users and the distribution of workload.

Another important result of our experiments is that using spare memory as a cache is much more efficient than using that memory in the database. To validate this, we ran `memcached` on same machine as the database, so that for the cached cases, the database has less memory. The throughput of `Update` in this experiment was 64 requests/s (down from 75), and the throughput of `Invalidate` was 48 requests/s (down from 62). This performance is still better than `NoCache` whose throughput was 30 requests/s.

Experiment 5: Measuring Trigger Overhead. In §5.3, we measured the overhead of triggers in a simple database with simple insert statements. That established the lower-bound that triggers impose on an unloaded database. In this experiment, we measure the impact of triggers on the throughput of CacheGenie for the social networking workload.

An *ideal* system will be one in which the cache is updated for "free", incurring no overhead to propagate writes from the database to the cache. In such a system, the cache always has fresh data, making reads fast, and writes are done at the maximum rate the database can support. The throughput of such a system will be the upper bound on what we could possibly achieve in CacheGenie. To estimate the performance of such a system, we re-ran query traces from our social networking workload with the default parameters and with the triggers removed from the tables in the database. In this way, we make the same queries to the database and `memcached` as in experiment 1, but without any cache consistency overhead. Since the triggers are off, the cache is not updated (or invalidated). Even though this means that the read queries to cache will return incorrect data, it gives us a correct estimate of the performance of the *ideal* system.

From this experiment, we measured the throughput for the ideal `Update` system trace to be 104 requests/s (up from 75) and for the ideal `Invalidate` system to be 80 requests/s (up from 62). In other words, adding triggers brings down the throughput by 22–28% for a fully loaded database. We believe that this overhead is reasonable. In the future, we plan to explore various trigger optimizations to minimize this overhead. One is to combine various triggers on a single table into one single trigger to avoid trigger launching overhead. Second, triggers written in C could be potentially faster than Python triggers. Third, we can reuse connections to `memcached` between various triggers.

6 Conclusion

We presented CacheGenie, a system that provides high-level caching abstractions for web applications. CacheGenie's semantic caching abstractions allow it to maintain cache consistency for an unmodified SQL database and cache system, by using auto-generated SQL triggers to update caches. Our prototype of CacheGenie for Django works with Postgres and `memcached`, and improves throughput of Pinax applications by a factor of 2 to 2.5. Modifying Pinax to take advantage of CacheGenie required changing only 20 lines of code, instead of requiring programmers having to manually maintain cache consistency at every database update.

Acknowledgments. This work was partially supported by Quanta.

References

[1] Amiri, K., Park, S., Tewari, R.: DBProxy: A dynamic data cache for Web applications. In: Proc. of the ICDE, Bangalore, India (2003)
[2] Benevenuto, F., Rodrigues, T., Cha, M., Almeida, V.: Characterizing user behavior in online social networks. In: Proc. of the IMC, Chicago, IL (2009)
[3] Bornhövd, C., Altinel, M., Mohan, C., Pirahesh, H., Reinwald, B.: Adaptive database caching with DBCache. IEEE Data Eng. Bull. 27(2), 11–18 (2004)
[4] Bouchenak, S., Cox, A., Dropsho, S., Mittal, S., Zwaenepoel, W.: Caching Dynamic Web Content: Designing and Analysing an Aspect-Oriented Solution. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 1–21. Springer, Heidelberg (2006)
[5] Challenger, J., Iyengar, A., Dantzig, P.: A scalable system for consistently caching dynamic Web data. In: Proc. of the INFOCOM, New York, NY (1999)
[6] Degenaro, L., Iyengar, A., Lipkind, I., Rouvellou, I.: A Middleware System Which Intelligently Caches Query Results. In: Coulson, G., Sventek, J. (eds.) Middleware 2000. LNCS, vol. 1795, pp. 24–44. Springer, Heidelberg (2000)
[7] Django Software Foundation. Django, `http://www.djangoproject.com/`
[8] Fitzpatrick, B., et al.: Memcached, `http://memcached.org`
[9] Garrod, C., Manjhi, A., Ailamaki, A., Maggs, B., Mowry, T., Olston, C., Tomasic, A.: Scalable query result caching for Web applications. In: Proc. of the VLDB, Auckland, New Zealand (August 2008)
[10] Gupta, A., Mumick, I.S.: Maintenance of materialized views: Problems, techniques, and applications. IEEE Data Engineering Bulletin 18, 3–18 (1995)

[11] Hansson, D.H., et al.: Ruby on Rails, http://rubyonrails.org/
[12] Hibernate Developers. Hibernate, http://hibernate.org/
[13] Johnson, R.: Facebook outage, http://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919
[14] Jónsson, B.T.: Application-oriented buffering and caching techniques. PhD thesis, University of Maryland, College Park, MD (1999)
[15] Kallen, N.: Cache Money, http://github.com/nkallen/cache-money
[16] Labrinidis, A., Roussopoulos, N.: WebView materialization. In: Proc. of the SIGMOD, Dallas, TX (2000)
[17] Labrinidis, A., Luo, Q., Xu, J., Xue, W.: Caching and materialization for Web databases. Foundations and Trends in Databases 2(3), 169–266 (2010)
[18] Larson, P.-Å., Goldstein, J., Zhou, J.: MTCache: Transparent mid-tier database caching in SQL server. In: Proc. of the ICDE, Boston, MA (2004)
[19] Liu, C., Cao, P.: Maintaining strong cache consistency in the World-Wide Web. In: Proc. of the ICDCS, Baltimore, MD (1997)
[20] Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B.G., Naughton, J.F.: Middle-tier database caching for e-Business. In: Proc. of the SIGMOD, Madison, WI (2002)
[21] Nelson, M.N., Welch, B.B., Ousterhout, J.K.: Caching in the Sprite network file system. ACM Trans. Comput. Syst. 6(1), 134–154 (1988)
[22] Perez-Sorrosal, F., Patiño-Martinez, M., Jimenez-Peris, R., Kemme, B.: Consistent and Scalable Cache Replication for Multi-tier J2EE Applications. In: Cerqueira, R., Pasquale, F. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 328–347. Springer, Heidelberg (2007)
[23] Perez-Sorrosal, F., Jimenez-Peris, R., Patiño-Martinez, M., Kemme, B.: Elastic SI-Cache: Consistent and scalable caching in multi-tier architectures. VLDB Journal (2011)
[24] Ports, D.R.K., Clements, A.T., Zhang, I., Madden, S., Liskov, B.: Transactional consistency and automatic management in an application data cache. In: Proc. of the OSDI, Vancouver, BC, Canada (2010)
[25] Sivasubramanian, S., Pierre, G., van Steen, M., Alonso, G.: GlobeCBC: Content-blind result caching for dynamic Web applications. Technical Report IR-CS-022, Vrije Universiteit, Amsterdam, Netherlands (June 2006), http://www.globule.org/publi/GCBRCDWA_ircs022.html
[26] Tauber, J., et al.: Pinax, http://pinaxproject.com/
[27] The TimesTen Team. Mid-tier caching: The TimesTen approach. In: Proc. of the SIGMOD, Madison, WI (2002)

Deploy, Adjust and Readjust: Supporting Dynamic Reconfiguration of Policy Enforcement

Gabriela Gheorghe[1], Bruno Crispo[1], Roberto Carbone[2], Lieven Desmet[3], and Wouter Joosen[3]

[1] DISI, Università degli Studi di Trento, Italy
First.Last@disi.unitn.it
[2] Security and Trust Unit, FBK, Trento, Italy
carbone@fbk.eu
[3] IBBT-Distrinet, K.U. Leuven, 3001 Leuven, Belgium
First.Last@cs.kuleuven.be

Abstract. For large distributed applications, security and performance are two requirements often difficult to satisfy together. Addressing them separately leads more often to fast systems with security holes, rather than secure systems with poor performance. For instance, caching data needed for security decisions can lead to security violations when the data changes faster than the cache can refresh it. Retrieving such fresh data without caching it impacts performance. In this paper, we analyze a subproblem: how to dynamically configure a distributed authorization system when both security and performance requirements change. We examine data caching, retrieval and correlation, and propose a runtime management tool that, with external input, finds and enacts the customizations that satisfy both security and performance needs. Preliminary results show it takes around two seconds to find customization solutions in a setting with over one thousand authorization components.

Keywords: configuration, policy, enforcement, middleware, cache.

1 Introduction

Systems like Facebook, with hundreds of millions of users that add 100 million new photos every day, have data centres of at least thousands of servers [17,20]. Similarly, eBay uses over 10000 Java application servers [23]. Such large applications need an infrastructure that can adapt to its usage constraints, since performance is a must for business. Performance usually refers to the number of user requests (e.g., clicked links on a web page) serviced per time unit. For Facebook, for instance, it has been stated that if website latency is reduced by 600ms, the user click-rate improves by more than 5%[24]. To achieve performance, Twitter, Google and eBay are using caching (page caching for user pages, local caching for scripts), replication, and data partitioning.

Application providers want security and privacy requirements to be satisfied. Application policies focus on user authentication and authorization, and data privacy. The problem of satisfying all these needs, i.e., *policy enforcement*, becomes complex in a distributed system with multiple constraints and multiple

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 350–369, 2011.

users. When user data, profiles and action histories are spread across several domains, it is difficult to make this data available safely and consistently across the system. Maintaining this data is essential for correct security decisions, but is complicated by different domains introducing specific data access patterns.

Caching can impair policy enforcement. Commonly, user certificates, authorization decisions or user sessions are cached so that they are quickly retrieved. But when this data changes faster than it is cached, the decision of authenticating or authorizing may no longer be correct, in which case revenue or human lives can be at stake. For instance, on eBay when a buyer wants to bid on an item that is advertised by a seller, there can be a policy on the seller's side that would reject receiving bids when the buyer has a history of bad payment. If at one moment in time the customer's history is clean, by the time the bid request gets to the seller's side it will be accepted since it relies on a cached value of the payment history that is clean; this decision might not be true, in fact, if there is one bad payment reported before the cache is refreshed.

Attribute retrieval can also affect policy enforcement. Attributes refer to all data needed for authorization decisions (e.g. user profile, history, system state, user state). Retrieval can be done directly or using a mediator, in which case it runs the risk of staleness. Also, if attributes are semantically related, their retrieval should be the same way (we call it *correlation*). Attributes are not always retrieved directly, for two reasons: (1) it is costly to retrieve them every time, and (2) the data is private. For instance, eBay has multiple security domains (i.e., groups of providers with similar policies); to authorize a buyer operation in domain A, the eBay system needs to check the buyer history of bad payments with any seller for the current month, and buyer history is maintained in domain B. If domain B gives the freshest buyer history to domain A just for computing the bad payment events, all non-payment data would be disclosed.

When security enforcement and performance collide, such vulnerabilities occur in different systems tuned for performance. The Common Vulnerabilities and Exposures database [21] reveals numerous entries related to cache and configuration management. For instance, OpenSSL suffers from race conditions when caching is enabled or when access rights for cache modification are wrong (CVE-2010-4180, CVE-2010-3864, CVE-2008-7270). Similar problems of access restrictions to cached content are with IBM's DB2 or IBM's FileNet Content Manager (CVE-2010-3475, CVE-2009-1953), as well as ISC's BIND (CVE-2010-0290, CVE-2010-0218). Exploiting such issues is reported to lead to buffer overflows, downgrade to unwanted ciphers, cache poisoning, data leakages, or even bypassing authentication or authorization. Protocol implementations like OpenSSL and BIND, or servers like IBM's and RSA Access Manager, are examples of technologies that need to scale fast, but cannot scale *fast and safely*.

We argue that the techniques to improve a distributed application's performance must be tailored to the application's security needs. In our view, caching, retrieval and correlation of enforcement attributes require a management layer that intersects both the application and its deployment environment. To our knowledge, these aspects have not yet been approached in security enforcement.

To bridge this gap, we suggest a method and framework for adjusting at runtime the security subsystem *configurations*, i.e., the ways to connect components and tune connection parameters. This adjustment requires to find ways to connect security components so that their connections satisfy a set of constraints, specified by domain experts or administrators (e.g., eBay security architects or security domain administrators) and included as annotations in the XACML security policy. The system configuration that allows for both security and performance needs is *dynamic*. Constraints can change because tolerance to performance overheads or inaccurate enforcement decisions can vary; security domains can vary in dimension; network topology can change. Since varying runtime constraints imply re-evaluation rounds, we want to automate the reconfiguration of the authorization subsystem. Thus, our contributions are:

1. We show the need to consider security and performance restrictions together, rather than separately. We focus on attribute retrieval, caching, and correlation. To our knowledge, we are the first to look at caching from the perspective of the impact of stale attributes over the authorization verdict in the policy enforcement process.
2. We present a method to dynamically compute the correct configurations of policy enforcement services, by transforming system constraints into a logic formula solved with a constraint solver.
3. We suggest the first middleware tool to perform adaptive system reconfiguration on security constraints. Having split computation in two phases, preliminary results show that the heavier computation phase takes 2.5 seconds to compute a solution for over 1000 authorization components.

The paper is structured as follows. After an overview of the standard enforcement model (Section 2.1) and an illustrative example (Section 2.2), Section 3 overviews properties of enforcement attributes. Section 4 describes our approach and proposed architecture, while Sections 5 and 6 describe our prototype in more details. Related work is presented in Section 7 and Section 8 concludes.

2 Background and Example

To be able to evaluate the configuration of the authorization system and how it can be adjusted, this section first describes the reference model in policy enforcement, and then presents an example that we find illustrative for our case.

2.1 The Reference Enforcement Model

The eXtensible Access Control Markup Language (XACML)[1] is the de facto reference in authorization languages and architectures. XACML has been widely adopted in industry, examples ranging from IBM's Tivoli Security Manager [13], to Oracle, JBoss and Axiomatics [3]. XACML has been an OASIS standard

[1] http://www.oasis-open.org/committees/xacml/

since 2003, and apart from the XML-based language specification, it proposes an authorization processing model based on the terminology of IETF's RFC3198[2]. Fig. 1 shows the three main components (greyed out) that we consider:

Policy Decision Point (PDP) is the entity that makes the authorization decision, based on the evaluation of the applicable policies;

Policy Enforcement Point (PEP) performs two functions: makes decision requests, and enforces (implements) the authorization decision of the PDP;

Policy Information Point (PIP) is the entity that acts like a source of attribute values to the PDP that makes the decisions.

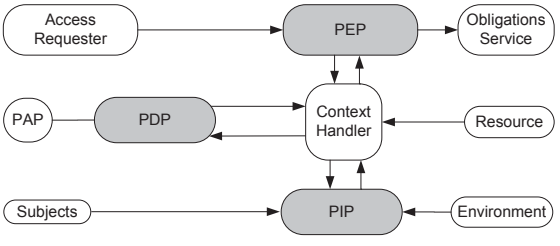


Fig. 1. The XACML reference architecture

There are two variations of the XACML model, shown in Fig.3: the *attribute push model*, where the PEP collects the attributes offered by the PIPs and sends them to the PDP, and the *attribute pull model*, when the PDP collects all relevant attributes in order to make a decision. The pull model has a minimal load on the client and PEP and is designed for cases when the authorization process is entirely performed on the PDP; the push model, conversely, allows for more flexibility to the client and the PEP needs to ensure all required data reaches the PDP. In practice (e.g., PERMIS [5]), a *hybrid model* is used, whereby some attributes are pushed and some are pulled. To our knowledge, no distinction has been made as to when to pull and when to push an attribute.

There have been efforts to augment the XACML reference architecture: Higgins[3] proposes a new authorization architecture for managing online identity. A similar identity management variation was proposed by Djordjevic et al. [7], suggesting a common governance layer that incorporates the PEP but also manages interactions with an identity broker. Overall, such meta-models still keep the original XACML architecture. Thus, using the XACML reference as our model suffices to support a claim of general applicability.

2.2 Illustrative Example

A *security subsystem* is the ensemble of components that perform authentication or authorization enforcement – a set of PIPs, PEPs, and PDPs. We follow

[2] http://tools.ietf.org/html/rfc3198
[3] http://www.eclipse.org/higgins/

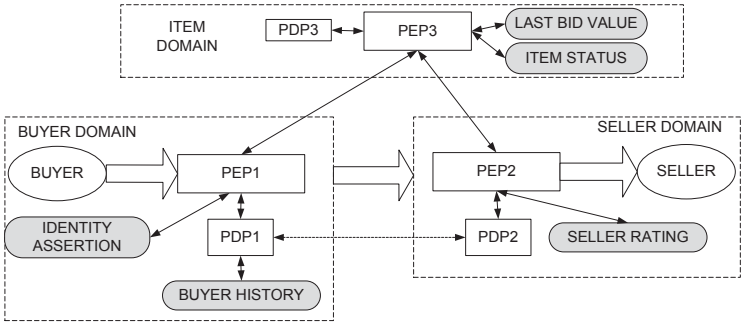


Fig. 2. Cross-domain authorization for a Buyer in BuyerDomain to bid in SellerDomain, with item information from ItemDomain. The greyed boxes are the bid authorization attributes: identity assertion, history, seller rating, item status and bid value.

some architectural points from eBay [16], whereby the security subsystem is divided into security domains (i.e., groups of components under similar security policies). User credentials, profile and state are spread across different domains; access patterns to such data can vary, and domains need to authenticate to other domains when user data is retrieved. We target policies whose evaluation require scattered security data, e.g. the policy (**P1**)"*an authenticated buyer is allowed to make a bid on available items from a seller only if the buyer's history shows no delayed payment for the last month*". The buyer history for the current month, seller's available items on sale, identity tokens, and sellers' ratings – this data is kept at various locations where it may change at runtime, with an impact over allowing or disallowing further user actions (e.g., bids). If buyer attributes are not updated correctly or if fresh values are not retrieved in time, buyers will be blocked from certain sellers, or buyers will buy items that they should not normally access. Fig. 2 shows a multi-domain system where in order for a buyer in domain Buyer Domain to be allowed to bid to a Seller in the Seller Domain, its request must pass through the security subsystem of Buyer Domain, gather some data from Item Domain, and pass through the security subsystem of Seller Domain, which in its turn can also require item data from Item Domain.

For us, the problem of attribute management as shown above can be solved by finding a tradeoff between security correctness and performance requirements. This observation is confirmed by Randy Shoup, distinguished architect at eBay, who admits that an efficient caching system aims to "maximise cache hit ratio within storage constraints, requirements for availability, and tolerance for staleness. It turns out that this balance can be surprisingly difficult to strike. Once struck, our experience has shown that it is also quite likely to change over time" [22]. In this paper, we propose a framework to compute and recompute such balance at runtime and adapt the security subsystem accordingly: we do not want to change the entities of the security subsystem, but the connections between them that involve retrieval and caching of security attributes.

3 Attribute Configuration

Attributes are application-level assets, and have value in the enforcement process (that is aware of application semantics). In eBay, buyer history, item status and seller rating are attributes; in Shibboleth [14], the LDAP class *eduPerson* defines name, surname, affiliation, principal name, targetId as usual identity attributes.

How to obtain attributes is specified by *configuration (meta)data*. For instance in Shibboleth, identity and service providers publish information about themselves: descriptors of provision, authentication authority and attribute authority, etc. Such metadata influences the enforcement process, and can cover:

- visibility (e.g., is the assertion server reachable? is the last bid value public?),
- location or origin restrictions (e.g., for EU buyers use EU server data),
- access pattern of security subsystem to contact third-party (e.g., if the item data can be encrypted, can be backed up or logged),
- connection parameters (e.g., how often should buyer history data be cached or refreshed? who should retrieve it?).

Our point is that this configuration level, including attribute metadata, complements the enforcement process and can influence it. We examine three aspects: (1) push/pull models for attribute retrieval, (2) caching of attributes, and (3) attributes to be handled in the same way (coined 'correlation').

3.1 Attribute Retrieval

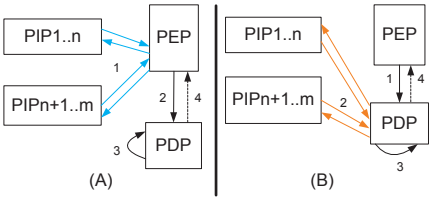
In our eBay example in Fig. 2, *attribute push* from PEP1 to PDP2 means that some attributes – 'identity assertion' , 'buyer history' and item information from Item Domain – are pushed to the Seller Domain and then to PDP2, which will make the final access decision. The 'buyer history', 'last bid value' and 'item status' can also be *pulled* by PDP2 at the moment when it needs such data.

From the point of view of performance, both attribute push and attribute pull can be problematic. In the attribute pull case, an excessive load on PDP2 can make it less responsive for other buyer requests. Performance is linked with security: low PDP2 performance can be made into a Denial of Service attack by saturating PDP2 with requests that require intensive background work. The case of all attributes being retrieved by PEP1 and given to PDP2 is also delicate: for example, if PEP1 knows that PDP2 requires the number of delayed payments of the buyer, it means that PEP1 will do the computation instead of PDP2; this scheme can put too much load on PEP1, that can have a negative effect on PEP1's fast interception of further events.

From the point of view of trust and intrusiveness, the push scheme is problematic. In Fig. 2, PDP2 must trust what PEP1 computed as the highest customer rating this month, and this potentially sensitive data would need to travel between PEP1 and PDP2. This decision bears several risks, of which: PEP1 might compute the wrong rating value; the decision might be delayed; if the rule changes from "any delayed payments", to "under five", then PEP1's logic should be updated. The security subsystem should decide if exporting the computation

of the delayed payment count to PEP1 costs more than the privacy of such local data to Buyer Domain. In either case, PEP1 and the communication channel should be trusted not to tamper with the data, and the policy should be fixed.

To determine when to use the push or the pull scheme, we have analysed a number of possible policy enforcement configurations and several existing scenarios and their solutions – PERMIS [4,5], Shibboleth [14], and Axiomatics [3]. In particular, PERMIS looks at role-based authorization in multiple Grid domains. There are two kinds of regulations on subject credentials in PERMIS: policies on what credentials can be attached to the authorization request leaving from Buyer Domain (policies enforced by PDP1 in Fig. 2) and can reach Seller Domain; and policies on what credentials can be trusted by Seller Domain as the destination of an authorization request from Buyer Domain. Validating a buyer credential in Seller Domain – done by PDP2 – involves a chain of trust among issuers that have different rights to issue attributes for different sets of subjects. On a similar note, Shibboleth [14] enforces attribute release policies whereby once a user has successfully authenticated to a remote website, certain user attributes can be provided on demand via back-channels by requesting Web sites while other attributes cannot. These examples confirm that user or system attributes are sensitive, and should not always be pushed to the PDP.



Push scheme	Pull scheme
stateless attributes	state attributes
environment attributes	behaviour history
not sensitive user data	sensitive user data
request details	obligations
destination details	domain decisions
stateless context	stateful context

Fig. 3. Diagrams A and B show the classic attribute push and pull models. The table to the right shows how different attributes fit the push or pull scheme better.

Based on the security considerations above, we separate the applicability of the push and pull schemes based on types of attributes (see Fig. 3, right). The push scheme is more appropriate for those attributes that can be safely collected by the PEP or can only be retrieved by the PEP rather than the PDP (e.g., message annotations whose semantics is known by the PEP: IP address of a buyer and payment server, description of a token service, location and country of a certificate server). A similar treatment may be for attributes unlikely to change frequently: user identity, group membership, security clearances or roles, authentication tokens, or any constant data about system resources or the environment that the PEP can retrieve easily. Conversely, the PDP directly interacts with PIPs when it needs specific application logic and related state (e.g., customer rating, payment obligations), history of a user (e.g., bidding history of a user), or data that only the PDP is allowed to access (e.g., highest bid).

3.2 Attribute Caching

Caching in policy enforcement can be of three types: of (1) attributes, (2) decision, and (3) policy [25]. The PDP may cache policies that change little in terms of content, but caching attributes with different change rates is difficult. For instance, bid history, while expected to change a lot, can be constant if the user does not bid; conversely, feedback rating, while expected to change slowly, can change fast for a seller who does not ship items to buyers over a period of time. For similar reasons, the PDP or the PEP might maintain a cache of the decisions already made. Unlike attributes, that tend to change values the most frequently, decisions and policies are more static. We hereafter focus on attribute caching, but our framework can apply to decision and policy caches too.

Table 1. Different caching concerns in enforcement

Scheme	**What to cache**	**Where to cache**	**How to invalidate**
Attribute caching	attributes	PEP or PDP	time limit
Decision caching	policy decisions	PDP	explicit
Policy caching	entire policy	PEP	explicit

Since cached attributes are attributes too, the push and pull scheme applies to caches as well as to attribute retrieving. New values of the attributes needed to make a policy decision can be either (1) pushed to the entity that needs them, be it the PEP or the PDP, or (2) pulled by the same entity that needs them. There are two cases that combine both caching and attribute retrieving issues:

1. *push to PEP cache, push to PDP*: a PEP pushes attributes to the PDP, and whenever an attribute is updated, the PEP cache is notified and updated. The scheme relies on an external entity to notify the PEP of attribute changes. Inaccurate notifications can compromise decision correctness.
2. *pull to PEP cache, push to PDP*: a PEP that pushes attributes to the PDP, and periodically queries attributes for fresh values (pull). This case does not use a third-party but puts more load on the PEP. Also, the PEP should have poll times that depend on the change rate of the attributes to be cached.

The cases when the PDP pulls attributes by itself and stores them are similar in that cached values need to be refreshed at a rate decided either by a (trusted) third-party or at the rate at which the PDP can query the data sources. From this last point of view – the polling rate of the PDP – cache management has the notion of cache invalidation policy, whereby the cached values have a validity time; when they become invalid, the manager will retrieve fresh copies. Table 1 shows how to invalidate caches depending on what security aspect is cached.

3.3 Attribute Correlation

In an interview with Gunnar Peterson (IEEE Security & Privacy Journal on Building Security), Gerry Gebel (president of Axiomatics Americas) pointed out that there can be different freshness values for related attributes [10]. The example given is that of a user with multiple roles in an organisation; whenever the user requests access to a system resource, the PDP needs to retrieve the current role of the user; some of the user's roles may have been cached (in the PDP, PEP or PIP) hence there might be different freshness values to be maintained for several role attributes. In our eBay scenario in Fig. 2, it can be that more than one item attribute changes from the moment a buyer reads it, and before the last bid. In order for the security subsystem to allow the buyer to bid, it must gather buyer information, item restrictions, *and* item information. If at this stage, the system can access the freshest bid value, but not the freshest item status flag, then the buyer could still be allowed to bid on a sold item.

We generalise such examples to the idea that correlated attribute need a common refresh rate for all the attributes in the group (e.g., all role attributes from an LDAP server, username and password attributes, source IP and port attributes, role and mutually exclusive role list, etc). Some attribute correlations are application-dependent (e.g., mutually exclusive roles, like bidder and item provider for the same item), others are application-independent (e.g., username and password for single sign-on). Hence, bundling attributes that should be used together is a must when enforcement decisions need to be accurate. With PEPs or PDPs likely to use overlapping attribute sets to enforce a cross-domain policy, synchronization over common attributes is essential for attribute consistency.

Having a control over the attributes used in policy enforcement implies the need for a management layer that we call 'configuration layer'. We continue by describing our solution to the issues above.

4 Approach and Proposed Architecture

We consider the setting of a distributed application on top of which there is a security subsystem in charge of enacting several security policies. This subsystem consists of multiple PEPs, PIPs and PDPs. In the state of the art so far, the connections between these components are fixed once a policy is deployed. In our view, they should change at runtime, since this way the system can *adapt* to varying security or performance constraints. How often these changes are incorporated into the security subsystem depends on which of security or performance is paramount, and on the disruption incurred when actually changing the connections among security components.

We will use the term *wiring* for enabling a configuration on the concrete infrastructure (realizing the connections among PIPs, PEPs and PDPs in order to support the attribute management features explained above). We want to rewire the different authorization components with hints from the application domain and knowledge of the enforced policies. We see such hints supplied with the XACML policy, since it is likely that the policy writer, aware of some

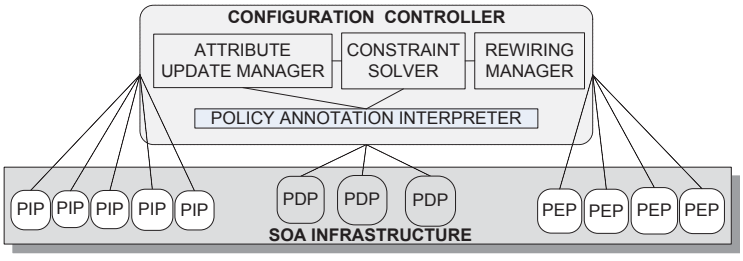


Fig. 4. The architecture of our solution

application domain details, can formulate requirements for the treatment of attributes. We also assume that security components have standard features: a cache storage space (hence also cache size and cache refresh policies), permissions over enforcement attributes, or other properties (domain, location, etc).

The architecture of our solution is presented in Fig. 4. Instead of having a set of hardwired connections between the PEPs, PIPs and PDP, we suggest a configuration layer on top of the SOA infrastructure. This layer enables the dynamic rewiring according to varying runtime conditions of the application. Responsible of this task is a Configuration Controller (CC) consisting of four components: a Policy Annotation Interpreter (PAI), an Attribute Update Manager (AUM), a Constraint Solver (CS) and a Rewiring Manager (RM).

The **Policy Annotation Interpreter (PAI)** extracts the policy annotations relevant for the configuration of the security subsystem.

The **Attribute Update Manager (AUM)** monitors the value changes of security attributes and notifies or directly propagates the new values to PEPs and PDPs or to their respective caches. This component also manages attribute synchronization and consistency. It should be connected to all PIPs needed to enforce a policy. The purpose of the AUM is to propagate attribute changes to the security subsystem. The environment, user properties or security relevant state might change at runtime, so the security subsystem needs to be notified.

The **Constraint Solver (CS)** is the component that finds solutions to satisfy the connection constraints of PEPs, PIPs and PDPs. Such constraints cover attribute retrieval, caching and correlation, and are specified in the security policy. The CS processes the policy, searches the solution space, and selects the configurations of the security subsystem that do not violate the constraints.

The **Rewiring Manager (RM)** enacts the solutions found by the CS. The RM resides at the middleware layer and is dependent on the communication infrastructure (in this case, a SOA infrastructure). How this infrastructure is rewired is not within the scope of this paper, and was addressed before [12].

With this approach in mind, we continue by examining types of runtime constraints, how they can be specified, and how the CS can handle them. There are several assumptions for our running system: first, all PIPs are considered to provide fresh information to the other security components. Then, PEPs can cache PIP data, and PDPs can cache enforcement decisions and policies.

4.1 Annotating XACML Policies

For the eBay authorization policy that requires attributes 'user token' and 'user feedback rating', our approach is qualitative: if there are multiple providers of user tokens, or if the rating attribute has to be fresh, our aim is to ensure such quality considerations are respected. This additional data (where an attribute can be retrieved from, if it can be stale or pushable) belongs in the authorization policy. The reason is that the policy writer usually has the correct idea over attribute usage and invalidation with respect to the correctness of the policy decision; the policy writer knows if the user token does not change a lot so that it can be pushed to the PDP, or if the buyer/seller feedback rating is critical and volatile so should not be cached.

```
<xs:element name="AttrProps" type="att-xacml:AttrPropsType"/>
<xs:complexType name="AttrPropsType">
   <xs:attribute name="AttrId" type="xs:anyURI" use="required">
   <xs:sequence>
     <xs:element ref="att-xacml:Providers" minOccurs="1" maxOccurs="unbounded">
     <xs:element name="AttrNotCached" minOccurs="0" maxOccurs="unbounded">
     <xs:element name="AttrCacheable" minOccurs="0" maxOccurs="unbounded">
     <xs:element name="AttrPushable" minOccurs="0" maxOccurs="unbounded">
     <xs:element ref="att-xacml:CorrelAttr" minOccurs="0" maxOccurs="unbounded">
   </xs:sequence>
</xs:complexType>
```

Fig. 5. Attribute meta-data elements in the `att-xacml` schema

The XACML 3.0 syntax does not natively support attributes about subject, environment or resource attributes. We suggest to enrich the default XACML syntax with annotations specific to the following aspects: (1) what PIP/PEP provides an attribute, (2,3) if an attribute can be cached and where, (4) if an attribute is pushable or pullable and where, (5) the correlation among different attributes. Until a XACML profile bridges this gap, a solution to this problem is to specify attribute metadata as an element in an enhanced schema we call `att-xacml` and part of which is shown in Fig. 5. The `AttrPropsType` type consists of the properties of attributes that we are interested in: `attribute-id`, and a series of elements to indicate on what PEPs or PDPs they can be pushed, cached or not cached, what PIPs provide them, and correlated attributes. We assume that these features, attached to each attribute in a policy, are interpreted by an extension of the XACML engine. The PAI processes the semantics of these attributes for the CS.

4.2 Satisfying Configuration Constraints

The restrictions on attributes that were specified in a XACML-friendly syntax in Section 4.1 will reach the Constraint Solver (CS). The CS has to match these constraints against its runtime view over the authorization subsystem, which we

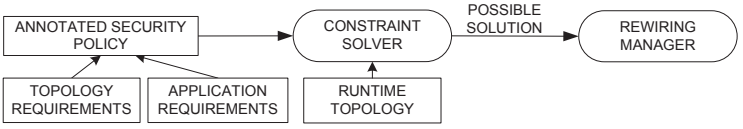


Fig. 6. Model for configuration and rewiring in our approach

call *runtime topology*. The result is a number of solutions that satisfy all constraints; we call these solutions *configuration solutions*, or *configurations*. If the CS finds no solution, then the constraints are not satisfiable and the security subsystem remains unchanged. The CS is shown in Fig. 6. Satisfying configuration constraints is complicated by the existence of different and interdependent constraint types. It is not in the scope of this paper to analyse the impact of each of these constraints over the resulting configurations. Yet, for now, we acknowledge that reconfiguration depends on how often and what triggers the CS to re-evaluate its previously found configurations and issue new ones. Here we see two possibilities: the re-evaluation can be per policy, or per policy subpart. In the first case, the trigger of the reconfiguration is a new policy version, and the changes required for the new configuration might be scarce and far apart in the topology. In the second case, the trigger of a reconfiguration is a change in the runtime topology that relates to an attribute provider or consumer referred to by a security policy; here, the reconfiguration changes are likely to be closer together in the topology. It is hard to say which one happens more often, since this is application dependent. A tradeoff must be made between a system that reconfigures itself either too frequently, or never.

4.3 (Re)Wiring

Rewiring of PEPs, PDPs, and PIPs refers to changing the connections among these components. These connections are established at the SOA *middleware* level, that enables component intercommunication. In our previous work [11], we envisioned the Enterprise Service Bus (ESB) as the manager of the security subsystem. The main reason is that, by design, the ESB controls the deployed security components and their connections; when runtime conditions change (e.g., components appear or disappear, security policies are updated), the ESB can modify message routing on the fly (e.g., validate credentials before they reach an authorization server), trigger attribute queries (e.g., check the feedback rating attribute every minute and not every hour) or change attribute propagation to security domains (e.g., what entities are entitled to receive bid product updates). In particular, the dynamic authorization middleware in [12] applies the dynamic dispatch system of an ESB to enable run-time rewiring of authorization components across services. Each authorization component (PEP, PDP, PIP and PAP) is enriched with an authorization composition contract, and a single administration point allows the wiring and rewiring of authorization components. Using lifecycle and dependency management, the architecture guarantees that

authorization wirings are consistent (i.e., all required and provided contract interfaces match), and that they remain consistent as the rewiring happens. Since rewiring was addressed before, here we focus on the constraint solving problem.

5 Configuration Prototype

From the components shown in Fig. 6, we concentrated on the constraint solver component. While aspects about the RM and the AUM have been approached in previous work, runtime constraint solving is the most challenging aspect of our solution. We assume the CS receives runtime data about the components of the security subsystem, and attribute constraints from the deployed policies (Section 3), and produces configuration solutions that satisfy the constraints. We have prototyped the constraint solving in Java with the SAT4J[4] SAT solver.

5.1 Constraint Solving with a SAT Solver

We used a SAT solver because we wanted to obtain configuration solutions for a wide range and number of constraints (Section 5.2 explains why it is difficult to do this without a SAT solver). Given a propositional formula in conjunctive normal form (CNF) describing both the problem and its constraints, a SAT solver is designed to find an assignment of the propositional variables that satisfy the formula. The solver can also address partial maximum (PMAX) SAT problems: given a set of clauses S (a 'maximization set'), find assignments that maximise the number of satisfied clauses in S. This approach naturally maps to the configuration problem that administrators are faced with. For a security-aware application, satisfying all security constraints is paramount, while performance constraints should be maximized if possible. The reciprocal is treated similarly.

Fig. 7 shows two ways to use a SAT solver in our problem: in the first case, an encoded input is fed to the solver to find a valid configuration. If some part of the input changes at runtime, the whole input will be encoded again, and the solver will recompute all solutions from scratch. This method requires a massive effort to re-encode the entire problem, even if only a small part of it has changed. We employed an alternative incremental approach. The SAT solver receives an initial set of clauses, and if a subset of these clauses change at runtime, its internal mechanisms recompute only the subpart of the problem that has changed. A maximisation set can be provided in both cases. Next, we describe the encoding, offline and runtime input in our approach.

Offline Input. The administrator sets up the general settings. The set $PIPs$ of all possible PIPs, the set $PEPs$ of PEPs, and the set $PDPs$ of PDPs, say which attributes can be provided by each component. Specifically, for each $pip \in PIPs$, $pep \in PEPs$, and $pdp \in PDPs$, the following data is provided:

- `provide`(pip, a) iff PIP pip can provide attribute a;
- `provide`(pep, a) iff PEP pep can provide attribute a;

[4] http://www.sat4j.org/

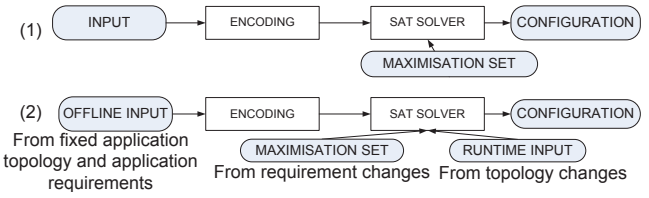


Fig. 7. Two SAT solver methods to solve constraints (inputs and outputs shown darker)

- $\texttt{needs}(pdp, a)$ iff PDP pdp needs attribute a;
- $\texttt{pull}(a, pdp)$ iff PDP pdp needs attribute a directly from some PIP;
- $\texttt{correlation}(pdp)$ iff PDP pdp either pulls or pushes all attributes;
- $\texttt{not\_cacheable}(a, pdp)$ iff PDP pdp needs freshness for attribute a;
- $\texttt{cached}(a, pep)$ iff PEP pep can tolerate a stale attribute a.

These predicates become propositional variables for the solver, along with:

- $\texttt{arch}(pdp, pip, a)$ means that a is pulled directly by pdp from pip.
- $\texttt{arch}(pdp, pep, a)$ means that a is pushed from pep to pdp.
- $\texttt{arch}(pep, pip, a)$ means that a is pushed from pip to pep.

The solver assigns the truth values to the $\texttt{arch}()$ variables; these give the active connections of each component, and each set of assignments is a configuration.

We also use the notation $PIPs_a \subset PIPs$ (and $PEPs_a \subset PEPs$) as the set of PIPs (PEPs) such that $\texttt{provide}(pip, a)$ ($\texttt{provide}(pep, a)$, resp.). Similarly, $PDPs_a \subset PIPs$ is the set of PDPs such that $\texttt{needs}(pdp, a)$ is true. In the attribute retrieval model, we consider the following formulae:

$$\texttt{pull}(a, pdp) \equiv \bigvee \{\texttt{arch}(pdp, pip, a) | pip \in PIPs_a\}$$

$$\texttt{push}(a, pdp) \equiv \bigvee \{\texttt{arch}(pdp, pep, a) | pep \in PEPs_a\}$$

saying that an attribute a is pulled by (pushed to) the PDP pdp if and only if there is an arch between pdp and a PIP pip (PEP pep, resp.). Thus, if the pdp asks for a, and $\texttt{pull}(a, pdp)$ is true, then a must be retrieved directly from a pip.

We use the notation $\oplus\{v_1, v_2, \dots\}$ meaning that exactly one of $v_1, v_2, \dots$ is true, and $\ominus\{v_1, v_2, \dots\}$ meaning that at most one of $v_1, v_2, \dots$ is true. Encoding the possible paths and constraints means a conjunction of the following formulae:

- For each a required by pdp, exactly one connection must exist between pdp and either a PIP providing a or a PEP providing a. This is expressed by the following formula, for each $\texttt{needs}(pdp, a)$ in the Offline Input:

$$\texttt{needs}(pdp, a) \supset \oplus\big\{\texttt{arch}(pdp, x, a) | x \in PIPs_a \cup PEPs_a\big\} \tag{1}$$

  Moreover, in case there is an arch between pdp and pep, providing a, then there must be also a connection between pep and a pip providing a (formula (2)). Otherwise (formula (3)) no arch between pep and pip providing a is necessary. For each $\texttt{needs}(pdp, a)$ and $pep \in PEPs_a$:

$$\texttt{arch}(pdp, pep, a) \supset \oplus\{\texttt{arch}(pep, pip, a) | pip \in PIPs_a\} \quad (2)$$

$$\neg\,\texttt{arch}(pdp, pep, a) \supset \bigwedge\{\neg\,\texttt{arch}(pep, pip, a) | pip \in PIPs_a\} \quad (3)$$

– If a *pip* does not provide a then no connection providing a can exist between *pip* and other components. For each $\texttt{provide}(pip, a)$ in the Offline Input:

$$\neg\,\texttt{provide}(pip, a) \supset \bigwedge\{\neg\,\texttt{arch}(pdp, pip, a) | pdp \in PDPs_a\}$$

$$\neg\,\texttt{provide}(pip, a) \supset \bigwedge\{\neg\,\texttt{arch}(pep, pip, a) | pep \in PEPs_a\}$$

Similarly for the PEPs: for each $\texttt{provide}(pep, a)$ in the Offline Input:

$$\neg\,\texttt{provide}(pep, a) \supset \bigwedge\{\neg\,\texttt{arch}(pdp, pep, a) | pdp \in PDPs_a\}$$

$$\neg\,\texttt{provide}(pep, a) \supset \bigwedge\{\neg\,\texttt{arch}(pep, pip, a) | pip \in PIPs_a\}$$

– To show the advantage of this approach, we consider another constraint: we suppose that each component can provide (e.g. for performance reasons) an attribute only to a finite number of components (in this case, one). To model that, for each $\texttt{provide}(pip, a)$ in the Offline Input, we consider the following:

$$\texttt{provide}(pip, a) \supset \ominus\{(\texttt{arch}(pdp, pip, a) | pdp \in PDPs_a\} \quad (4)$$

$$\texttt{provide}(pip, a) \supset \ominus\{\texttt{arch}(pep, pip, a) | pep \in PEPs_a\} \quad (5)$$

Formula (4) (same for (5)) says that if *pip* provides a then there must be at most one connection between *pip* and the PDPs (PEPs, resp.) requesting a. It is similar for the PEPs, so for each $\texttt{provide}(pep, a)$ in the Offline Input:

$$\texttt{provide}(pep, a) \supset \ominus\{\texttt{arch}(pdp, pep, a) | pdp \in PDPs_a\}$$

$$\texttt{provide}(pep, a) \supset \ominus\{\texttt{arch}(pep, pip, a) | pip \in PIPs_a\}$$

– Without data showing that PDPs cache attributes, we considered only PEPs to do that. This choice does not limit generality. We had to encode the constraint: if *pdp* asks for a, and $\texttt{not\_cacheable}(a, pdp)$ is true, then a cannot be retrieved from a *pep* that cached a. So for each $\texttt{not\_cacheable}(a, pdp)$ in the Offline Input, we have the conjunction:

$$\texttt{not\_cacheable}(a, pdp) \supset \bigwedge\{\neg(\texttt{cached}(a, pep) \wedge \texttt{arch}(pdp, pep, a)) | pep \in PEPs_a\}$$

– The attribute correlation constraint can be described as follows: if *pdp* has correlated attributes, then all its attributes must be either pulled or pushed. For each $\texttt{needs}(pdp, a)$ in the Offline Input, this can be modeled as follows:

$$\texttt{correlation}(pdp) \supset (\texttt{pull}(pdp) \vee \texttt{push}(pdp))$$

where: $\texttt{pull}(pdp) \equiv \bigwedge\{\texttt{pull}(a, pdp)\}$, and $\texttt{push}(pdp) \equiv \bigwedge\{\texttt{push}(a, pdp)\}$

Notice that by defining PMAX-SAT problems it is also possible to impose constraints like: if possible, preference should be given to attribute provisioning via indirect paths, i.e., from PIP to PEP to PDP, over direct paths, i.e., from PIP to PDP. This can be obtained by maximising the number of truth values for the variables corresponding to the arches from PEPs to PDPs.

In satisfiability problems, coding the at-most-one operator $\ominus$ is often problematic. If there are $n > 1$ variables, an improved SAT encoding of $\ominus\{v_1, \ldots, v_n\}$ is the logarithmic bitwise encoding in [9]. This operator's CNF encoding requires $\lceil \log_2(n-1) \rceil$ auxiliary variables and $n\lceil \log_2(n-1) \rceil$ clauses. With this encoding, the CNF formula corresponding to (1) has a complexity (in terms of number of clauses) of $O(|PDPs| \cdot \mathrm{N_{a\_pdp}} \cdot (\mathrm{N_{pips\_a}} + \mathrm{N_{peps\_a}}) \cdot \log_2(\mathrm{N_{pips\_a}} + \mathrm{N_{peps\_a}})$, where $\mathrm{N_{a\_pdp}}$ is the average number of attributes requested by each PDP, and $\mathrm{N_{pips\_a}}$ ($\mathrm{N_{peps\_a}}$) is the average number of PIPs (PEPs, resp.) providing a specific attribute. These constraints generate a large number of clauses. This is why the encoding is computed offline and fed to the solver "on stand-by mode".

Runtime Input. The offline input abstracts a global view of the scenario. The runtime input specifies a current scenario, indicating the security components currently being used, their current set of attributes provided/needed, and any other current constraints. The runtime input is an instantiation of the offline input. For instance, if the Offline Input file contains `provide(pip1, a1)` and `provide(pip2, a2)`, but in the current scenario `pip1` is not available, then the Runtime Input will contain the following assignment: `provide(pip1, a1) = false`.

5.2 Constraint Solving without a SAT Solver

For independent constraints over components, an algorithm to choose configuration solutions may be faster than the SAT solver. Such an algorithm can perform a graph traversal to select the first PIP or PEP that satisfies the requirements of a PDP and some boolean constraints. Graph traversal works best in scenarios where existing connections have no impact on choosing further connections; yet it may become problematic when the graph is dynamic and local connections change global state, that in turn changes other local connections. This can happen, for example, when bandwidth restrictions limit the number of possible connections of a component; or when a user cannot re-rate a seller until the next time they win the auction. Designing such an algorithm is not easy, since it is tightly bound to the constraints: whenever constraints appear or disappear, the algorithm needs to be rewritten. With a SAT solver, the set of constraints can vary without influencing the process of producing a solution, but performance might suffer. Still, it is worth to design such a generic algorithm as future work.

6 Evaluation and Discussion

The performance of our prototype depends on several aspects: the size and semantics of the offline input file, the size and semantics of the runtime input, and

the internal performance of the SAT solver. Since each SAT solver uses a different solving technique, the performance of our prototype depends on the choice of solver. Apart from this dependency, we considered that the most relevant aspect to test is the encoding of the offline input, that need be recomputed several times along the lifetime of an application. This recomputing can be triggered by components that appear or disappear, or by changes in what components provide or require. The runtime input, on the other hand, is only a (moderately small) subset of the maximal offline one, and its impact over the solution finding time depends on the SAT technique used by each individual solver.

Therefore, we wanted to measure the offline CNF generation and the constraint solving time for a number of configuration topologies, while keeping the runtime input minimal. To obtain different offline inputs, there were several parameters to vary: the number of authorization components, the number of attributes and constraints on attributes, the distribution of attributes per authorization component. In choosing which of these parameters to vary and which to keep constant, we considered that the number of attributes is fixed in each scenario, since the administrator should know beforehand the attributes required for all security decisions in a particular application. Hence, we generated over 100 offline input files to describe topologies with varying numbers of PIPs, PDPs, and PEPs, with a fixed number of attributes provided/needed by each. In order to assign attributes to each component, we implemented an algorithm for unbiased random selection as described by Knuth [18, Sec.3.4.2].

To test our Java encoding against the SAT4J solver with the aims described above, we configured the JVM to run with 500 to 1000MB of memory, and we used JRE1.5.0. We ran each of the input files against a minimal runtime configuration file of one constraint, with the following parameter changes: 25, 50, 75 PDPs; 50, 100, 250, 500, 750 PIPs; 25, 50, 75, 100, 150, 200, 250 PEPs. The number of attributes was constant to 100, with 3 attributes per component in all cases. We measured the time (in milliseconds as per the Java call $System.currentTimeMillis()$) for the static CNF generation, as well as the time it takes SAT4J to compute the first solution from the already generated encoding (the solve time). The results for 50 PDPs can be seen in Fig. 8; those for 25 and 75 PDPs are similar. The figures show a linear increase in the offline CNF generation in the number of authorization components (Fig. 8, right) with a similar linear increase in solve time (Fig. 8, left).

These preliminary results are very encouraging in that the time taken to compute the constraints from the offline input is very short for a moderate application size (e.g., about 2.5 seconds for 750 PIPs, 250 PEPs and 50 or 75 PDPs), and also that they grow linearly with the number of components, for a fixed number of attributes. Even though satisfiability is generally known to be an NP-complete problem (and on a general case, it can be expected to obtain an exponential growth in problem complexity and hence performance), the linear relation that we have obtained is justified in that we are using a generic tool to solve a particular instance of a satisfiability problem.

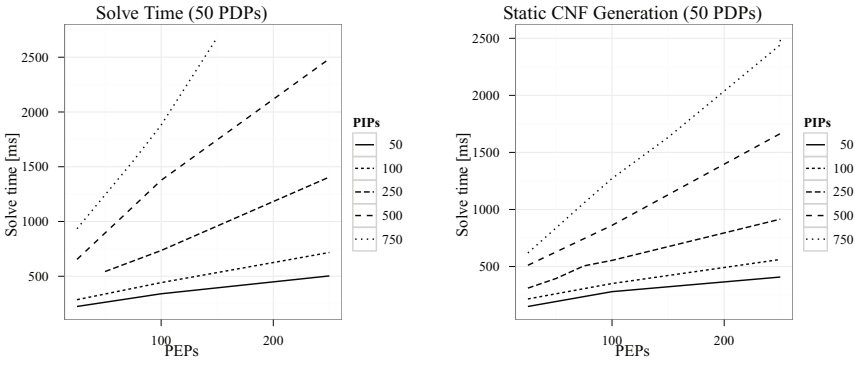


Fig. 8. The time to solve our constraint problem with 50PDPs, and varying numbers of PEPs and PIPs (left) and the time to generate the offline CNF clauses (right)

7 Related Work

Some policy configuration features come with commercial products: Tivoli Access Manager [13], Axiomatics Policy Server [3], Layer 7 Policy Manager [19], Shibboleth [14] and Globus Toolkit 4 [1]. Tivoli provides central management and enforcement of data security policies across multiple PEPs, with partial policy replication and decision caching. Axiomatics offers central administration of policies, and can manage multiple PDP instances. Shibboleth focuses on federated identity management, allows for session caching and attribute backup, but does not consider attribute freshness. Globus considers resource and policy replication. All these products address different policy management aspects differently, and never consider together cache consistency, synchronisation and security guarantees. The users of these products cannot change such features from a single console, and hence cannot understand how one impacts another.

Two previous works are particularly relevant to our approach. First, Colombo et al. [6] observe that attribute updates may invalidate PDP's policy decisions. The work only focuses on XACML's expressive limitations, does not consider system reconfigurations and is not supported by any implementation. Second, the work of Goovaerts et al. [12] focuses on matching of provisions in the authorization infrastructure. The aim is to make the system that enforces a policy seamless and scalable when those components that provide security attributes change or disappear. They do not discuss how attribute provisioning impacts the correctness of the enforcement process.

Several other works are related to ours. Policy management is also tackled in Ponder [8] but aspects such as caching consistency and synchronisation are not considered. PERMIS [5] proposes an enforcement model for the Grid that uses centralised context data stores and PDP hierarchies with visibility restrictions over attributes provided across domains. Like us, Ioannidis [15] separates policy enforcement from its management, suggests a multi-layer architecture to enforce

locally some part of a global policy, but does not discuss consistency of policy data at different locations and propagation of updates. The thesis of Wei [25] is the first to look at PDP latency in large distributed applications for role based access control. He does authorization recycling by caching previous decisions, but does not consider cache staleness. Atluri and Gal [2] offer a formal framework for access control based on changeable data, but the main difference is that their work is on changing the authorization process rather than configuring security components to manage such data appropriately.

8 Conclusions and Future Work

Our work is the first to consider the impact of properties about authorization components and their connections, over the correctness of the enforcement process. We believe that performance tuning uncovers security flaws in distributed applications and we concentrate on security attribute management. Our management solution can help administrators in two ways: it can generate system configurations where a set of security constraints need always be satisfied (along with maximising performance constraints), or can check an existing configuration of the system against a given set of (security or performance) constraints. The configuration solutions can be recomputed at runtime and preliminary results show an overhead of a few seconds for a system with one thousand components. To our knowledge, this is the first tool to perform a fully verified authorization system reconfiguration for a setting whose security constraints would otherwise be impossible to verify manually.

Acknowledgements. This work was partially supported by EU-FP7 NESSoS, the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy, the Research Fund K.U.Leuven, and by the "Automated Security Analysis of Identity and Access Management Systems (SIAM)" project funded by Provincia Autonoma di Trento within "team 2009-Incoming" FP7-COFUND. We also thank Wouter De Borger and Stephan Neuhaus for technical comments.

References

1. Globus Alliance: Globus Toolkit 4 API (November 2010), http://www.globus.org/toolkit/docs/4.2/4.2.1/security/
2. Atluri, V., Gal, A.: An authorization model for temporal and derived data: securing information portals. ACM Trans. Inf. Syst. Secur. 5, 62–94 (2002)
3. Axiomatics: Axiomatics Policy Server 4.0 (November 2010), http://www.axiomatics.com/products/axiomatics-policy-server.html
4. Chadwick, D., Zhao, G., Otenko, S., Laborde, R., Su, L., Nguyen, T.A.: PERMIS: a modular authorization infrastructure. Concurr. Comput.: Pract. Exper. 20, 1341–1357 (2008)
5. Chadwick, D.W., Su, L., Laborde, R.: Coordinating access control in grid services. Concurrency and Computation: Practice and Experience 20(9), 1071–1094 (2008)

6. Colombo, M., Lazouski, A., Martinelli, F., Mori, P.: A Proposal on Enhancing XACML with Continuous Usage Control Features. In: Grids, P2P and Services Computing, pp. 133–146. Springer, US (2010)
7. Djordjevic, I., Dimitrakos, T.: A note on the anatomy of federation. BT Technology Journal 23, 89–106 (2005)
8. Dulay, N., Lupu, E., Sloman, M., Damianou, N.: A policy deployment model for the ponder language. In: 2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings, pp. 529–543 (2001)
9. Frisch, A.M., Peugniez, T.J., Doggett, A.J., Nightingale, P.W.: Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings. J. Autom. Reason. 35, 143–179 (2005)
10. Gebel, G., Peterson, G.: Authentication and TOCTOU (2011), http://analyzingidentity.com/2011/03/18/
11. Gheorghe, G., Neuhaus, S., Crispo, B.: xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement. In: Nishigaki, M., Jøsang, A., Murayama, Y., Marsh, S. (eds.) IFIPTM 2010. IFIP AICT, vol. 321, pp. 63–78. Springer, Heidelberg (2010)
12. Goovaerts, T., Desmet, L., Joosen, W.: Scalable Authorization Middleware for Service Oriented Architectures. In: Erlingsson, Ú., Wieringa, R., Zannone, N. (eds.) ESSoS 2011. LNCS, vol. 6542, pp. 221–233. Springer, Heidelberg (2011)
13. IBM: IBM Tivoli Access Manager (November 2010), http://www-01.ibm.com/software/tivoli/products/access-mgr-e-bus/
14. Internet2MiddlewareInitiative/MACE: Shibboleth 2 (2011), https://wiki.shibboleth.net/confluence/display/SHIB2/Home
15. Ioannidis, S., Bellovin, S.M., Ioannidis, J., Keromytis, A.D., Anagnostakis, K.G., Smith, J.M.: Virtual private services: Coordinated policy enforcement for distributed applications. I. J. Network Security 4(1), 69–80 (2007)
16. Kassaei, F.: eBay Identity Assertion Framework (May 2010), http://www.slideshare.net/farhangkassaei/ebay-identity-assertion-framework-iaf
17. Kerner, S.M.: Inside Facebook's Open Source Infrastructure (July 2010), http://www.developer.com/open/article.php/3894566/
18. Knuth, D.E.: The art of computer programming, vol. 2. Addison-Wesley Longman Publishing Co., Inc., Boston (1997)
19. Layer 7 Technologies: Policy Manager for XML Gateways (November 2010), http://www.layer7tech.com/products/policy-manager-for-xml-gatewaysl
20. Miller, R.: The Facebook Data Center FAQ (September 2010), http://www.datacenterknowledge.com/the-facebook-data-center-faq/
21. Mitre: Common Vulnerabilities and Exposures (2011), http://cve.mitre.org/
22. Shoup, R.: Scalability best practices - Lessons from eBay. InfoQ (May 2008), http://www.infoq.com/articles/ebay-scalability-best-practices
23. Shoup, R.: More Best Practices from Large Scale Websites - Lessons from eBay. Talk at QCon San Francisco (November 2010), http://qconsf.com/sf2010
24. Wei, D., Jiang, C.: Frontend Performance Engineering in Facebook. In: O'Reilly Velocity, Web Performance and Operations Conference (June 2009)
25. Wei, Q.: Towards Improving the Availability and Performance of Enterprise Authorization Systems. Ph.D. thesis, University of British Columbia (2009)

A Middleware Layer for Flexible and Cost-Efficient Multi-tenant Applications

Stefan Walraven, Eddy Truyen, and Wouter Joosen

IBBT-DistriNet, Katholieke Universiteit Leuven, 3001 Leuven, Belgium
{stefan.walraven,eddy.truyen,wouter.joosen}@cs.kuleuven.be

Abstract. Application-level multi-tenancy is an architectural design principle for Software-as-a-Service applications to enable the hosting of multiple customers (or tenants) by a single application instance. Despite the operational cost and maintenance benefits of application-level multi-tenancy, the current middleware component models for multi-tenant application design are inflexible with respect to providing different software variations to different customers.

In this paper we show that this limitation can be solved by a multi-tenancy support layer that combines dependency injection with middleware support for tenant data isolation. Dependency injection enables injecting different software variations on a per tenant basis, while dedicated middleware support facilitates the separation of data and configuration metadata between tenants. We implemented a prototype on top of Google App Engine and we evaluated by means of a case study that the improved flexibility of our approach has little impact on operational costs and upfront application engineering costs.

Keywords: Multi-tenancy, Dependency injection, Software-as-a-Service, Google App Engine.

1 Introduction

Context. An important trend in the landscape of service-oriented software has been the rise of the "Software-as-a-Service" (SaaS) delivery model [31] where software applications are created and sold as highly configurable web services. A well-known SaaS provider delivers for instance a Customer Relationship Management (CRM) application [28] as a configurable service to a variety of customers that each have their specific preferences and required configurations.

SaaS applications differ from traditional application service provisioning (ASP) in the sense that economies of scale play a much more important role. A traditional application service provider typically manages one dedicated application instance per customer. In contrast, SaaS providers typically adopt a *multi-tenant architecture* [7], meaning that a shared application instance hosts multiple customers, which are called tenants. The primary benefit of this approach is that the *operational costs can be significantly reduced*: (i) hardware and software resources can be more cost-efficiently divided and multiplexed across customers, and (ii) the overall maintenance effort is seriously simplified because upgrading the application software can be performed for all tenants at once.

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 370–389, 2011.

Problem. Application-level multi-tenancy comes however also with a number of disadvantages. More specifically, in this paper we focus on two challenges when implementing multi-tenancy at the application level. First *application engineering complexity* is increased. The engineering of multi-tenant application software is more complex than traditional single-tenant applications that are deployed per individual tenant. The primary cause is that the application developer should take measures to ensure isolation between different tenants with respect to the application configuration and data of each tenant [15]. Moreover, a tenant-specific management facility needs to be created such that application configuration management per tenant is separated from the core application management by the SaaS provider.

Secondly, in order to meet the unique requirements of the different tenants, the application must be *highly configurable and customizable*. The current state of practice in SaaS development is that configuration [7,15] is preferred over customization which is considered too complex [30]. Configuration usually supports variance through setting pre-defined parameters for the data model, user interface and business rules of the application. Customization on the other hand involves software variations in the core of the SaaS application in order to address tenant-specific requirements that cannot be solved by means of configuration. Compared with configuration, customization is currently a much more costly approach for SaaS vendors because it introduces an additional layer of application engineering complexity and additional maintenance overhead.

Approach & Contribution. This paper presents a software development and execution platform[1] for building and deploying customizable multi-tenant applications, narrowing down the gap between configuration and customization. More specifically, we present a multi-tenant middleware layer on top of Platform-as-a-Service (PaaS) platforms that (i) supports improved customization flexibility, (ii) preserves the operational cost benefits of the application-level multi-tenancy principle, and (iii) frees the application developer from a lot of initial application engineering costs for multi-tenancy.

We implement our middleware layer on top of Google App Engine (GAE) [13]. We extend the Guice dependency injection framework [14] with support for tenant-specific activation of software variations and use the scalable and high-performance datastore of GAE for storing and isolating tenant-specific application metadata. We evaluate the feasibility of our middleware layer by comparing a standard single-tenant and multi-tenant application with a flexible version that is developed using our middleware layer. This shows that the impact of our middleware layer on operational costs and additional application engineering complexity is minimal.

Structure of the Paper. The remainder of this paper is structured as follows. Section 2 introduces the case study and motivates the need for a middleware that supports true application-level multi-tenancy with improved customization

[1] Other aspects of SaaS applications such as SLA management, metering and billing are out of the scope of this paper.

flexibility. Subsequently, Section 3 presents the architecture of our middleware layer and its implementation on top of Google App Engine. Section 4 presents the evaluation of our middleware architecture in the three dimensions of customization flexibility, operational costs, and initial engineering costs. Section 5 elaborates on related work and Section 6 concludes the paper.

2 Problem Elaboration and Motivation

This section first explores the design space of multi-tenant applications and positions our intended middleware architecture in this space. Subsequently our work is motivated by means of an application case. Finally, the main requirements for our middleware layer are derived from a customization scenario in this application case.

2.1 Multi-tenancy Architectural Strategies

Multi-tenancy aims to maximize resource sharing among customers of a SaaS application and to reduce operational costs. However different architectural strategies can be applied to achieve multi-tenancy. As shown in Fig. 1, multi-tenancy can be realized at the application level, middleware level or virtualized infrastructure level. Each approach makes a different trade-off between (i) minimizing operational costs (including infrastructural resources as well as maintenance cost), (ii) minimizing upfront application (re-)engineering costs, and (iii) maximizing flexibility to meet different customer requirements.

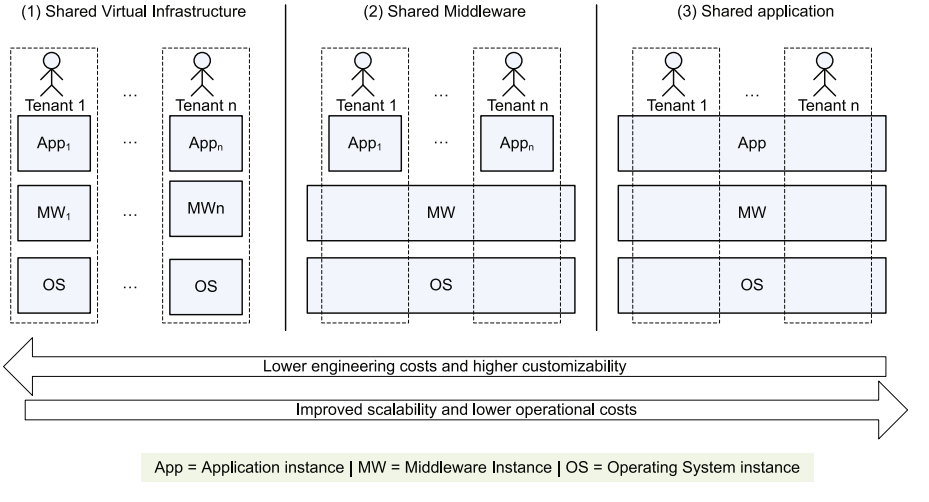


Fig. 1. Different architectural approaches to achieve multi-tenancy

As stated in the introduction, application-level multi-tenancy maximizes the level of resource sharing but is also the least flexible choice with additional engineering overhead. At the other end of the spectrum, virtualization technology can be used to run multiple operating system partitions with dedicated application and middleware instances for each tenant on shared servers. The advantage of this approach is its increased flexibility and low upfront application engineering cost. However, fewer tenants can be hosted on a single server and maintaining separate application instances per tenant also has a much higher cost than with application-level multi-tenancy.

Middleware-level multi-tenancy [5,2] uses a separate middleware platform that is able to host multiple tenants on top of a shared operating system, which may be either placed on a physical or virtualized hardware. In this way, the initial engineering complexity for multi-tenancy is shifted from the application level to a reusable middleware layer that also offers basic support for isolation of tenants. However, the component and deployment model of these middleware architectures still require that a separate application instance is deployed for each tenant which again implies a higher maintenance cost.

Our proposal is to create middleware support for building true multi-tenant applications with the flexibility to adapt to tenant-specific requirements. Because all tenants are served by the same instance of the application, this means that there is need for tenant-specific software variability in the application components. We assume that such multi-tenant application components do not maintain tenant-specific state, but that all tenant-specific state is stored in a (separate) database. To ensure scalability when user load increases, a pool of identical application instances with our middleware layer have to be created. Existing PaaS platforms already take care of this scalability requirement in a transparent way. For example, Google App Engine automatically scales up (and down) by creating extra instances as the load increases. We therefore propose to incept our middleware layer as an extension for PaaS platforms.

2.2 Motivating Example

Consider the example of a SaaS provider for on-line hotel booking (see Fig. 2). The SaaS provider offers a highly configurable web service that travel agencies can use for booking hotels and flights on behalf of their customers. Travel agencies play in this example the role of tenant whereas employees and customers of a travel agency are considered the users that belong to a tenant. Employees are offered a customized user interface and customers of the travel agency can login to check the status of the travel items through a URL with a custom-made domain-name that corresponds with the travel agency. A special 'tenant administrator' role is assigned to someone who is responsible for configuring the SaaS application, setting up the application data and monitoring the overall service. This role can be played by an internal or external client of the SaaS provider or even resellers who are an intermediate business proxy. In the context of this simple example, the tenant administrator belongs to the ICT staff of a travel agency company.

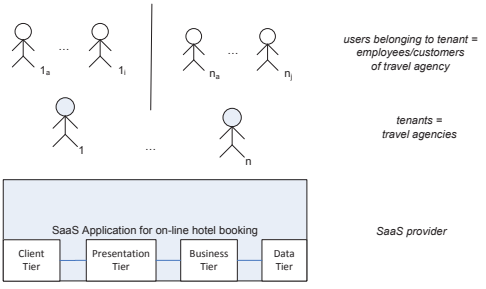


Fig. 2. SaaS application for on-line hotel booking

2.3 Requirements Derived from a Customization Scenario

Suppose that a particular travel agency wants to be able to offer price reductions to their returning customers. As such, the on-line hotel booking application should be extended with an additional service for managing customer profiles and a service for calculating price reductions. We assume that SaaS providers employ a business model where the base application is offered to tenants at no or low cost, but tenants incur an additional price for additional services. Based on this simple scenario, we can derive requirements with respect to core development, service customization and runtime support.

With respect to development, the application development team of the SaaS provider should be offered a simple way to *manage the different tenant-specific variations* as separate units of deployment that can be selectively bound to the core architecture of the application. Moreover, the overall 'multi-tenancy concern' should be well separated from the application layer.

With respect to customization, tenant administrators should be offered a *configuration facility* to select what software variations should be enabled for them (e.g. the price reduction service). In addition, this facility should also allow to specify specific configuration parameters (e.g. business rules for the price reduction service). This configuration data should be stored in the datastore of the SaaS provider in an isolated way under a specific tenant ID.

The runtime support of the middleware layer must provide support for *injecting software variations on a per tenant basis*. When a user (either customer or employee) logs in, the tenant to which the user belongs should be determined. Based on the acquired tenant ID, the multi-tenant middleware should then activate the appropriate software components to process the requests of the user. Another key requirement of the execution platform is that the tenant-specific software variations should be applied in an isolated way without affecting the service behavior that is delivered to other tenants.

3 Middleware Support for Tenant-Specific Customization

This section presents the overall architecture of our middleware layer to support tenant-specific customization of SaaS applications. The component model of our

middleware layer targets multi-tier applications and structures the application into a core architecture with declared variability points for multi-tenant software variations. Building on top of this component model, the middleware layer consists of a support layer for tenant administrators and run-time support for injecting software variations on a per tenant basis.

In this paper we focus on the customization of component-based multi-tier applications, rather than business processes (e.g. BPEL). The latter requires a different approach where software variations are deployed as separate services, and per tenant a separate business process is responsible for the coarse-grained composition of the appropriate services. In the context of component-based applications, dependency injection (DI) [11] is a common composition mechanism. With standard DI however, separate object hierarchies are maintained per tenant in a shared address space which increases heap memory storage and supports only static binding of software variations. Therefore, we prefer a composition mechanism that allows in situ run-time rebinding of variations. This requires an extension to the DI mechanism.

This section is structured as follows. We first propose an extension to the multi-tier component model to make it tenant-aware. Next we describe in depth the architecture of our multi-tenancy support layer. Finally, the prototype implementation of this middleware layer on top of Google App Engine [13] is presented.

3.1 Tenant-Aware Component Model

To cope with the different and varying tenant requirements, we apply a *feature-based approach*. Software variations are then expressed in terms of features. A feature is a distinctive functionality, service, quality or characteristic of a software system or systems in a domain [17]. Ideally these features are modular software units that can be easily composed into the base application. As illustrated in Fig. 3, variation points are specified in the base application, representing the locations where features should be composed. A feature can have several alternative implementations (e.g. `I1` and `I2` in the figure). Based on the tenant-specific configuration, one of the feature implementations is bound to the variation points across the different tiers.

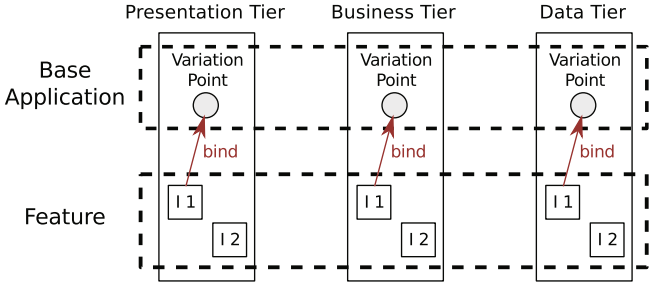


Fig. 3. Illustration of the feature-based approach

Our extension to the component model supports the application developers of the SaaS provider to develop features as software modules. For each feature different implementations can be registered. A feature implementation consists of a set of software components (possibly at different tiers) and specifies how these components are bound to the base application. The concept of features is necessary to enable the SaaS provider to easily ensure the *consistency* of software variations across the different tiers of the SaaS application.

In addition, the developers need to be able to tag the locations in the base application where tenant-specific variation is allowed. To *annotate these variation points*, we introduce a new annotation: `@MultiTenant`. Listing 1 shows the annotation of a field with the price calculation service interface. This variation point initiates customization of the on-line hotel booking application based on the currently applicable tenant-specific configuration, for example price calculation with price reduction. Because a variation point can be bound by different features, the annotation has an optional parameter specifying the feature it belongs to. This enables developers to limit the variation point to a specific feature.

Listing 1. Annotation of a variation point for price calculations

```
...
 @MultiTenant
 private IPriceCalculatorStrategy priceCalculatorStrategy;
...
```

3.2 Architecture of the Multi-tenancy Support Layer

The architecture of our middleware layer supporting flexible multi-tenant applications is presented in Fig. 4. This support layer consists of a *flexible middleware extension framework* to manage features, specify tenant-specific configurations and to dynamically activate the required variations on a per tenant basis via dependency injection. This approach relies on a *multi-tenancy enablement layer*, offering basic multi-tenancy support and facilitating the separation of data and configuration metadata. Our multi-tenancy support layer serves as an extension to middleware platforms, but especially to Platform-as-a-Service (PaaS) solutions. Possibly such a PaaS already offers built-in support for tenant data isolation.

Multi-tenancy Enablement Layer. The base for application-level multi-tenancy is isolation between the different tenants, such as isolation of data, performance and faults. To achieve tenant-specific customization the main requirement is isolation of data, more specifically configuration metadata. With the default single-tenant approach, the configuration of an application is specified in a global configuration file. In a multi-tenant context a global configuration file results in a uniform application for all tenants, preventing tenant-specific customization. Any change to the configuration would affect all tenants. Therefore tenant-specific configurations have to be stored separately and applied within the scope of a tenant, instead of globally.

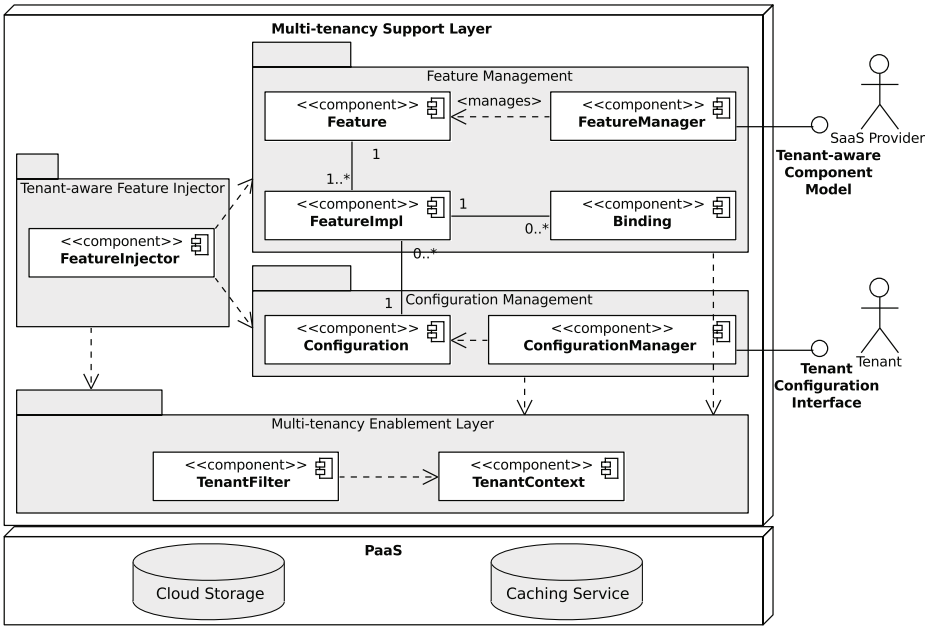


Fig. 4. Overview of the multi-tenancy support layer

To achieve tenant data isolation three main components are required: (i) the *tenant context* containing the information of the tenant linked to the current request (via a unique tenant ID), (ii) *tenant-specific authentication* to identify the tenant, and (iii) *multi-tenant data storage*. Incoming requests are filtered to retrieve the tenant ID (e.g. based on the request URL) and to set the current tenant context. Multi-tenant data storage can be obtained by applying filters that intercept the calls to the storage API and inject the tenant ID from the associated tenant context. In addition, comparable interceptors are necessary for the caching service (distributed in-memory storage). This allows to rapidly retrieve tenant-specific configurations, without large I/O performance overhead.

Flexible Middleware Extension Framework. The flexible middleware extension layer provides the following functionality:

1. a *feature management facility* providing an API to manage the variability of the application and the available feature implementations,
2. a *configuration management facility* to manage the default and tenant-specific configurations,
3. a *feature injector* to dynamically inject the required software variations conforming the tenant-specific configurations.

Feature Management. The `FeatureManager` manages the set of available features and their different implementations. A `Feature` specifies at least the following

information: a unique identifier (e.g. feature name) and description for the feature, and the set of registered implementations for that feature.

A `FeatureImpl` contains the description of the feature implementation, a set of bindings, and a reference to the configuration interface of this implementation. Each `Binding` specifies the mapping from a variation point to a specific software component. This metadata about the features is globally accessible by both the SaaS provider and the tenants, and therefore should not be isolated. The `FeatureManager` offers a development API to enable the SaaS provider to create and register features and feature implementations, while the tenants are able to inspect the different features via the tenant configuration interface.

Configuration Management. Since a feature can have multiple implementations, each tenant can specify its preference for a specific feature implementation via the tenant configuration interface. Such a `Configuration` description defines the mapping from a feature to a specific feature implementation, more specifically from a feature ID to a `FeatureImpl`. The different tenant-specific configurations are then managed by the `ConfigurationManager`. In contrast to the feature descriptions, the tenant-specific configurations are stored on a per tenant basis.

Furthermore, the SaaS provider has to specify a configuration containing for each feature the mapping to a default feature implementation. If a tenant does not specify his tenant-specific configuration, this default configuration will be automatically selected.

Tenant-aware Feature Injection. Based on the features registered in the `FeatureManager` and the default as well as tenant-specific configurations, our multi-tenancy support layer has to activate the appropriate feature implementations when required. To achieve this we apply the *dependency injection (DI)* pattern [11]. Instead of instantiating the feature implementations directly in the application, the flow of control is inverted: the life cycle management of feature implementations is controlled by a dependency injector or provider. This injector binds dependencies in the application to an implementation file. Such a *binding* is traditionally but not necessarily a mapping between a type (generally an interface or abstract class) and an implementation type (a class or component). This concept of a binding between a dependency and an implementation corresponds to our `Binding` between a variation point and a software component, as specified in the `FeatureImpl`s. As a result, in the above `ConfigurationManager` a tenant-specific configuration corresponds to a specific configuration of the DI framework.

For each variation point in the application the tenant-aware `FeatureInjector` decides at runtime which implementation needs to be used, based on the configuration that applies. First, the `FeatureInjector` intercepts the requests to a dependency and consults the `ConfigurationManager`. The latter queries the multi-tenant data storage using the tenant ID to retrieve the tenant-specific configuration. Subsequently, the right binding is obtained from the `Configuration`, specifying the mapping between the variation point and a specific software component. This software component is instantiated and injected in the application

to further handle the request. If the appropriate binding is not available in the tenant-specific configuration, the default configuration is used. In case the feature ID parameter was given, the search to the appropriate binding can be narrowed down to the bindings of a specific feature implementation.

Finally the injected instance is stored in the cache in an isolated way using the tenant ID. For the following requests by this tenant that involve the same variation point, the `FeatureInjector` queries the cache. Using this tenant-aware caching service enables us to support flexible multi-tenant customization of a shared instance without the associated performance overhead.

3.3 Implementation

We implemented a prototype of our multi-tenancy support layer on top of Google App Engine (GAE) [13] (SDK 1.5.0), using the Java programming language and the Guice dependency injection framework [14] (v3.0). Google App Engine is a PaaS plaform to build and host traditional web applications developed with Java Servlets and Java Server Pages (JSP). GAE has built-in support for tenant data isolation via the Namespaces API. A separate namespace is assigned to each tenant. We only had to implement a `TenantFilter` to map incoming requests to a specific namespace and to configure that all requests have to go through this filter. For caching we use the Memcache service.

We chose Guice as DI framework because it is type-safe and compatible with GAE. However, it does not support the execution of tenant-specific injections: all dependencies are set globally. Any modification would affect all tenants. This is a general problem with dependency injection because it does not support activation scopes.

To solve this issue, we added an extra level of indirection. Instead of injecting features, we inject a `Provider` for that feature. This way the servlets have a dependency to a provider of a feature instead of to the feature itself. This generic `FeatureProvider` decides based on the tenant-specific configuration which feature implementation should be selected. However, the customizations that can be performed this way are limited to switching between implementations of an interface or abstract class.

4 Evaluation

The evaluation of our approach consists of several measurements of the operational and reengineering costs for our multi-tenancy support layer. In particular we want to measure the overhead introduced by the multi-tenancy support layer. We compare the results of our multi-tenancy support layer with a multi-instance, single-tenant approach and the default multi-tenant solution without flexibility.

We first describe the general methodology we applied. Next, a general cost model for the operational and reengineering costs of SaaS applications is specified. Finally we present the measurements we performed and compare the results with our cost model.

4.1 Methodology

In this evaluation we measure and compare the operational and engineering costs between a default and flexible single-tenant version, a default multi-tenant version (without flexibility), and a multi-tenant version using our multi-tenancy support layer. For these measurements we use the hotel booking application described in the case study. The source code of these four versions including our multi-tenancy support layer, is available on http://distrinet.cs.kuleuven.be/projects/CUSTOMSS.

To determine the operational costs the diferent versions of the application are deployed on top of Google App Engine (SDK 1.5.0), using the high replication datastore (default option). In the case of the single-tenant application, we deploy a separate application for each tenant, while both multi-tenant versions only need one application each. Each tenant is represented by 200 users who each execute a booking scenario. This booking scenario consists of 10 requests to the application: first several requests to search for hotels with free rooms in a given period, then creating a tentative booking in one hotel and finally the confirmation of the booking. The different users of one tenant execute the booking scenario sequentially, while the tenants run concurrently. Notice that it is not our goal to create a representative load for this application, but to compare the operational costs of the different versions under the same load. We retrieve the information about the execution cost via the GAE Administration Console. It provides a dashboard displaying the resource usage by the application. The focus of this comparison is on the relative differences between the execution costs, since the absolute numbers depend on the current (global) load on the GAE platform.

The reengineering costs are compared based on the quantity of source code used to develop the case study application for the different versions. We make a distinction between Java code, JSP pages (for the user interface), and configuration files (XML). The number of source lines of code are determined using David A. Wheeler's 'SLOCCount' application.

4.2 Cost Model

The goal of the cost model is to define the metrics for our measurements, and to represent our hypothesis about the operational and reengineering costs associated with single-tenant and multi-tenant applications. In addition, it enables us to analyze the impact of customization flexibility on these costs.

Operational Costs. The operational cost can be subdivided in (i) the application's execution cost (resource usage), (ii) the costs to maintain the application such as performing upgrades, and (iii) the administration cost, i.e. the cost to provision a new customer (tenant) with an application.

Execution Cost. We use CPU time, memory and storage usage as the main execution cost drivers. Another important resource is network bandwidth. However, the introduction of multi-tenancy has no effect on the required bandwidth.

Let t be the number of tenants, u the number of active users per tenant, and $Cpu(t, u)$, $Mem(t, u)$ and $Sto(t, u)$ the total usage of respectively CPU, memory and storage. Then, in the case of a single-tenant application (ST),

$$\begin{aligned} Cpu_{ST}(t,u) &= t * f_{CpuST}(u) \\ Mem_{ST}(t,u) &= t * (M_0 + f_{MemST}(u)) \\ Sto_{ST}(t,u) &= t * (S_0 + f_{StoST}(u)) \end{aligned} \tag{1}$$

where $f_{CpuST}(u)$, $f_{MemST}(u)$ and $f_{StoST}(u)$ are functions of u, representing the usage of CPU, memory and storage by one single-tenant application instance. M_0 and S_0 are constants for the memory and storage usage by an idle instance.

In the multi-tenant case (MT) we introduce an extra parameter i, i.e. the number of identical multi-tenant instances managed by a load balancer (see SaaS maturity level 4 in [7]). Then,

$$\begin{aligned} Cpu_{MT}(t,u,i) &= t * (f_{CpuST}(u) + f_{CpuMT}(u)) \\ Mem_{MT}(t,u,i) &= i * M_0 + t * f_{MemST}(u) + f_{MemMT}(t) \\ Sto_{MT}(t,u,i) &= S_0 + t * f_{StoST}(u) + f_{StoMT}(t) \end{aligned} \tag{2}$$

where $f_{CpuMT}(u)$ is a function of u, representing the additional CPU necessary for tenant-specific authentication and isolation of the incoming requests. $f_{MemMT}(t)$ and $f_{StoMT}(t)$ are functions of t for the additional memory and storage required to store (global) data about the tenants, for instance the tenant's name and address.

Since the number of multi-tenant instances is limited compared to the number of tenants and the additional amount of memory and storage for multi-tenancy support is relatively small compared to the shared amount of memory and storage (M_0 and S_0), this results in:

$$\begin{aligned} i &\ll t \\ f_{MemMT}(t) &\ll (t-i) * M_0 \\ f_{StoMT}(t) &\ll t * S_0 \end{aligned} \tag{3}$$

Thus from Equations (1), (2) and (3), we can compare the execution costs of the single-tenant and multi-tenant versions:

$$\begin{aligned} Cpu_{ST}(t,u) &< Cpu_{MT}(t,u,i) \\ Mem_{ST}(t,u) &> Mem_{MT}(t,u,i) \\ Sto_{ST}(t,u) &> Sto_{MT}(t,u,i) \end{aligned} \tag{4}$$

As a result a multi-tenant application consumes less storage and memory than a single-tenant application, but requires more CPU. However, the latter is limited to authenticating the tenant and ensuring isolation.

Maintenance Cost. The maintenance cost largely consists of the cost to develop and deploy upgrades to the application. Let f be the upgrade frequency, i the number of instances to upgrade, and $Upg(f,i)$ the total upgrade cost, then:

$$\begin{aligned} Upg_{ST}(f,t) &= f_{DevST}(f) + t * f_{DepST}(f) \\ Upg_{MT}(f,i) &= f_{DevST}(f) + i * f_{DepST}(f) \end{aligned} \tag{5}$$

where $f_{DevST}(f)$ and $f_{DepST}(f)$ are functions of f, representing the development and deployment cost of one single-tenant application instance. The number of single-tenant instances equals the number of tenants t. Often there is only one multi-tenant application instance that is automatically cloned to spread the load over multiple identical instances, resulting in i being equal to 1. Besides the application, the multi-tenancy support should also be upgraded, but since this is part of the middleware it should not be taken into account here.

Administration Cost. For the SaaS provider the administration cost consists of two constant costs: (i) creating and configuring a new application instance (A_0), and (ii) provisioning a new tenant with an application (T_0), for instance by registering the tenant ID in the application and providing a URL to access the application. Let t be the number of tenants, then:

$$\begin{aligned} Adm_{ST}(t) &= t * (A_0 + T_0) \\ Adm_{MT}(t) &= A_0 + t * T_0 \end{aligned} \tag{6}$$

Reengineering Costs. When migrating an application to the cloud, reengineering is required to make use of the available cloud services, for example storage. In addition, making an application multi-tenant results in an additional reengineering cost. The latter is the difference in reengineering costs between a single-tenant and a multi-tenant application, and is dependent on the middleware platform that is used. For example, when an API for multi-tenancy is provided, this reengineering cost stays limited. Without this support, additional development is required to provide tenant-specific authentication and to ensure isolation between the different tenants.

Impact of Flexibility. Our multi-tenancy support layer provides multi-tenant SaaS applications with the flexibility to adapt to the different and varying requirements of the tenants. However, this also has an effect on the operational and reengineering costs.

Operational Costs. The tenant-specific configuration of single-tenant applications can be set at deployment time. Therefore the effect of tenant-specific variations have a negligible effect on the execution cost of single-tenant applications. Only the base storage S_0 will increase with the core application and its features. In the case of the flexible multi-tenant application, CPU usage $f_{CpuMT}(u)$ (see Eq. (2)) will increase because the tenant-specific configuration should be retrieved and activated by the `FeatureInjector`. Further, additional memory ($f_{MemMT}(t)$) and storage ($f_{StoMT}(t)$) is required to store this tenant-specific configuration and the different feature implementations. Though, these differences are not in such quantity that they will affect Eq. (4).

The impact of adding flexibility on the maintenance cost will be especially noticeable in the upgrade frequency f, because the features also have to be maintained. Since the tenant-specific configuration of a single-tenant application is set at deployment time, changes to this configuration will require additional work for the SaaS provider (C_0). We add an extra parameter c, the (average) number

of tenant-specific configuration changes which cannot be done by the tenant. Tenants of a multi-tenant application can set their tenant-specific configuration themselves. This results in no maintenance overhead for the SaaS provider.

$$Upg_{ST}(f,t,c) = t * (f_{UpgST}(f) + c * C_0) \tag{7}$$

For the administration cost, flexibility only affects the initial configuration of the application (A_0 in Eq. (6)) for both versions. In the single-tenant case this consists of setting the tenant-specific configuration, while the SaaS provider needs to specify the default configuration for the multi-tenant application.

Reengineering Costs. To add the necessary flexibility, multi-tenant applications require development support for the application developers, support to retrieve and activate the tenant-specific configurations when needed, and a configuration interface to let tenants specify their configuration based on the set of available features. Our multi-tenancy support layer provides this support: multi-tenant applications only have to interact with it. This still results in additional but limited reengineering cost, for example to define the variation points, register the features and specify the default configuration. In a single-tenant application additional reengineering is only needed to facilitate the instantiation of a tenant-specific configuration.

Providing tenants with the flexibility to customize the application, also requires the development of the different software variations. However, this is part of the core application development cost and therefore is not taken into account as reengineering cost.

4.3 Measurements

We focus on the execution cost of running the different versions on top of Google App Engine, and the reengineering cost. Since the maintenance and administration costs are hard to measure, we refer to our cost model for more details.

Execution cost. To determine the execution cost we run the four different versions of our case study application on top of GAE: a single-tenant version, a multi-tenant version, a single-tenant version with variability, and a multi-tenant version using our multi-tenancy support layer. However, we noticed that there is no difference in execution cost between the two single-tenant versions, since all variability is hard-coded. Therefore we only show the results of the default single-tenant version. Furthermore, the storage cost is not measured. Because the case study is not a data intensive application, data usage is too limited to make any conclusions about the storage cost.

In Fig. 5 we present the evolution of the average CPU usage with an increasing number of tenants. The CPU usage by the single-tenant version is linearly proportional to the number of tenants, as in Eq. (1). We also notice that the CPU usage by both multi-tenant versions is also rather linear, but lower than the single-tenant application, which differs with our cost model (see Eq. (2)). However, our cost model represents the usage of CPU by the application, while

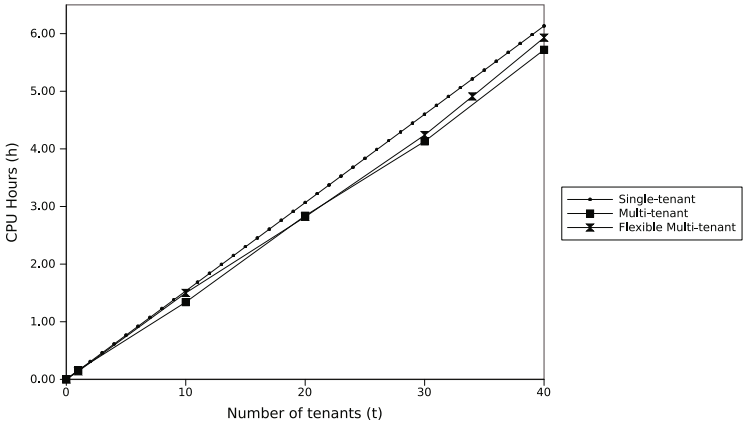


Fig. 5. Overview of the CPU usage by the different versions

on GAE the CPU time for the runtime environment is included. This is an additional cost per application and therefore has more influence on the single-tenant version. We can conclude that the multi-tenant versions require less CPU time than the single-tenant application, and that our multi-tenancy support layer shows limited overhead compared to the default multi-tenant version.

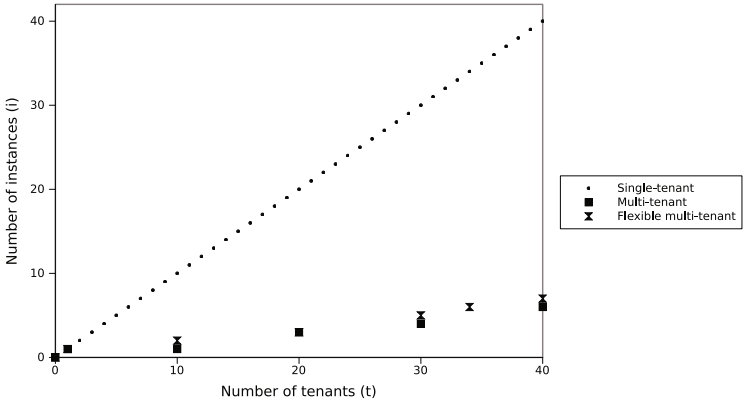


Fig. 6. Overview of the number of instances used by the different versions

The total memory usage cannot be measured precisely, because several other factors despite the application binaries add or reduce memory consumption: a rising number of requests triggers an increase in memory because a new instance (i.e. process required to handle the incoming requests) is started to provide better load balancing, and once the requests decline, instances become idle and are removed to release memory (M_0 in Eq. (1) and (2) is 0). Therefore, we use the

average number of instances to represent the maximal possible memory usage. Figure 6 shows the evolution of the average number of application instances when increasing the number of tenants. As can be seen, the difference between the single-tenant and multi-tenant versions is significant. The number of instances for both multi-tenant versions increases only slightly with the number of tenants.

Reengineering cost. Table 1 shows the quantity of source code used to develop the case study application. The engineering cost to develop multi-tenancy support is not taken into account, because this is part of the middleware. The differences in lines of source code between the single-tenant and multi-tenant versions is the reengineering cost required to let the application use the multi-tenancy support.

Table 1. Overview of the source lines of code (sloc) of the different versions

	Java	JSP	XML (config)
Default single-tenant	915	514	131
Default multi-tenant	915	514	139
Flexible single-tenant	1016	514	131
Flexible multi-tenant	1090	514	74

In the default multi-tenant version without flexibility, the developer only has to write 8 extra lines of configuration compared to the single-tenant version. This is to specify that the `TenantFilter` should be used, which uses the Namespaces API of Google App Engine to ensure data isolation.

When using our multi-tenancy support layer, the difference with the flexible single-tenant application is bigger. However, the majority of these 74 extra lines of Java code are required to use Guice, and not to use our layer. Moreover, the use of Guice resulted in a decrease of configuration lines. Furthermore, in the flexible single-tenant version the configuration is hardcoded and not user friendly. Making this more accessible for the developers to configure will result in more reengineering cost. Finally, we can conclude that adding flexibility to multi-tenant applications by means of our multi-tenancy support layer requires a limited reengineering cost. This cost consists of creating and registering features and their feature implementations, and defining the default configuration.

5 Related Work

Related work can be divided into three domains: a) middleware support for developing multi-tenant applications, b) work on customization of multi-tenant SaaS applications, and c) adaptive middleware.

Middleware Support for Multi-tenancy. Multi-tenancy is a key enabler to deliver SaaS applications with high cost effectiveness. The current state of the art especially focuses on approaches to support isolation in multi-tenant software applications [15,5]. For instance, Guo et al. [15] discuss design and implementation principles for application-level multi-tenancy, exploring different approaches to

achieve better isolation of security, performance, availability and administration among tenants.

Only a few Platform-as-a-Service (PaaS) solutions offer support to build multi-tenant applications. Google App Engine (GAE) [13] facilitates the development of multi-tenant applications via the Namespaces API. Application data is partitioned across tenants by specifying a unique namespace string for each tenant (the tenant ID). These namespaces are supported by several GAE services, such as the datastore and the caching service, enabling tenant data isolation in a transparent way. The Namespaces API is also supported by GAE's open-source implementation AppScale [6]. Other PaaS platforms supporting tenant data isolation are Apprenda SaaSGrid [1] and GigaSpaces SaaS-Enablement platform [12]. None of these platforms directly support tenant-specific customizations and therefore do not offer the same flexibility as our solution. Note that these platforms can also be used as underpinning PaaS for our approach.

In the traditional middleware space JSR 342, the Java EE 7 Specification [9], aims to enhance the suitability of the Java EE platform for cloud environments, including support for multi-tenancy. A descriptor for application metadata will enable developers to describe certain cloud-related characteristics of applications, for example by tagging them as multi-tenant or by specifying the sharing of resources. This extension of the component model with cloud-specific application metadata focuses on persistence and security. Our multi-tenancy support layer, however, offers a way to annotate points of tenant-specific variation, increasing the flexibility of multi-tenant applications, and thus is complementary.

Customization of Multi-tenant SaaS Applications. Although tenant-specific customizations are an important requirement [7,30,3], it is not trival to adapt the business logic and data to the requirements of the different tenants [15], especially in Java or .NET, the programming languages commonly used for enterprise applications.

Bezemer et al. [3] applied their multi-tenancy reengineering pattern to enable multi-tenancy in software services. This pattern requires three additional components: a multi-tenant database, tenant-specific authentication and configuration. Configuration is however limited to the look-and-feel and workflows.

In [21] variability modeling techniques from software product line engineering (SPLE) [25] are applied to support the management of variability in service-oriented SaaS applications. Application templates describe the variability via variability descriptors. Our work focuses on the realization of tenant-specific customizations in SaaS applications, which is not covered by this work.

Existing approaches for dynamic customization of multi-tenant SaaS applications utilize dynamic interpreted languages [28,22]. However, we focus on customization of enterprise multi-tier applications, which are commonly written in statically typed languages such as Java or C#. In this context, a dynamic software adaptation approach such as dynamic aspect weaving or dynamic component reconfiguration is preferred.

Adaptive Middleware. The state of the art in adaptive middleware [4,18,8,20,26] has mostly focused on adapting applications to one usage context at a time. This

means that application software is adapted by replacing an old configuration to a new configuration. In other words, the existing configuration interfaces of adaptive middleware are inherently oriented towards the dimension of the application owner or end user, but have no good ways of managing software variations on behalf of tenants. Adaptive middleware techniques include reflection and aspect-oriented development. The following paragraphs more closely relate our work to these two techniques.

Reflective middleware platforms, such as DynamicTAO [19] and OpenORB [8], provide a configuration interface to inspect and adapt the structure of applications and middleware at runtime. However, these adaptations are based on a global configuration and result in the replacement of components, thus affecting all tenants. They do not allow adaptations scoped to a specific tenant.

Aspect-oriented frameworks such as JAC [24], JBoss AOP [16] and Spring AOP [29], have improved the modularization and customization capabilities of middleware platforms and applications. By means of a declarative configuration application-specific or user-specific extensions can be weaved in where necessary. Currently also dynamic and distributed aspect weaving are supported [24,20,27], including in a reliable and atomic manner [23,32]. These AO-techniques are therefore suitable for usage in a multi-tenant context. Lasagne is an aspect-oriented middleware [33] that supports concurrent, co-existing configurations of the same application instance. This approach is however limited to traditional client-server architectures and does not support customization of multi-tenant software. Still, aspect-oriented software development (AOSD) [10] looks a promising alternative for dependency injection to support tenant-specific injections of crosscutting feature implementations.

6 Conclusion

This paper presented a reusable middleware layer on top of an existing PaaS platform to support customizable multi-tenant applications while maintaining the operational cost benefits of true application-level multi-tenancy. We have implemented a prototype on top of Google App Engine and extended the Guice dependency injection framework to achieve activation of software variations on a per tenant basis. This prototype shows improved flexibility with a minimal impact on operational costs for the SaaS provider.

Dependency injection proved to be useful to support the customization of multi-tenant applications. However, adding new features requires the introduction of new variations points in the core application. In addition, for each variation point only one software variation can be injected at a time. This complicates more advanced customizations, such as feature combinations. In this respect, AOSD is a more powerful alternative which we will investigate in the future.

A future research challenge with respect to application-level multi-tenancy is adding support for tenant-specific monitoring and ensuring performance isolation between different tenants. When performing our measurements we experienced that GAE lacks performance isolation between the different tenants. Especially

when a number of tenants heavily uses the shared application, this results in a denial of service for the end users of certain tenants. Additional support from the operating system and middleware layers is needed to ensure this performance isolation. Furthermore, tenant-specific monitoring enables SaaS providers to better check and guarantee the necessary SLAs.

Acknowledgments. We thank the reviewers for their helpful comments to improve this paper. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

1. Apprenda Inc.: SaaSGrid Middleware, http://apprenda.com/saasgrid/
2. Azeez, A., Perera, S., Gamage, D., Linton, R., Siriwardana, P., Leelaratne, D., Weerawarana, S., Fremantle, P.: Multi-tenant SOA Middleware for Cloud Computing. In: International Conference on Cloud Computing, pp. 458–465 (2010)
3. Bezemer, C.P., Zaidman, A., Platzbeecker, B., Hurkmans, T., Hart, A.: Enabling Multi-Tenancy: An Industrial Experience Report. In: ICSM 2010: International Conference on Software Maintenance (2010)
4. Blair, G.S., Coulson, G., Robin, P., Papathomas, M.: An architecture for next generation middleware. In: Middleware 1998: IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, pp. 191–206 (1998)
5. Cai, H., Wang, N., Zhou, M.J.: A Transparent Approach of Enabling SaaS Multitenancy in the Cloud. In: SERVICES-1 2010: Congress on Services, pp. 40–47 (2010)
6. Chohan, N., Bunch, C., Pang, S., Krintz, C., Mostafa, N., Soman, S., Wolski, R.: AppScale: Scalable and Open AppEngine Application Development and Deployment. In: International Conference on Cloud Computing, pp. 57–70 (2010)
7. Chong, F., Carraro, G.: Architecture Strategies for Catching the Long Tail. Microsoft Corporation (April 2006), http://msdn.microsoft.com/en-us/library/aa479069.aspx
8. Coulson, G., Blair, G.S., Clarke, M., Parlavantzas, N.: The design of a configurable and reconfigurable middleware platform. Distributed Computing 15(2), 109–126 (2002)
9. DeMichiel, L., Shannon, B.: JSR 342: Java^{TM} Platform, Enterprise Edition 7 (Java EE 7) Specification (2011), http://www.jcp.org/en/jsr/detail?id=342 (last visited at May 26, 2011)
10. Filman, R.E., Elrad, T., Clarke, S., Akşit, M.: Aspect-Oriented Software Development. Addison-Wesley, Boston (2004)
11. Fowler, M.: Inversion of Control Containers and the Dependency Injection pattern (January 2004), http://martinfowler.com/articles/injection.html
12. GigaSpaces Technologies Inc.: SaaS-Enablement Platform for ISVs, http://www.gigaspaces.com/saas-enablement
13. Google, Inc.: Google App Engine, http://code.google.com/appengine/
14. Google Inc.: Guice, http://code.google.com/p/google-guice/
15. Guo, C.J., Sun, W., Huang, Y., Wang, Z.H., Gao, B.: A Framework for Native Multi-Tenancy Application Development and Management. In: CEC/EEE 2007: International Conference on E-Commerce Technology and International Conference on Enterprise Computing, E-Commerce, and E-Services, pp. 551–558 (2007)

16. JBoss Community: JBoss AOP, http://www.jboss.org/jbossaop/
17. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. 21, SEI, CMU, Pittsburgh, PA (1990)
18. Kon, F., Costa, F., Blair, G., Campbell, R.H.: The case for reflective middleware. Commun. ACM 45(6), 33–38 (2002)
19. Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., Pasquale, F.: Monitoring, Security, and Dynamic Configuration with the *dynamicTAO* Reflective ORB. In: Coulson, G., Sventek, J. (eds.) Middleware 2000. LNCS, vol. 1795, pp. 121–143. Springer, Heidelberg (2000)
20. Lagaisse, B., Joosen, W.: True and Transparent Distributed Composition of Aspect-Components. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 42–61. Springer, Heidelberg (2006)
21. Mietzner, R., Metzger, A., Leymann, F., Pohl, K.: Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications. In: PESOS 2009: ICSE Workshop on Principles of Engineering Service Oriented Systems, pp. 18–25 (2009)
22. Müller, J., Krüger, J., Enderlein, S., Helmich, M., Zeier, A.: Customizing Enterprise Software as a Service Applications: Back-End Extension in a Multi-tenancy Environment. In: Filipe, J., Cordeiro, J. (eds.) Enterprise Information Systems. LNBIP, vol. 24, pp. 66–77. Springer, Heidelberg (2009)
23. Nicoară, A., Alonso, G.: Dynamic AOP with PROSE. In: ASMEA 2005: Workshop on Adaptive and Self-Managing Enterprise Applications, pp. 125–138 (2005)
24. Pawlak, R., Seinturier, L., Duchien, L., Florin, G.: JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In: Matsuoka, S. (ed.) Reflection 2001. LNCS, vol. 2192, pp. 1–24. Springer, Heidelberg (2001)
25. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, New York (2005)
26. Rouvoy, R., Eliassen, F., Floch, J., Hallsteinsen, S., Stav, E.: Composing Components and Services Using a Planning-Based Adaptation Middleware. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 52–67. Springer, Heidelberg (2008)
27. Rouvoy, R., Eliassen, F., Beauvois, M.: Dynamic planning and weaving of dependability concerns for self-adaptive ubiquitous services. In: SAC 2009: Symposium on Applied Computing, pp. 1021–1028 (2009)
28. Salesforce.com, Inc., http://www.salesforce.com
29. SpringSource: Aspect Oriented Programming with Spring, http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/aop.html
30. Sun, W., Zhang, X., Guo, C.J., Sun, P., Su, H.: Software as a Service: Configuration and Customization Perspectives. In: SERVICES-2 2008: Congress on Services Part II, pp. 18–25 (2008)
31. Tao, L.: Shifting paradigms with the application service provider model. Computer 34(10), 32–39 (2001)
32. Truyen, E., Janssens, N., Sanen, F., Joosen, W.: Support for Distributed Adaptations in Aspect-Oriented Middleware. In: AOSD 2008: International Conference on Aspect-oriented Software Development, pp. 120–131 (2008)
33. Truyen, E., Vanhaute, B., Jørgensen, B.N., Joosen, W., Verbaeten, P.: Dynamic and selective combination of extensions in component-based applications. In: ICSE 2001: International Conference on Software Engineering, pp. 233–242 (2001)

Bridging the Interoperability Gap: Overcoming Combined Application and Middleware Heterogeneity

Yérom-David Bromberg[1], Paul Grace[2], Laurent Réveillère[1], and Gordon S. Blair[2]

[1] LaBRI, University of Bordeaux, France
{david.bromberg,laurent.reveillere}@labri.fr
[2] School of Computing and Comunications, Lancaster University, UK
p.grace@lancaster.ac.uk, gordon@comp.lancs.ac.uk

Abstract. Interoperability remains a significant challenge in today's distributed systems; it is necessary to quickly compose and connect (often at runtime) previously developed and deployed systems in order to build more complex systems of systems. However, such systems are characterized by heterogeneity at both the application and middleware-level, where application differences are seen in terms of incompatible interface signatures and data content, and at the middleware level in terms of heterogeneous communication protocols. Consider a Flickr client implemented upon the XML-RPC protocol being composed with Picasa's Service; here, the Flickr and Picasa APIs differ significantly, and the underlying communication protocols are different. A number of ad-hoc solutions exist to resolve differences at either distinct level, e.g., data translation technologies, service choreography tools, or protocol bridges; however, we argue that middleware solutions to interoperability should support developers in addressing these challenges using a unified framework. For this purpose we present the Starlink framework, which allows an interoperability solution to be specified using domain specific languages that are then used to generate the necessary executable software to enable runtime interoperability. We demonstrate the effectiveness of Starlink using an application case-study and show that it successfully resolves combined application and middleware heterogeneity.

Keywords: Application, Middleware, Interoperability, Evolution, Domain Specific Languages, Automata.

1 Introduction

Nowadays, complex distributed systems are composed from systems that are developed independently of one another (including legacy systems). This composition occurs either statically, or at runtime as in the case of spontaneous interactions between mobile and pervasive systems. However, existing systems are highly heterogeneous in their interaction methods making such composition challenging.

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 390–409, 2011.

Applications and systems are developed using a multitude of incompatible middleware abstractions and protocols. For example, remote procedure call protocols such as SOAP and IIOP differ in message content, message format, and addressing meaning that they cannot directly interoperate. The range of incompatible protocols drastically limits interoperability, and thus the practical benefit of systems composition. Protocol standardization should address this issue but has been demonstrably ineffective in practice. Indeed, new competing protocols are frequently introduced to cope with the emergence of new application domains (e.g. sensors, ad-hoc networks, Grid Computing, Cloud Computing, etc.), whereas standardization is slow to complete in comparison.

Interoperability is the ability of one or more systems to exchange and understand each other's data. However, there can be significant mismatches between the interfaces of various systems that provide similar application functionalities, making interoperation impossible. Indeed, developers often implement similar application functionalities in different ways, resulting in incompatible operation signatures and data types. In addition, the behavior of the interfaces may also differ, e.g. a single operation in one case may correspond to a sequence of operations in another.

Existing solutions to these interoperability challenges have generally made assumptions about one another, e.g. that the application is fixed and the protocol heterogeneity must be resolved, or the protocol is common and application differences must be addressed. The former is the view of middleware-based solutions such as protocol bridges [1], Enterprise Service Buses, and interoperable middleware [3] [7] [16]. However, none of these solutions work when there is a difference in application functionalities. For example, in a protocol bridge even a simple difference in the operation name breaks the solution. Service Choreography and Workflow execution languages and tools underpinned by Business Processing Execution Language (BPEL) offer methods to overcome application differences but commonly assume an underlying service platform and description language, e.g. SOAP and WSDL. As a consequence, these approaches are not fit for purpose when applications rely on different middleware. Overall, there is no consistent view of how to tackle problems where both application and middleware heterogeneity is encountered in combination. This leads to the use of solutions involving ad-hoc integration of a number of different technologies.

Due to the potential differences in both middleware protocols and application behavior, a single universal bridge can not be developed to address the heterogeneity issue. Instead, a mediated solution is required in each specific case. Many protocol and application specific mediators are thus required to cover the broad solution space. Nevertheless, such a mediator needs to be dynamically generated to manage the runtime composition of services, because developing this mediator for each particular case can be a challenge for many application programmers. To address this issue, we argue that a domain-specific modelling approach can be used for describing application and protocol specificities.

This paper proposes the following contributions towards reaching this goal:

- *Application and protocol models.* We use automata to model application behaviour where a transition represents the application action and associated input and output data. Similarly, we use automata to model middleware protocols where a transition represents either a sent or received message.
- *Application-Middleware Mediators.* A merged automaton models the merge of two application automata, i.e., this states how the application states from one system are merged with the states of the other heterogeneous system. This mediator model is then used to generate a concrete application-middleware mediator that binds application transitions to physical middleware protocol messages.
- *An Interoperability Framework.* We have implemented a middleware framework to support the generation and execution of mediators. The Starlink Framework [2] interprets a concrete merged automaton to enable dynamic interoperability at both the application and protocol level. In previous work we have described how Starlink is used to achieve middleware protocol interoperability; here we expand on the approach to achieve combined application and middleware interoperability.

The remainder of the paper is structured as follows. Section 2 presents a motivating case study to highlight the interoperability challenges. In Section 3, we introduce the application and protocol models. Subsequently, in Section 4 we describe how the interoperability framework realizes and executes these models. Our case-study based evaluation is presented in Section 5 and an analysis of related work is provided in Section 6. Finally, we draw conclusions in Section 7.

2 Motivation: Flickr and Picasa Case Study

To highlight the problem of combined application and middleware heterogeneity we examine the Flickr and Picasa API services, highlight the interoperability challenges and then identify the requirements to overcome them.

2.1 Observing Application and Middleware Heterogeneity

Flickr and Picasa are both Web based services that provide similar application functionality. They allow client applications to view, search, add, edit and delete photographs. In addition, they both allow comments to be added to individual photos or sets of photographs. Although they offer similar services, clients of both can not be composed with the services of the other. In practice, interoperability between the two is hindered due to the heterogeneity at both the application level and at the protocol level.

Application Heterogeneity. The APIs of Flickr[1] and Picassa[2] are large and complex (Flickr has over 100 operations available); hence we concentrate on a

[1] http://www.flickr.com/services/api/
[2] http://code.google.com/apis/picasaweb/

```
flickr.photos.search(api_key, tags, text, per_page, page, ...)
flickr.photos.getInfo(api_key, photo_id)
flickr.photos.comments.getList(api_key, photo_id, min_comment_date, max_comment_date)
flickr.photos.comments.addComment(api_key, photo_id, comment_text)
```

```
PicasaBaseURL - https://picasaweb.google.com/data/feed/api
photos.search(q, max-results)   [GET PicasaBaseURL/all?q=tree&max-results=3]
getComments(kind)   [GET PhotoURL?kind=comment]
addComment(entry)  [POST PhotoURL, <entry> </entry>]
```

Fig. 1. Highlighting the Flickr and Picasa APIs

small subset of the behavior available. Fig. 1 illustrates how the APIs of Flickr and Picasa offer a set of operations for performing similar application requirements; namely, performing a keyword search on publicly searchable photos, listing the comments that have been added to a particular photo result and then finally adding a comment to that same photograph. From these APIs it is clear that application heterogeneity exists in the following two distinct ways:

1. The interface signatures contain sets of operations that differ in operation name, and the types of input and output data of the operation. Consider the operation to perform a general keyword search of public photographs. The Flickr API provides the `search` operation with a number of parameters including the optional `text` parameter for the keyword and `page` and `per_page` parameters to restrict the returned results; alternatively, Picassa provides a search operation with input parameters: **q** for the keyword and `max-results` to restrict the results (n.b. the GET syntax is also shown).
2. The application behavior is captured in different behavior sequences. The Flickr `search` operation returns a set of identifiers. The `getInfo` operation should then be called to obtain more information about the photo, including the URL of the jpeg. Alternatively, the Picasa search operation returns the information about the photograph directly in the search results.

Middleware Heterogeneity. The Flickr and Picassa APIs also differ in the protocols they use to access the services. Picassa provides only a RESTful implementation atop HTTP with the Google Data API as an associated data model, whereas Flickr relies either on REST, SOAP, or XML-RPC. A Flickr client (e.g. a smartphone application) implemented using either SOAP or XML-RPC cannot interoperate with Picasa due to the protocol heterogeneity.

2.2 Interoperability Requirements

Heterogeneous systems that have been developed independent to one another can not interoperate. To address this issue, mediators need to be created and deployed in the network, so as to provide dynamic composition of existing systems. However, the development of such mediators must consider the following factors:

- The extreme heterogeneity in applications and middleware suggests ad-hoc, manually coded solutions will lead to significant development costs and consumption of computational resources due to the continuous redevelopment of equivalent solutions.
- When a new middleware protocol emerges, the API of a service may be migrated to it. Therefore, both existing clients and interoperability mediators for this service would no longer operate correctly.
- Similarly, when a new version of an API is released, any changes to the syntax or behavior of the API may mean that the existing clients or interoperability mediators that rely on this API no longer function.

As a consequence, to overcome the combined application and middleware heterogeneity, we propose two key requirements. First, mediators that act as interoperability enablers must be automatically generated and dynamically deployed. Second, middleware protocol migration and API evolutions must be handled with minimal development effort. In [21], Vinoski argues that interoperability is a mapping problem and that diversity and heterogeneity should be embraced rather than attempt to homogenize distributed systems. Therefore, developers should be supported in creating these mappings. Hence, in this paper we first propose a high-level, model-based specification of the application differences (independent of any middleware protocol); we subsequently propose that this be used to generate the concrete mediator by binding the solution to particular protocol-to-protocol use cases. For example, an application model of the differences between Flickr and Picasa generates an XML-RPC to REST application-specific mediator or a SOAP to REST application-specific mediator.

3 Models

Modern software development trends imply that developers implement applications through the use of reusable API operations, that, in a distributed environment, are remotely invoked through the use of an underlying middleware. For instance, Flickr, Picasa, Bing and/or Google maps API define a set of remote operations that can be invoked with different kind of middleware. The way operations are combined together by developers to perform a particular task depends on particular constraints related to the APIs used, and consequently defines an *API usage protocol*. Inherently, applications performing similar functionalities (i.e. semantically equivalent) but implemented with different APIs, behave differently, and thus have a different API usage protocol. Providing interoperability among applications based on heterogeneous APIs requires first to capture formally their respective APIs usage protocol in order to reason about their behavior.

3.1 APIs Usage Protocol

An API usage protocol $\mathcal{S}$ defines sequences of ordered operation invocations. Signatures of invoked operations are expressed in terms of input and/or output

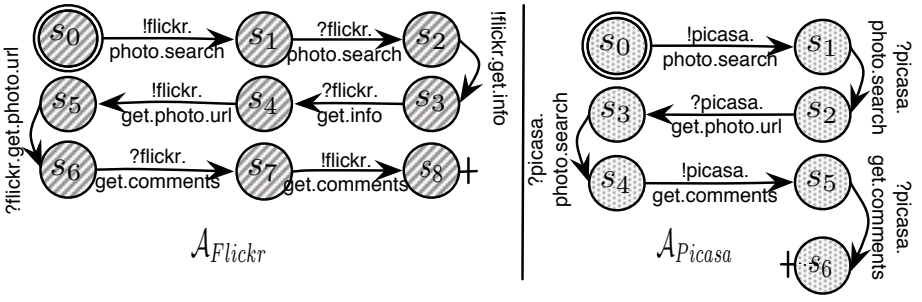


Fig. 2. Flickr and Picasa usage protocol APIs

messages, more precisely in terms of messages exchanged (as developers of Web Services are used to). Syntactical description of message data fields, including their data types are formalized through the use of *abstract messages*. An abstract message consists of a set of fields, either primitive or structured [2]. The former is composed of: (i) a label naming the field, (ii) a type describing the type of the data content, (iii) a length defining the length in bits of the field, and (iv) the value, i.e., the content of the field. A structured field is composed of multiple primitive fields. Hence, we abstract an operation invocation request, noted $rvalue\ operation(arg_1...arg_n)$, as two abstract messages. First, an abstract message named *operation* that is sent and which is composed of a set of n fields such as $field_1 = arg_1, ..., field_n = arg_n$. Second, an abstract message named *rvalue* that is received. We note $msg \triangleright field$ the operation that selects the field $field$ from the abstract message msg.

As a result, a sequence of operation invocations $\mathcal{S}$ describing an API usage protocol is formalized as an automaton $\mathcal{A}_\mathcal{S}$ with edges labeled with abstract messages sent or received according to the signature of remote operations invoked. More formally, $\mathcal{A}_\mathcal{S}$ is defined as a 6-tuple such as $\mathcal{A}_\mathcal{S} = (Q, M, q_0, F, Act, \rightarrow)$ with Q a finite set of states, M a finite set of both incoming or outgoing abstract messages, $q_0 \in Q$ the starting state and $F \subset Q$ a set of accepting states. $Act = \{?, !\}$ defines two kinds of actions: ! to invoke a remote operation and ? to receive a reply from a previously invoked remote operation.

Hence, the transition relation, noted $\rightarrow \subseteq Q \times Act \times M \times Q$, can be either an *invoke-transition* or a *receive-transition*. The former is noted $s_1 \xrightarrow{!operation} s_2$ for $(s_1, !, operation, s_2) \in \rightarrow$ and indicates the next state to which the automaton passes as soon as the operation *operation* is invoked. The latter has the following form $s_1 \xrightarrow{?rvalue} s_2$ for $(s_1, ?, rvalue, s_2) \in \rightarrow$ and changes the state of the automaton from s_1 to s_2 once the invocation reply *rvalue* is received. As a result, $\mathcal{A}_\mathcal{S}$ acts as a call graph of invoked operations and specifies the order in which they should be invoked.

For instance, Fig. 2 demonstrates the Picasa and Flickr API usage protocols that developers might follow to implement either a Picasa or a Flickr application with similar functionalities.

3.2 API Usage Protocol Mismatches

From an application perspective, two applications, $\mathcal{A}_1$ and $\mathcal{A}_2$, which follow respectively an API usage protocol $\mathcal{A}_{\mathcal{S}1}$, and an API usage protocol $\mathcal{A}_{\mathcal{S}2}$, may interact seamlessly with each other if and only if there is a way to intertwine their respective API usage protocols. Performing this kind of merging assumes to resolve different kinds of mismatches. As we express operation invocations in terms of messages exchanged, we can leverage on guidelines of possible mismatches that have already been identified for developing Web services adapters [13] and apply them to API usage protocol mismatches. For instance, the comparison of two API usage protocols such as $\mathcal{A}_{Flickr}$ and $\mathcal{A}_{Picasa}$, depicted in Fig. 2, enables us to point out the different mismatches that occur:

Ordering mismatch. When applications invoke similar remote operations in a different order, an ordering mismatch may occur. For instance, according to $\mathcal{A}_{Picasa}$, a Flickr developer should invoke a `getPhotoUrl` operation right after a `photoSearch`. However, the `getPhotoUrl` operation is called, in fact, later in the call graph.

Extra or missing message mismatch. If one application invokes a remote operation that another application never invokes, there is an extra or missing message mismatch. For instance, a Picasa developer does not invoke any operations similar to the Flickr operation `getInfo`, which is specific to the Flickr API.

One-to-many mismatch. An API can perform a particular task with only one remote operation, whereas another API may require several operations to do a similar task. For instance, obtaining a photo URL requires only one `search` operation using Picasa, whereas it requires two operations (i.e. `search` and `getInfo`) with Flickr.

In the context of an API usage protocol, the aforementioned mismatches emerge as soon as there are mismatches among operations at their signature level. As a result, there is not always a one-to-one mapping between messages. Note that resolving API usage protocol heterogeneity is theoretically similar to resolving heterogeneity at the protocol layer but acts on messages that abstract operation invocations instead of network messages. So resolving operation signatures mismatches leads us to reason about semantic equivalence among the abstract messages exchanged. To this end, we extend our model with a semantic equivalence operator $\cong$ that acts on messages, abstracting operations, as defined below.

Definition 1. *Let $\overrightarrow{m}$ a sequence of abstract messages. Further, $!m$ or $?m$ denotes a message to be sent or received, and $!s_i.m$ or $?s_i.m$ denotes a message sent or received in a specific state s_i.*

Definition 2. *Let $\cong$ a semantic equivalence operator such that $n \cong \overrightarrow{m}$ is true if and only if for every mandatory field of n, noted $\mathcal{M}_{fields}(n)$, there exists a semantically equivalent field in one message of the sequence $\overrightarrow{m}$. So $n \cong \overrightarrow{m}$ if and only if $\forall n \triangleright field \in \mathcal{M}_{fields}(n), \exists m \in \overrightarrow{m} = \langle m_1 \dots m_n \rangle$ such as $n \triangleright field \models m \triangleright field$.*

3.3 k-Colored Automata: Intertwining API Usage Protocol

Informally, application $\mathcal{A}_1$ may interact with application $\mathcal{A}_2$ if the following conditions are satisfied: (i) their respective API usage protocols $\mathcal{A}_{\mathcal{S}1}$ and $\mathcal{A}_{\mathcal{S}2}$ share a sufficient number of similar operations that enables them to have a successful sequence of operations to reach their respective final state, (ii) the identified semantically equivalent can be intertwined together,i.e. invoked in an alternate order when required. In other terms, invoked operations, i.e. request messages, from $\mathcal{A}_1$ must be sequentially translated into a semantically equivalent request message followed by a corresponding reply message from $\mathcal{A}_2$. Based on the previous introduced definition, we extend the model with a history and an intertwining operator to formally define the aforementioned constraints.

Definition 3. *Let $\mathcal{I}(\mathcal{A}_\mathcal{S})$ the set of initial states and $\mathcal{ENDD}(\mathcal{A}_\mathcal{S})$ the set of final states of $\mathcal{A}_\mathcal{S}$. The set of all states of $\mathcal{A}_\mathcal{S}$is $States(\mathcal{A}_\mathcal{S}) = \mathcal{I}(\mathcal{A}_\mathcal{S}) \cup \mathcal{ENDD}(\mathcal{A}_\mathcal{S})$. Further, let $\mathcal{M}_{sg}$ $(\mathcal{A}_\mathcal{S})$ the set of all messages and $\mathcal{T}(A_S)$ the set of all transitions of $\mathcal{A}_\mathcal{S}$.*

Definition 4. *Let $\Rightarrow$ the history operator defined such as $\Rightarrow \subseteq States(A_S) \times Act \times \overrightarrow{m} \times States(A_S)$ with $Act = \{!, ?\}$ and $\overrightarrow{m} = \{m_i, ..., m_k, ..., m_n\} \in \mathcal{M}_{sg}(A_S)$ with $(i, k, n) \in \{1, .., n\}$. Thus, $s_1 \overset{!\overrightarrow{m}}{\Longrightarrow} s_2$ (resp. $s_1 \overset{?\overrightarrow{m}}{\Longrightarrow} s_2$) gives the sequence of abstract messages sent (resp. received) from the state s_1 to s_2.*

Definition 5. *Let $\rightsquigarrow$ the intertwining operator such as $!s_i.method_1 \rightsquigarrow !s_j.method_2$ is true iff*
$\exists s_0 \in \mathcal{I}(\mathcal{A}_{S1}).\exists s_i \in States(\mathcal{A}_{S1}).\exists s_j \in States(\mathcal{A}_{S2}).$
$\exists method_1 \in \mathcal{M}_{sg}(A_{S1}).\exists method_2 \in \mathcal{M}_{sg}(A_{S2})|$
$((?method_1 \cong ?method_2) \vee (?method_2 \cong (s_0 \overset{?\overrightarrow{m}}{\Longrightarrow} s_i, ?method_1)) \wedge$
$!s_j.method_2 \cong (s_0 \overset{?\overrightarrow{m}}{\Longrightarrow} s_i, s_0 \overset{!\overrightarrow{m}}{\Longrightarrow} s_i)$
Reciprocally, $?s_i.method_1 \rightsquigarrow ?s_j.method_2$ is true iff $!s_i$ $.method_1 \rightsquigarrow !s_j.method_2$.

Thus, if $\mathcal{A}_{\mathcal{S}1}$ has a sequence of n intertwined operations with $\mathcal{A}_{\mathcal{S}2}$, it means that there exists n transitions, named γ-transitions, that go back and forth between $\mathcal{A}_{\mathcal{S}1}$ and $\mathcal{A}_{\mathcal{S}2}$ without sending or receiving messages but applying successful data transformation on semantically equivalent messages as described in Section 4. As a result, the resulting automaton is said to be a k-colored automaton. The k color enables one to identify states that belong to either $\mathcal{A}_{\mathcal{S}1}$ or $\mathcal{A}_{\mathcal{S}2}$ as depicted in Fig. 3. Further, states linked by a $\gamma-$transition are represented by bicolored nodes such as nodes ❶, ❷, ❸, ❹,❺, ❻. For instance, at node ❶, Flickr `photoSearch` invocation can be intertwined with the corresponding Picasa `photoSearch` as $!flickr.photoSearch \cong !picasa.photoSearch$, and a $\gamma-$transition is taken to move from the Flickr API usage protocol to the Picasa one through some translations on data fields as messages are semantically equivalent.

Definition 6. *A k-colored automaton is an automaton with all its states colored by a color k. Thus an automaton $\mathcal{A}_{\mathcal{S}}$ colored by a color k is noted ${\mathcal{A}_{\mathcal{S}}}^k$ where $States({\mathcal{A}_{\mathcal{S}}}^k) = \{s_0^k, ..., s_i^k, ..., s_n^k\}$.*

Definition 7. *An application $\mathcal{A}_1$ may interact with an application $\mathcal{A}_2$ iff their colored API usage protocol A_{S1}^1 and A_{S2}^2 are mergeable, and noted $A_{S1}^1 \bigoplus A_{S2}^2$, such that $\exists Seq = \{..., (s_x^1, s_y^2), ...\} \subseteq States(A_{S1}^1) \times States(A_{S2}^2) with(x, y) \in \{1, ..., n\} \wedge \exists(!m_1, !m_2) \subset \mathcal{M}_{sg}(A_{S1}^1) \times \mathcal{M}_{sg}(A_{S2}^2) \mid \{!s_x.m_1 \rightsquigarrow !s_y.m_2\} \bigwedge \exists(s_i, s_j) \subset \mathcal{ENFD}(A_{S1}^1) \times \mathcal{ENFD}(A_{S2}^2)$ with $(i, j) \in \{1, ..., n\} \mid s_i, s_j \in \mathcal{ENFD}(A_{S1}^1 \bigoplus A_{S2}^2)$.*

Note that all invocation operations from $\mathcal{A}_{\mathcal{S}1}$ can not be always intertwined with the ones of $\mathcal{A}_{\mathcal{S}2}$, but it does not hinder necessarily the interoperation. Hence, it is required to consider that it is possible to get two different kinds of k-colored automaton: strongly or weakly merged. The former case still arises even if some invocation operations from $\mathcal{A}_{\mathcal{S}1}$ are not intertwined, however their corresponding replies must be semantically equivalent to replies received from $\mathcal{A}_{\mathcal{S}2}$. Otherwise, if this condition is not satisfied the k-colored automaton is said to be weakly merged. For instance, in Fig. 3, the Flickr operation `getInfo` has no equivalent in the Picasa API, however, its reply is semantically equivalent to the Picasa `photoSearch` reply previously received. The flickr `getInfo` operation can be invoked (i.e through the send and receive of $flickr.getInfo$) without both being interleaved and hindering the interoperation. The depicted automaton is still strongly merged.

Definition 8. *The resulting $A_{S1}^1 \bigoplus A_{S2}^2$ is a k-colored automaton, with $k = \{1, 2\}$, defined as a $7-tuple$ $(Q, M, q0, F, Act, \rightarrow, \xrightarrow{\gamma}, P, \cong)$ where $Q = \bigcup_{k=1...2} States(A_{Sk}^k)$, $M = \bigcup_{k=1...2} \mathcal{M}_{sg}(A_{Sk}^k)$, q_0 a starting state $\in \mathcal{I}(A_{S1}^1)$, $F = \bigcup_{k=1...2} \mathcal{ENFD}(A_{Sk}^k)$, $P = \{\lambda\}$ a set of data transformations on messages semantically equivalent according to the $\cong$ relation, and $\rightarrow = \bigcup_{k=1...2} \mathcal{T}(A_{Sk}^k)$. Finally, $\xrightarrow{\gamma} \subseteq States(A_{S1}^1) \times P \times States(A_{S2}^2)$ are $\gamma-$transitions that occur when sent (or received messages) from $States(A_{S1}^1)$ can be interleaved with the ones from $States(A_{S2}^2)$ according to the $\rightsquigarrow$ operator. $\gamma-$transitions take the form $s_i \xrightarrow{\gamma(\{\lambda\})} s_j$. The operator $\cong$ is defined as previously.*

Note that a data transformation $\lambda_i \in \{\lambda\}$ is a function $\lambda_i : field_1 \times ... \times field_n$ that performs a data transformation and may require as arguments some fields extracted from previously received messages.

4 Applying the Starlink Framework

Starlink is a runtime middleware framework which provides an engine to dynamically interpret and execute middleware models. The key design principles are based upon the knowledge that middleware technologies are built upon message-based solutions, i.e., middleware protocols consist of sending to and receiving messages from a network. We have previously documented how the framework [2]

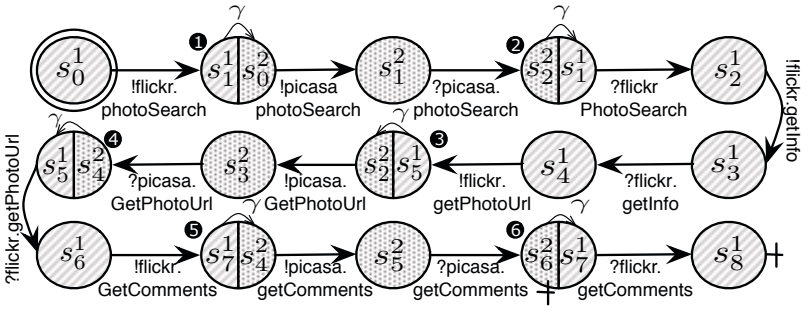


Fig. 3. Merged Flickr and Picasa usage protocol API

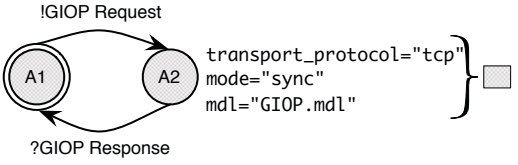
can be used to dynamically generate direct protocol bridges (i.e. connecting middleware protocols of similar types, such as service discovery and RPC). We now describe how it can be used more broadly to develop and deploy *application-middleware mediators*. We first describe the models used by Starlink and then how they are executed. Subsequently, we describe how the application models introduced in Section 3 are used to generate the Starlink executable models.

4.1 Starlink Models

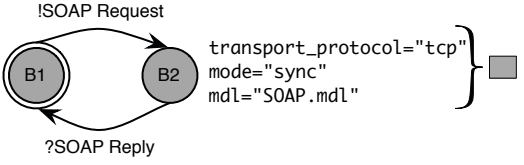
In this section we introduce the core models that are interpreted by Starlink. Firstly, how protocol message sequences are specified. Secondly, how message format is defined. Thirdly, how message translation logic is described.

Message Sequences. The behavior of a protocol is traditionally described by an automaton where transitions represent message exchanges. However, protocols vary in their interaction with the network, in terms of the transport protocol used, whether requests are sent by unicast or by multicast, and whether responses are received synchronously or asynchronously. Starlink uses the previously introduced *k-colored automata* to capture the properties of a protocol by a color k and ensure that the messages are executed using the appropriate network services [2]. Fig. 4(a) illustrates the k-colored automaton for general IIOP client behaviour, i.e. a GIOP request message is sent synchronously to an IIOP server and on the same connection it receives the GIOP reply message. Fig. 4(b) illustrates SOAP client behaviour.

Message Format and Content. A network message is organized as a sequence of text lines, or of bits, for a binary protocol, containing both fixed elements and elements specific to a given message. Extracting values from a message represented as a sequence of text or binary characters is unwieldy, and creating messages is even more complex, because the element values may become available at different times, making it difficult to predict the message size and layout. Hence, we have proposed a domain specific language approach to describe messages such that the required message parsers and composers can be generated automatically.



(a) IIOP client k-colored automaton



(b) SOAP client k-colored automaton

Fig. 4. Examples of concrete k-colored automata

```
<Message:GIOPRequest>
<Rule:MessageType=0>
<RequestID:32><Response:8>
... <ObjectKeyLength:32><ObjectKey:ObjectKeyLength>
... <OperationLength:32><Operation:OperationLength>
... <align:64><ParameterArray:eof>
<End:Message>

<Message:GIOPReply>
<Rule:MessageType=1>
<RequestID:32><ReplyStatus:32><ContextListLength:32>
...<align:64><ParameterArray:eof>
<End:Message>
```

Fig. 5. MDL specification of the GIOP message format

The Starlink framework is flexible to allow different types of language to be used to specify message formats; each language can be termed a Message Description Language (MDL). This flexibility better supports the parsing and composing of a wide range of protocols. For example, specialised languages for binary messages, text messages and XML messages can be plugged into the framework. From an MDL specification, Starlink dynamically generates parsers that transform network messages to the *abstract message representation*. Reciprocally, the generated composers do the reverse. An example of MDL specification for GIOP messages is presented in Fig. 5. Detailed discussion of the language is left from here, and further information is available in [2].

Message Translations. When several protocols need to interoperate, it is necessary to express the relation among them and to describe the *message translation logic (MTL)*, which defines how to translate messages from one protocol to another. Translation logic is used to describe the translation of data and behaviour where messages are semantically equivalent, i.e. the messages

perform similar operations. This logic is executed at the bi-colored states of a colored automata and typically consists of field transformation where a field in the message to be composed is assigned a value from a received field (there will typically be a transformation function as part of this assignment). One key operator of the MTL language is the *assignment operation*.

4.2 The Starlink Framework: Dynamically Interpreting Middleware Models

As illustrated in Fig. 6, the Starlink framework interprets the previously described middleware models at runtime in order to support the necessary middleware behaviour on demand.

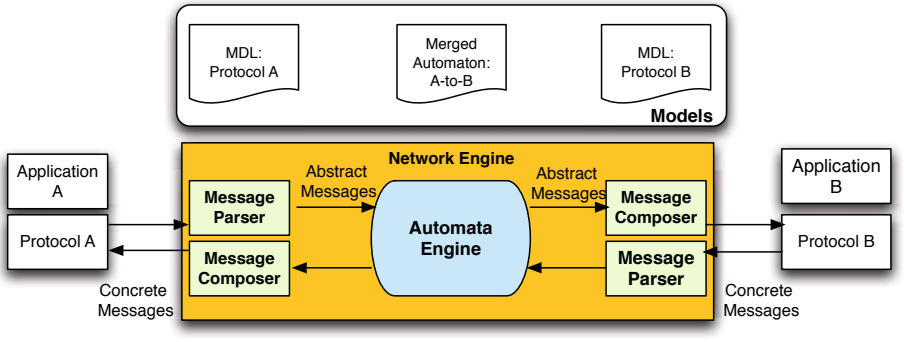


Fig. 6. Architecture of the Starlink framework

The *network engine* sends and receives physical messages (i.e. data packets) to and from the network. A transition in the k-colored automata attaches network semantics to describe the requirements of the network. The network engine then provides the services to meet these requirements, which could include different types of transport or multicast behaviour. The current implementation of the network engine provides traditional TCP and UDP services for infrastructure networks. However, the architecture is configurable so that if Starlink were to be deployed in more heterogeneous environments, e.g. ad-hoc networks, this network engine could be replaced with configurable services for ad-hoc routing [18].

The *message parsers* read the contents of a network packet and parse them into the *AbstractMessage* representation such that the data can be manipulated during the mediation process. For example, if a HTTP message is received a HTTP parser reads all the fields of the header and body. Correspondingly, *message composers* construct the data packet for a particular protocol message, e.g. constructing the content for a HTTP GET message. Importantly, the message composers and parsers are generic reusable software elements that interpret high-level specifications of message content. The Message Description Language

(MDL) specification (as described previously) specializes these generic components at runtime to create a specific protocol parser or composer.

The *automata engine* executes the behaviour of the merged automata, i.e. it controls the sequence of sending, receiving, parsing, composing and translation of messages. In Starlink, there a three types of states: i) a *receiving* state waits to receive a message and will only follow a matching receive transition when a matching message is received; ii) a *sending* state sends a message described in the single transition; iii) a *no-action* state is a translation state that translates data from the fields on one or more of the prior messages into the message to be constructed.

4.3 Generating and Executing Application-Middleware Automata

API usage protocol automata are specified independent from particular middleware. We now describe how these are bound to a specific protocol to create a Starlink executable k-colored automaton specific to the API implementation.

As described in Section 3 the API usage protocol automaton defines the application `actions` in terms of sending actions (invocation) and receiving actions (reply response). These transitions contain the *action label* and the *abstract message* that includes the input or output data values. Actions correspond to distributed interactions. However, they cannot be executed because they do not relate to a specific communication protocol. Indeed, the labels and data are only made concrete using protocol messages. Therefore, the API usage automaton must be bound to a concrete protocol automaton in order to be executed. We term the resulting model an *application-middleware automaton*. To better illustrate this procedure, Fig. 7 shows how a simple API usage protocol automaton is bound to two heterogeneous middleware protocols, namely IIOP and SOAP. The client application performs an addition operation (Add) from a remote service. For this, it sends an `Add action`, followed by the reception of the `Add action response`. The input values consist of the `x` and `y` integer parameters to be added. The output value is the returned integer parameter `z`.

To bind to a particular protocol we require: i) the k-colored automaton of the middleware protocol (e.g. Fig. 4(a)), ii) the MDL specification of that protocol's messages (e.g. Fig. 5) and iii) the set of rules that describe how a particular protocol (e.g. GIOP) is bound to the application automata concepts (i.e. the action labels, and the parameters). The rules to bind applications to SOAP in one case and IIOP in the other are illustrated in Fig. 7. IIOP and SOAP are both RPC protocols and hence the actions correspond to the request and response messages of each protocol, as seen by the corresponding k-colored sequence. The action label then binds to specific fields within the message described by MDL: the `operation` field of the GIOP Request message, and the `methodname` field of the SOAP request envelope. Similarly, the request action parameters (the x and y integers) relate to the first two parameters in the ParameterArray field of the GIOP Request message. The return value parameter (the z integer value) relates to the first parameter of the GIOP reply ParameterArray.

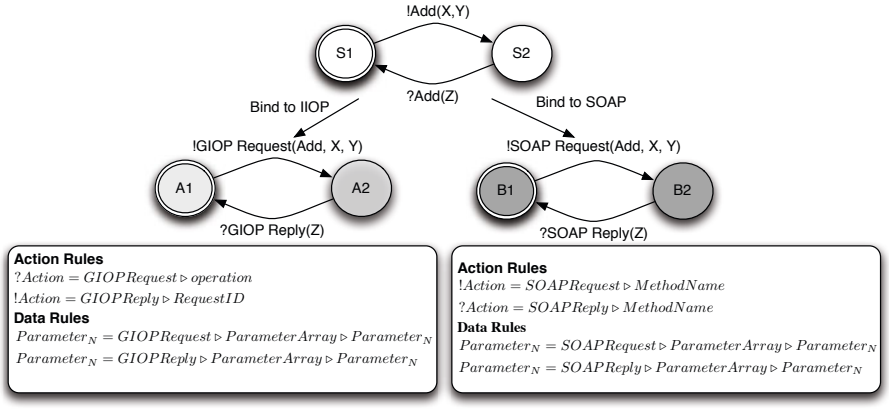


Fig. 7. Binding to concrete application-middleware automata

Starlink is then able to execute the application-middleware automaton in order to concretely achieve the application behaviour. At receiving states, the automata engine waits for middleware messages of a particular type (e.g. a SOAP Reply) and also with a particular action label (e.g. add). Subsequently, at send states the middleware message (e.g. SOAP request) is constructed placing the appropriate application labels and input values in the identified fields as described by the protocol binding rules.

4.4 Generating and Executing the Intertwining API Usage Protocol Automata

A similar binding process is carried out to generate the concrete version of an intertwining API Usage Protocol Automaton, i.e., where two heterogeneous applications are merged. For transitions, the bindings are identical to those explained in section 4.3; the difference occurs at the bi-colored states where MTL rules must be executed to translate application data from a parsed message into the composed message. In this situation we must generate the concrete MTL rules relating to the MTL definitions in the Intertwining automaton.

To illustrate this procedure we continue with the simple addition example. In this case the SOAP service provides an add operation with an `int Plus(int, int)` signature whereas the IIOP client interface signature is `int Add(int, int)`. Hence, the application difference is in the operation name. Fig. 8 shows how the merged application automaton is bound to the concrete merged automaton. On the left side of the figure is the specified application merge, with the bi-colored states representing the translation of parameters between actions. On the right side is the concrete merged k-colored automaton, where the action transitions

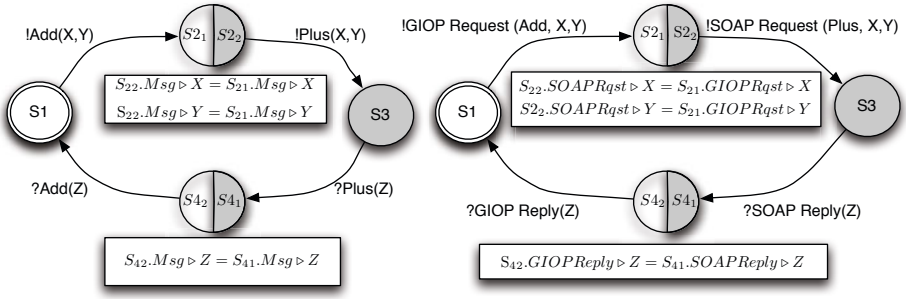


Fig. 8. Construct a concrete merged application automaton

are bound to specific middleware protocols (the operation name difference is overcome by this, after an Add action is received a Plus action is sent). Note, the application translations are bound to the specific MTL translations based upon the binding rules specified in Section 4.3.

5 Evaluation

To evaluate our approach for overcoming combined application and middleware heterogeneity, we use a case-study based methodology. That is, we apply Starlink to particular use cases and observe the extent to which interoperability is achieved. For this purpose we consider the application scenario described in Section 2. This application performs search and display of public photographs and requires interoperation between independently developed XML-RPC and SOAP Flickr clients and the Picasa Rest implemented API. We hypothesize the following:

1. The Starlink models can specify the application differences between Flickr and Picasa independent of SOAP, XML-RPC and HTTP messages.
2. Concrete models for both the XML-RPC and SOAP use cases can be successfully generated, deployed and executed to achieve the required interoperability with the Picasa API.
3. The use of high-level specifications simplifies the development of mediators and resolves evolution problems.

5.1 Flickr-Picasa Case-Study

In this case study we develop and deploy two mediators: a Flickr-Picasa mediator for XML-RPC to Rest, and a Flickr-Picasa mediator for SOAP to Rest. In the first instance we specify the application automata describing the API usage of both the client and service, as shown Fig. 2. Although automata are written using the XML-based Starlink language for k-colored automata, we use visual representations for clarity. Subsequently, we specified: i) the intertwined API

usage automaton as shown in Fig. 3, ii) the SOAP protocol models consisting of the MDL and the k-colored protocol automaton, iii) the XML-RPC models, and iv) the Rest models.

The next step consists of generating the application-middleware mediators by binding the single intertwined automaton to the two particular use cases. We now present in the remainder of this section the results of this binding. For sake of clarity, we only describe subsets of it to highlight key results.

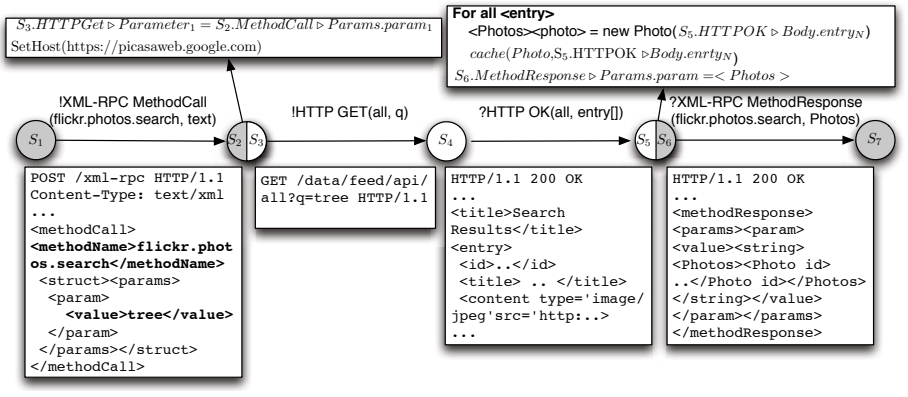


Fig. 9. XML-RPC to Rest binding for Search operation

An extract of the binding of the intertwined Flickr and Picasa search operations to an 'XML-RPC to Rest' concrete mediator is shown in Fig. 9. It illustrates how the XML-RPC Flickr message is parsed to extract the application information from transition S_1 to S_2 (e.g. the action label `flickr.photos.search` and the data parameter labelled `text`). The MTL for S_2 to S_3 then describes how the fields are translated before constructing the HTTP message to perform a Picasa search. The subsequent translation of the responses, from state S_5 to S_6, highlights a case where further functionality is required. The values that must be returned to the Flickr search operation is a list of Flickr photo identifiers in the format <`photo id id='1111' owner='1111111@NO1'`>. To handle this mapping, the bridge creates a cache of dummy identifiers for each photo result returned in the Picasa action response (the <`entry`><`id`> value from the XML data). The MTL provides a keyword operation `cache` that caches data values for arbitrary data identifiers.

An example of operation mismatch in the intertwined automaton is illustrated in Fig. 10. Indeed, when the Flickr client sends a `getInfo` action request, there is no corresponding operation in Picasa because the required action result data has already been received in Picasa's search response. Hence, when the `getInfo` XML-RPC message is received at S_8 then a data translation is performed:

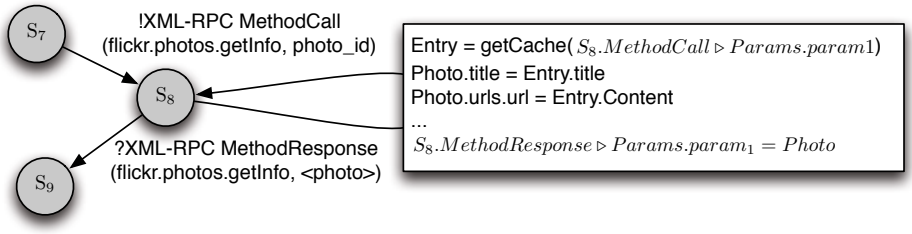


Fig. 10. MTL translation for Flickr-Picasa mismatch behaviour

the *photo_id* parameter is used to extract the Picasa <`entry`> value from the cache using the `getcache` MTL keyword. The Flickr <`photo`> structure is then filled using the corresponding tags from the Picasa <`entry`> structure.

The binding of the intertwined Flickr and Picasa comment operations to a 'SOAP to Rest' concrete mediator is similar to the XML-RPC binding and uses the rules provided in section 4. In this case, the generated MTL and k-colored automata refer to SOAP message content rather than XML-RPC.

Finally, we hand developed two test standalone client applications in SOAP and XML-RPC that searched and displayed photographs from the Flickr API. We then deployed Starlink in the network and loaded the concrete models. When executed, both clients were able to search and view photographs from the Picasa API. For our experiments, we deployed a simple proxy to redirect the Flickr requests (originally directed to the Flickr servers) to the local Starlink mediator.

5.2 Analysis

The automaton that specifies the application model contains no reference to a concrete protocol, message format or network semantics. As a consequence, it is seen that the first hypothesis that application behaviour can be modelled independent of middleware is successfully achieved. The generated concrete mediators, when deployed in the network, successfully parse and compose middleware messages and bridge the heterogeneous application behaviour in both the XML-RPC and SOAP case. Hence, the hypothesis that such code can be generated for multiple specific protocols is shown to be true. Finally, it can be argued that the definition of a single application model simplifies the development of interoperability solutions. There is no need to hand code each use case, and it is similarly straightforward to handle API migrations or changes using only the models.

6 Related Work

Middleware solutions to interoperability generally focus on bridging the gap between the various middleware technologies involved. These assume a common application *standard*, i.e., that applications wishing to interoperate use the same interface defined in the same language (e.g. Interface Description Language (IDL)

or Web Services Description Language (WSDL)). In this situation the interoperability gap is between heterogeneous middleware protocols. Protocol bridges [1], Enterprise Service Buses [12,11], and Interoperability Frameworks e.g. WSIF [6], uMiddle [16], OSDA [15], and UIC [19] are well known solutions to this problem. However, because they do not consider heterogeneity at the application-level they are not suited to the composition of complex systems-of-systems; where independently developed applications are composed dynamically it is unlikely that the application interface has been agreed in advance.

Several technologies are available to manage the differences between application service interfaces in terms of operation and message sequences. As an example, Web Services orchestration and choreography methods [17] provide languages such as Web Services-Choreography Description Language (WS-CDL) and Business Process Execution Language (BPEL) to handle such translations. These languages are similar to Starlink in that they provide high-level constructs to mediate behaviour sequences and also perform data translations. However, they assume an underlying platform (e.g. Web Services) and focus on choreography rather than on direct interoperability. As a consequence, differences in underlying protocols cannot be handled. Furthermore, they cannot manage the differences in interface languages. For example, BPEL cannot be used to generate a solution to make a CORBA IDL-based client interoperate with a SOAP WSDL-based service.

Model Driven Architecture (MDA) [8] proposes a similar methodology to Starlink, which indeed is inspired by the modelling ideas put forward by MDA. Application systems are specified using an abstract model, called the Process Independent Model (PIM). The PIM is deployed atop middleware based platforms described by the Platform Specific Model (PSM). Bridges are then deployed where there are exchanges between different PSMs to ensure that the platform heterogeneity is resolved. However, MDA is characterised by ad-hoc solutions with limited support for the generation of bridges between the platform models.

Formal specifications have been proposed to generate mediators between heterogeneous systems. Yellin and Strom [22] describe a method to enhance application interfaces with sequencing constraints in addition to rules that describe how the application data can be bridged. This information is then used to generate the code of the software adapters. Similarly, [14] describes a discrete event systems method for describing a converter between disparate protocols. While closely related to our formal models of protocol and translation, our approach further investigates the concrete realities of application differences based upon heterogeneous middleware paradigms and message formats.

Currently Starlink developers construct the merged automata; however, emerging solutions have investigated how to generate the mediator automatically. [20] models protocols as labelled transition systems (LTS) and presents an algorithm to identify the merge of the two; however at present it considers only message sequence differences not data heterogeneity. Similarly, work in the CONTESSA project [9] presents a reflective approach to compose heterogneous protocol-based services. This utilises semantic models of the transitions between the

configurations of the protocols and services. While this doesn't cover the complete interoperability mappings that Starlink proposes it does offer important insights into how reasoning and composition can be performed automatically at runtime.

7 Conclusions

In this paper we have shown that the interoperability problem is characterised by differences in both application APIs and middleware protocols. Existing solutions have focused on one of these dimension while making assumptions about the common nature of the other. In complex and dynamic systems such assumptions are invalid, and new approaches are required to consider application and middleware together. For this purpose, we have presented Starlink[3] –a framework to model applications and protocols such that the code to interoperate can be generated. Specifically, this consists of application and application-middleware mediator models (specified using automata and domain specific languages) that are interpreted at runtime. We have performed evaluation of Starlink using a case study involving heterogeneous web-based services (i.e. photo sharing). Preliminary results show that Starlink can successfully address the interoperability challenges, and simplify the task of connecting disparate systems.

Starlink requires the developer to write models, however given the scale of heterogeneous applications and protocols automated generation of these models is the ultimate goal. Hence, it is necessary to *reason* about the individual application and protocol models and generate the merged automata between. To underpin this reasoning we believe that additional semantic models can be used to infer the translation logic and we are investigating the use of ontologies [5] and their associated tools. For full automation, *machine learning* is required and hence we are also investigating learning techniques to understand and model the behaviour of the individual protocols. For example, dynamic binary analysis approaches have been used to identify the field structure of network messages [4] and learning algorithms have been utilised to learn the interaction behaviour of application and middleware protocols [10].

Acknowledgments. Part-funded unded under the EU FP7 Connect project (http://connect-forever.eu).

References

1. Soap2corba and corba2soap, http://soap2corba.sourceforge.net/
2. Bromberg, Y.-D., Grace, P., Réveillère, L.: Starlink: runtime interoperability between heterogeneous middleware protocols. In: The 31st International Conference on Distributed Computing Systems, Minneapolis, MN, USA (June 2011)

[3] The framework is available to download at http://starlink.sourceforge.net

3. Bromberg, Y.-D., Issarny, V.: INDISS: Interoperable Discovery System for Networked Services. In: Alonso, G. (ed.) Middleware 2005. LNCS, vol. 3790, pp. 164–183. Springer, Heidelberg (2005)
4. Caballero, J., Yin, H., Liang, Z., Song, D.: Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS 2007, pp. 317–329. ACM, New York (2007)
5. Daconta, M., Obrst, L., Smith, K.: The Semantic Web: A Guide to the Future of XML, Web Services and Knowledge Management. Wiley, Indianapolis (2003)
6. Duftler, M., Mukhi, N., Slominski, S., Weerawarana, S.: Web services invocation framework (wsif). In: OOPSLA Workshop on Object Oriented Web Services (2001)
7. Grace, P., Blair, G., Samuel, S.: A reflective framework for discovery and interaction in heterogeneous mobile environments. ACM SIGMOBILE Mobile Computing and Communications Review 9(1), 2–14 (2005)
8. Object Management Group. Model driven architecture (mda), document number ormsc/2001-07-01. Technical report (2001)
9. Gutierrez-Nolasco, S., Venkatasubramanian, N.: A Reflective Middleware Framework for Communication in Dynamic Environments. In: Meersman, R., et al. (eds.) CoopIS 2002, DOA 2002, and ODBASE 2002. LNCS, vol. 2519, pp. 791–808. Springer, Heidelberg (2002)
10. Howar, F., Jonsson, B., Merten, M., Steffen, B., Cassel, S.: On Handling Data in Automata Learning - Considerations from the Connect Perspective. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 221–235. Springer, Heidelberg (2010)
11. IBM. Websphere message broker, www.ibm.com/websphere/wbimessagebroker
12. IONA. Artix esb (2007), http://www.iona.com/products/artix/
13. Kongdenfha, W., Motahari-Nezhad, H., Benatallah, B., Casati, F., Saint-Paul, R.: Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. IEEE Trans. Serv. Comput. 2, 94–107 (2009)
14. Kumar, R., Nelvagal, S., Marcus, S.: A discrete event systems approach for protocol conversion. Discrete Event Dynamic Systems 7, 295–315 (1997)
15. Limam, N., Ziembicki, J., Ahmed, R., Iraqi, Y., Li, D., Boutaba, R., Cuervo, F.: Osda: Open service discovery architecture for efficient cross-domain service provisioning. Computer Communications 30(3), 546–563 (2007)
16. Nakazawa, J., Tokuda, H., Edwards, W., Ramachandran, U.: A bridging framework for universal interoperability in pervasive systems. In: 26th IEEE International Conference on Distributed Computing Systems, ICDCS 2006 (2006)
17. Peltz, C.: Web services orchestration and choreography. IEEE Computer 36(10), 46–52 (2003)
18. Ramdhany, R., Grace, P., Coulson, G., Hutchison, D.: MANETKit: Supporting the Dynamic Deployment and Reconfiguration of Ad-Hoc Routing Protocols. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 1–20. Springer, Heidelberg (2009)
19. Roman, M., Kon, F., Campbell, R.: Reflective middleware: From your desk to your hand. IEEE Distributed Systems Online 2(5) (August 2001)
20. Spalazzese, R., Inverardi, P., Issarny, V.: Towards a formalization of mediating connectors for on the fly interoperability. In: The IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (2010)
21. Vinoski, S.: It's just a mapping problem [computer application adaptation]. IEEE Internet Computing 7(3), 88–90 (2003)
22. Yellin, D., Strom, R.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. 19, 292–333 (1997)

The Role of Ontologies in Emergent Middleware: Supporting Interoperability in Complex Distributed Systems

Gordon S. Blair[1], Amel Bennaceur[2], Nikolaos Georgantas[2], Paul Grace[1], Valérie Issarny[2], Vatsala Nundloll[1], and Massimo Paolucci[3]

[1] School of Computing and Communications, Lancaster University, UK
[2] INRIA, CRI Paris-Rocquencourt, France
[3] DOCOMO Euro-Labs, Munich, Germany
gordon@comp.lancs.ac.uk

Abstract. Interoperability is a fundamental problem in distributed systems, and an increasingly difficult problem given the level of heterogeneity and dynamism exhibited by contemporary systems. While progress has been made, we argue that complexity is now at a level such that existing approaches are inadequate and that a major re-think is required to identify principles and associated techniques to achieve this central property of distributed systems. In this paper, we postulate that emergent middleware is the right way forward; emergent middleware is a dynamically generated distributed system infrastructure for the current operating environment and context. In particular, we focus on the key role of ontologies in supporting this process and in providing underlying meaning and associated reasoning capabilities to allow the right run-time choices to be made. The paper presents the CONNECT middleware architecture as an example of emergent middleware and highlights the role of ontologies as a cross-cutting concern throughout this architecture. Two experiments are described as initial evidence of the potential role of ontologies in middleware. Important remaining challenges are also documented.

Keywords: interoperability, ontologies, emergent middleware, system-of- systems.

1 Introduction

Interoperability is a fundamental property in distributed systems, referring to the ability for two or more systems, potentially developed by different manufacturers, to work together, including the ability to exchange and interpret action requests and associated data sets. Indeed, interoperability is absolutely foundational—without a solution to interoperability, distributed systems become impossible to develop and evolve. In the first generation of distributed systems, interoperability was relatively straightforward to achieve. Such systems were small-scale, fairly homogenous in terms of languages, operating system platforms and hardware architectures, and also under the control of a single organisation and associated administration team. This

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 410–430, 2011.

was of course unsustainable and very quickly distributed systems expanded in terms of scale, level of heterogeneity and complexity of administrative control, leading to the Internet-scale distributed systems that we are familiar with today. A number of interoperability solutions emerged both in terms of proposed standards for interoperability and solutions to bridging between standards. Distributed systems have, however, continued to evolve, and we particularly note two important trends:

1. The level of heterogeneity has increased dramatically in recent years with developments such as ubiquitous computing potentially coupled with enhanced modes of interaction (for example using ad hoc networking), mobile computing where an increasing range of mobile devices provide a window on to greater distributed system services, and cloud computing where complex distributed system services are offered in the greater Internet. We refer to this as extreme heterogeneity, whereby the levels of heterogeneity significantly exceed the previous generation of distributed systems in terms of the size and capabilities of end system devices, the operating systems used by different devices, the style of communication protocols employed to provide network-level interoperability, the languages and indeed programming paradigms utilized, and so on. Some observers refer to such systems as Systems of Systems [1], and this certainly captures rather elegantly the complexity of the resultant system structures.
2. The level of dynamism in such systems has also increased significantly, partly as a result of the trends noted above, for example the increasing mobility involved in distributed systems has led to the need to support spontaneous interoperation whereby devices interoperate with services that are discovered in a given location, coupled with solutions that need to be intrinsically context-aware (including of course location-aware access to services). The level of dynamism is also affected by the need for more adaptive and/ or autonomic approaches, again stemming from the complexity of modern distributed systems.

The end result is that it is very difficult to achieve interoperability in such complex distributed systems. Indeed, we can say that distributed systems are in crisis with no principled solutions to interoperability for such complex and dynamic distributed systems structures. Note that we can go further in this analysis and not just consider the ability to interoperate but also the quality of service of interoperability solutions in terms of a range of non-functional properties, for example related to security or dependability. This is a very valid dimension to consider but is beyond the scope of this paper (we return to this in the final section, and in particular our statements on future work).

It is interesting to note the definition of interoperability from Tanenbaum [2]:

> *"The extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard"*

This definition emphasizes the role of a global, or at least common, standard and, while this offers one solution to interoperability, it is not a realistic option for the complex distributed systems of today. For example, competitive pressures have inevitably led to competing standards emerging in the marketplace. Where standards have reached a level of acceptance, for example with web services, it is recognized by

the community that they may be problematic for certain operating environments, for example, ubiquitous systems. In addition, any given standard can very quickly become a legacy system as time elapses and requirements evolve.

We argue that with the above pressures we need a fundamental re-think of distributed systems. In particular, we advocate a solution whereby the necessary middleware to achieve interoperability is not a static entity but rather is generated dynamically as required by the current context. We refer to this as *emergent middleware*. Furthermore, we investigate the key role of ontologies in supporting this process and, in particular, in providing the ability to interpret meaning and associated reasoning capabilities in generating emergent middleware. Ontologies have already been studied in the context of distributed systems, most prominently in the semantic web community, offering a means of interpreting the meaning of data or associated services as they are dynamically encountered in the World-Wide Web. This however limits the scope of ontologies to support the top-level access to data and services. We are interested in a more comprehensive role for ontologies in supporting meaning and reasoning in the distributed systems substrate which supports and enables access to such services, i.e., in the middleware itself, offering a cross-cutting approach where ontologies provide support to fundamental distributed systems engineering elements.

This paper focuses exclusively on the role of ontologies in supporting the concept of emergent middleware (further discussion of the broader area of emergent middleware can be found in [3]). More specifically, the aims of the paper are:

1. To investigate previous work on interoperability in the middleware community and in the semantic web community with a view to seeking a unification between these (to date) largely distinct areas of research;
2. To understand both the role and scope of ontologies in supporting key middleware functions, particularly related to emergent middleware solutions;
3. To investigate more generally the role of ontologies within a general architecture for emergent middleware.

The paper is structured as follows. Section 2 examines the interoperability-related challenges associated with complex distributed systems and the associated responses both from the middleware and the semantic web community. Section 3 moves into the solution space, presenting the key components of an emergent middleware approach, before charting the role of ontologies within this approach. Section 4 presents two experiments, which together provide evidence of the key role ontologies can play in different levels of a middleware architecture. Finally, Section 5 contains an overall analysis and reflections over the experience of working with ontologies in emergent middleware, including the identification of key areas of future work related to this area.

2 The Interoperability Problem Space: Challenges and Responses

The problem space for interoperability must consider the differences of: i) *applications*, and ii) *middleware protocols*. In both cases, there will typically be differences in *data* and *behaviour*:

- Application *data* differs in terms of format and meaning, e.g., the data value of a `price` parameter can be defined in an object or XML document. It can also mean different things, e.g., the price is in Pounds versus Euros.
- Depending upon application interfaces, the *behaviour* may be significantly different, e.g., multiple operations of one interface performing the same functionality of a single operation of another.
- Middleware protocols providing the same communication abstraction may differ in the *data* format and type model, e.g., different RPC protocols capture data and types using different methods and formats.
- There now exists a broad range of communication abstractions (e.g., publish-subscribe, tuple spaces, message-orientation, group communication) offered by middleware protocols; these exhibit significant *behavioural* differences.

We now examine the responses to these challenges from two distinct communities (the middleware and the semantic web communities) and investigate the extent to which comprehensive application and middleware interoperability has been achieved.

2.1 Response from the Middleware Community

The first responses by the middleware community to address interoperability problems proposed *standards*-based approaches, i.e., common protocols and interface description languages. CORBA, DCOM, and web services are effective examples of this approach. However, as previously described, such solutions are not suited to today's highly complex distributed systems that exhibit extreme heterogeneity and dynamic behaviour. The second set of responses then looked at the challenges of heterogeneous middleware protocols interoperating with one another. One example of this, *software bridge*, acts as a one-to-one mapping between domains; taking messages from a client in one format and then marshalling this to the format of the server middleware. As examples, the OMG created the DCOM/CORBA Inter-working Specification [6]. OrbixCOMet is an implementation of the DCOM-CORBA bridge, while SOAP2CORBA[1] bridges SOAP and CORBA middleware. Further, Model Driven Architecture advocates the generation of such bridges to underpin deployed interoperable solutions. However, developing bridges is a resource intensive, time-consuming task, which for universal interoperability would be required for every protocol pair.

Alternatively, *intermediary-based* solutions take the ideas of software bridges further; rather than a one-to-one mapping, the protocol or data is translated to an intermediary representation at the source and then translated to the legacy format at the destination. Enterprise Service Buses (ESB), INDISS [8], uMiddle [9] and SeDIM [10] are examples that follow this philosophy, and these allow differences of both behaviour and data to be overcome. However, this approach suffers from the greatest common divisor problem, i.e., between two protocols the intermediary is where their behaviour matches, they cannot interoperate beyond this defined subset. As the number of protocols grows, this common divisor then becomes smaller, such that only limited interoperability is possible.

[1] `http://soap2corba.sourceforge.net/`

A radically different response involved *substitution solutions* (e.g., ReMMoC [11] and WSIF [12]); rather than bridging, these embrace the philosophy of speaking the peer's language. That is, they substitute the communication middleware to be the same as the peer or server they wish to use. A local abstraction maps the behaviour onto the substituted middleware. This approach allows interoperation among different abstractions and protocols. However, as with software bridges this is particularly resource consuming; every potential (and future) middleware must be developed such that it can be substituted. Further, it is generally limited to client-side interoperability with heterogeneous servers.

The limitation of all the above responses is that they ignore the heterogeneity of the application, assuming that there are no differences, due to the adoption of a common interface. In complex systems, this is clearly not the case.

2.2 Response from the Semantic Web Community

The semantic web community's responses to the interoperability problem are based upon the principles of reasoning about and understanding how different systems can work together. Their key contribution is *ontologies*. An ontology is defined as a logic theory, and more precisely as a tuple <A, L, P>, where A is a set of axioms, L is a language in which to express these axioms, and P is a proof theory, that supports the automatic derivation of consequences from the axioms. In turn, the proof theory P allows us to derive consequences, which extract relations that have never been stated explicitly, but that are implicit in the description of the systems. Ultimately, the proof theory allows recognition of the deeper "semantic" similarity between structures that are syntactically very different.

The work in *semantic web services* demonstrates how ontologies can be used to address interoperability problems at the application level. Specifically, ontologies have been used during discovery to express the capabilities of services, as well as the requests for capabilities; in this case, the proof theory recognizes whether a given capability fits a given request. A number of semantic middleware technologies provide this ability, e.g., the Task Computing project [13], and the Integrated Global Pervasive Computing Framework [14]. One important solution, EASY [15], implements efficient, semantic discovery and matching to foster interoperability in pervasive networking environments. Further, ontologies have been used during composition to address the problem of application data interoperability, as well as the problem of recognizing whether the conditions for executing the service indeed hold. The limitation of these responses lies in the assumption of a specific middleware, namely web services. There is a need to represent heterogeneous middleware and networking environments, which is almost completely absent in the semantic web services work.

Ontologies introduce a new meta-level, which can produce its own interoperability problems. Heterogeneous ontologies push the interoperation problem one level up. The computational complexity of the proof theories, which is often beyond exponential, makes ontologies resource expensive. Finally, there is a problem of generating the ontologies. The problems listed here are fundamental problems with which the semantic web at large is grappling, and fortunately a number of partial

solutions exist that mitigate these problems. For example, ontology matching can be used to address the problem of different ontologies, and smart and efficient inference engines are now available. As a result, ontologies may be used effectively to automatically address many interoperability problems.

2.3 Summary

It is clear that semantic technologies and interoperability middleware have mostly been developed in isolation by distinct communities. The middleware community made assumptions of common application interfaces and focused on middleware behaviour and data heterogeneity. The semantic web community made the opposite assumption, that there was a common middleware, and the solutions focused on differences in application behaviour and data.

In our view, semantic technologies and interoperability middleware must be comprehensively combined to enable emergent middleware, that is, on-the-fly generation of the middleware that allows networked systems to coordinate to achieve a given goal. Semantic technologies bring the necessary means to rigorously and systematically formalize, analyze and reason about the behaviour of digital systems. Semantic web service technologies have further highlighted the key role of process mediation in an Internet-scale open network environment where business processes get composed out of services developed by a multitude of independent stakeholders. Then, in a complementary way, interoperability middleware solutions hint towards an architecture of emergent middleware that mediates interaction among networked systems that semantically match while possibly behaviourally mismatching, from the application down to the network layer.

3 The Solution Space

The realisation of emergent middleware faces significant challenges, which we are in particular investigating as part of the CONNECT project [3]: i) discovering what is there in terms of application and middleware behaviour and data, ii) enhancing this information using learning techniques, and iii) reasoning upon the required mediation and synthesizing the resulting software to enable interoperability between heterogeneous networked systems. In this section, we first introduce the architecture of the generated emergent middleware, and then we present the ontology-based models of the networked systems used by *Enablers*, i.e., active software entities that collaborate to realise the emergent middleware. Finally, we describe the architecture of Enablers that need to be deployed in the network toward allowing networked systems to interact seamlessly.

3.1 Architecture of Emergent Middleware

Building upon previous interoperability middleware solutions [8, 10, 16], the architecture of the emergent middleware (as shown in Fig. 1) decomposes into: (i) *message interoperability* that is dedicated to the interpretation of messages

from/toward networked systems (listeners parse messages and actuators compose messages) and (ii) *behavioural interoperability* that mediates the interaction protocols run by the communicating networked systems by translating messages from one protocol to the other, from the application down to the middleware and further to the network layer.

However, interoperability can only be achieved based on the unambiguous specification of networked systems' behaviour, while not assuming any a priori design-time knowledge about the given systems. This is where the key role of semantic technologies, i.e., ontologies, comes into play. As discussed in the next section, ontological concepts are employed to characterise the semantics of exchanged messages, from the application down to the network layer, and thus allow the analysis of and reasoning about the external actions performed by systems. This is a major step in the realization of interoperability, since it allows the mediation of interaction protocols at all layers, provided their respective functionalities semantically match.

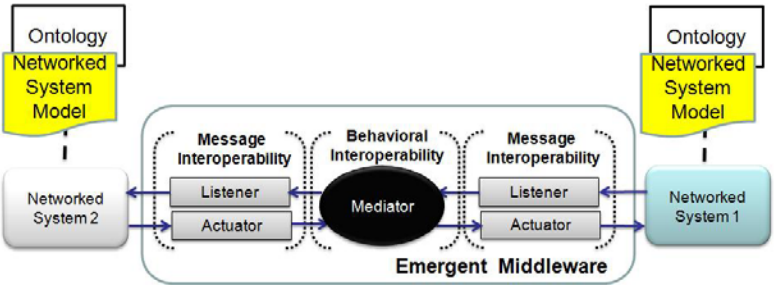


Fig. 1. The emergent middleware architecture

3.2 Ontology-Based Networked System Model

The *networked system model* builds upon semantic technologies and especially semantic web services ontologies [17]. Fig. 2 depicts key elements of the system model with ontologies cross-cutting these elements. The model decomposes into:

- The *Affordances* (*aka capabilities* in OWL-S[2]) provide a macroscopic view of networked system features. An affordance is specified using ontology concepts defining the semantics of its *functionality* and of the associated *inputs* and *outputs*. Essentially, the affordance describes the high-level roles a networked system plays, e.g., 'prints a document'. This allows semantically equivalent action-relationships/interactions with another networked system to be matched; in short, they are doing the same thing. Then, provided the matching of affordances that are respectively required and provided by two networked systems, it should be possible to synthesize an emergent middleware that allows the networked systems to coordinate toward the realization of the affordance despite possible mismatches in the messages they exchange and even their behaviour. In practice,

[2] http://www.w3.org/Submission/OWL-S/

networked systems do not advertise affordances but rather interfaces, as discussed below. Nevertheless, recent advances on learning techniques, combining solutions to the cohesion of system interfaces [18] and semantic knowledge inference [19], provide base ground that can be exploited to support the automated inference of affordances from interfaces, although this remains area for future work.

- *The Interface* provides a refined or microscopic view of the system by specifying finer actions or methods that can be performed by/on the networked system, and used to implement its affordances. Each networked system is associated with a unique interface. However, there exist many interface definition languages and actually as many languages as middleware solutions. Nevertheless, existing languages may easily be translated into a common IDL so as to allow the matchmaking of interfaces [20]. Still, a major requirement and challenge are for interfaces to be annotated with ontology concepts so that the semantics of embedded actions can be reasoned upon. While this is already promoted by web services standards (e.g., SA-WSDL[3]), it still remains an exception for middleware solutions at large. Here too, research on advanced learning techniques can lead to automated solutions to the semantic annotation of syntactic interfaces [22].
- The *Behaviour* describes how the actions of the interface are co-ordinated to achieve a system's affordance, and in particular how these are related to the underlying middleware functions. The language used to specify the behaviour of networked systems revolves around process algebra enriched with ontology knowledge, so as to allow reasoning about their behavioural matching based on the semantics of their actions, and subsequently support the generation of the emergent middleware. Such behaviour description has been acknowledged as a fundamental element of system composition in open networks in the context of the Web[4]. However, in the vast majority of cases, networked systems do not advertise their behaviour. On the positive side, different techniques have emerged to learn the interaction behaviour of systems, either reactively or proactively [23, 24, 33]. Still, major research challenges remain in the area, as provided techniques need to be made more efficient as well as be improved, considering, e.g., the handling of data and non-functional properties.

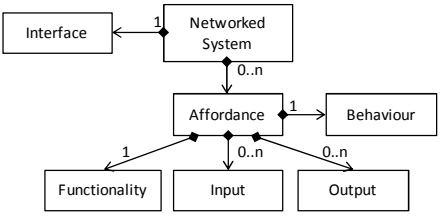


Fig. 2. The networked system model

[3] http://www.w3.org/TR/sawsdl/
[4] www.w3.org/TR/wscl10/

3.3 Enablers for Emergent Middleware

The realization of emergent middleware is supported by cooperating core Enablers as depicted in Fig. 3.

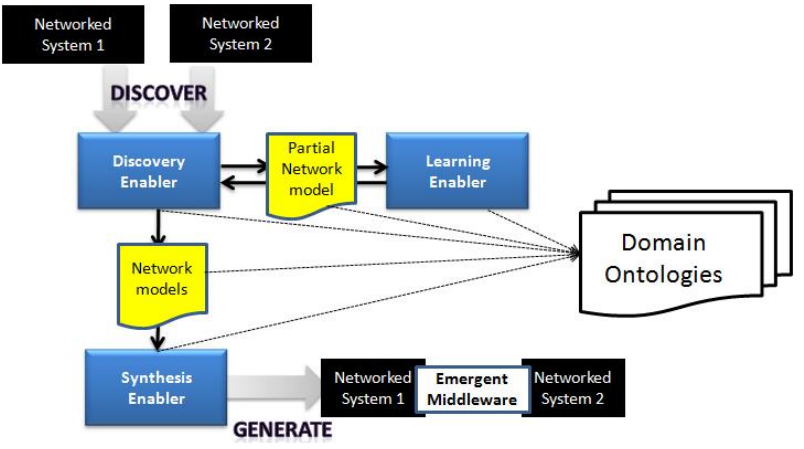


Fig. 3. The architecture of the emergent middleware Enablers

The *Discovery Enabler* receives both the advertisement messages and lookup request messages that are sent within the network environment by the networked systems. The enabler obtains this input by listening on known multicast addresses (used by legacy discovery protocols), as common in interoperable service discovery [25]. These messages are then processed; information from the legacy messages is extracted. At this stage, the networked system model includes at least the interface description, which can be used to infer the ontology concepts associated to the affordance in the case they are not specified. The semantic matching of affordances is then performed to determine whether two networked systems are candidates to have an emergent middleware generated between them. The semantic matching of affordances is based on the *subsumption* relationship possibly holding between concepts of the compared affordances [26]; briefly, the functionality of a required affordance matches a provided one if the former is subsumed by the latter. Other semantic relations such as sequence [29] or part-whole[5] can also be beneficial to concept matching. On a match, the process of emergent middleware generation is started; the current networked system model is sent to the Learning Enabler, which adds more semantic knowledge to it. On completion of the model, the Discovery Enabler sends this to the Synthesis Enabler.

More specifically, the *Learning Enabler* attaches semantic annotations to the interface, and uses active learning algorithms to dynamically determine the interaction behaviour associated to an affordance. Interaction behaviour learning is built upon the

[5] `http://www.w3.org/2001/sw/BestPractices/OEP/SimplePartWhole/index.html`

LearnLib tool [27], and employs methods based on monitoring and model-based testing of the networked systems. It takes the semantic annotations of the interface as input, and returns the system's behaviour description.

The role of the *Synthesis Enabler* is to take the completed networked system models of two systems and then synthesize the emergent middleware that enables the networked systems to coordinate on a given affordance. The emergent middleware specifically implements the needed mediation between the protocols run by the systems to realize the affordance, which are abstractly characterized by the behavioural description. The synthesis of the mediator results from the automated behavioural matching of the two protocols based on the ontological semantics of their actions. In few words, the mediator defines the possible sequences of actions that serve translating semantic actions of one protocol to semantic actions of the other. Obviously, many approaches to behavioural matching and related protocol mediation may be applied considering the state of the art in the area [30, 31]. Basically, the solution to automated protocol mediation shall allow for efficient mediator synthesis, while at the same time enabling interoperability beyond current interoperability middleware solutions. In particular, protocol mediation shall span all the targeted protocol layers, dealing with the semantics of both application and middleware actions [28], as illustrated in the next section. An approach that is particularly promising and that we are investigating lies in ontology-based model checking [32]; this exploits the power of both ontologies to systematically reason about the semantics of actions and model checking to systematically reason about the compatibility of protocols. Still, the more flexible is the compatibility check, the more complex is the reasoning process. The challenge is then to find the appropriate tradeoffs so as to foster interoperability in open networks in a computationally tractable way.

Finally, the *emergent middleware* is deployed, with the resultant connector following the architecture as depicted in Fig. 1, with listeners and actuators providing message interoperability and the synthesized mediator dealing with behavioural differences and translating the message content between heterogeneous message fields. Note the listeners and actuators are automatically generated using the Starlink framework[6].

While this section has focused on the core Enablers toward the generation of emergent middleware, additional enablers are necessary to cope with the uncertainty associated with emergent middleware. Indeed, the learning phase is a continuous process where the knowledge about networked systems is being enriched over time, which implies that emergent middleware possibly needs to adapt as the knowledge evolves. Furthermore, it is important that emergent middleware respects the quality requirements of networked systems regarding their interactions, which requires appropriate dependability and security enablers.

The development, from the supporting theory to concrete prototype implementation, of such enablers is currently ongoing as part of the CONNECT EU project[7]. Despite the tremendous challenges that are raised in unifying and combining the principles of semantic technologies and interoperability middleware to enable emergent middleware, we have been developing experimental enablers to validate this vision. Our initial experiences with the use of ontologies within this broad solution

[6] http://starlink.sourceforge.net/
[7] http://connect-forever.eu/

space are sketched in the next section; these further highlight the important role ontologies have to play in realising our vision of emergent middleware.

4 Experiments

To provide initial insight into the benefits of using ontologies to support interoperability, we now present two experiments that show how semantic technologies can underpin the automatic generation of emergent middleware. The first experiment examines the use of ontologies to address data and behavioural heterogeneity at both application and middleware layers. The second experiment demonstrates how ontologies are used to perform automated matching of message fields to support interoperability at the network layer.

4.1 Reasoning about Interoperability at Application and Middleware Layers

This experiment illustrates the role of ontologies in handling heterogeneity both at application and middleware layers. For this purpose, we consider two travel agency systems that have heterogeneous application interfaces and are implemented using heterogeneous middleware protocols (one is implemented using SOAP and the other with HTTP REST). We use application-specific and middleware ontologies to reason about the matching of both application and middleware behaviour.

The travel agencies example. The first networked system, called *EUTravelAgency,* is developed as an RPC-SOAP web service. Thus, data is transmitted using SOAP request and response envelopes transported using HTTP Post messages. The service allows users to perform the following operations concurrently:

- *Selecting a flight*. The client must specify a destination, a departure and a return date. The service returns a list of eligible flights.
- *Selecting a hotel*. The client indicates the check-in and check-out dates. The service returns a list of rooms.
- *Selecting a car to rent*. The user indicates the period of rental and their preferred model of car. The service then proposes a list of cars.
- *Making a reservation.* Once the user has chosen a flight and/or a hotel room and/or a car, they confirm their reservation. The service returns an acknowledgment.

The interface signature for *EUTravelAgency* (abstracted from WSDL 2.0) is given below, where we provide only the ontology concepts associated with the syntactic terms embedded in the interface:

```
SelectFlight({destination, departureDate, returnDate}, flightList)
SelectHotel({checkIndate, checkOutdate, pref}, roomList)
SelectCar({dateFrom, dateTo, model}, carList)
MakeReservation({flightID, roomID, carID}, Ack)
```

The second system is called *USTravelAgency* and allows users to perform the following two operations:

- *Finding a trip*. The client specifies a destination, departure and return date. The service finds a list of "packages" including a flight and hotel room and car.
- *Making a reservation.* The user selects a trip package and confirms it. The service acknowledges the reception of the selection.

The interface signature, although giving only embedded ontology concepts, is abstracted as follows:

```
FindTrip({destination,departureDate,returnDate,needCar},flightList)
ConfirmTrip(tripID,Ack)
```

The *USTravelAgency* service is implemented as a REST web service over the HTTP protocol. The *findTrip* operation is performed as a HTTP Get and the *confirmTrip* operation is performed using a HTTP Post as shown below (the outputs of both service operations are formatted using JSON[8]):

```
GET http://ustravelagency.com/rest/tripervice/findTrip/{destination}/
{departureDate}/{returnDate}/{needCar}
POST http://ustravelagency.com/rest/tripervice/confirmTrip/{tripID}
```

A client of the *EUTravelAgency* cannot interact with the *USTravelAgency*, and similarly a client developed for the *USTravelAgency* cannot communicate with the *EUTravelAgency* due to the aforementioned heterogeneity dimensions:

- *Application data*. The *EUTravelAgency* refers to the *Flight*, *Hotel* and *Car* concepts, whereas the *USTravelAgency* makes use only of the *Trip* concept. Additionally, the *EUTravelAgency* specifies the departure and the return dates using Greenwich Mean Time (GMT), while the USTravelAgency uses Pacific Standard Time (PST) to describe them.
- *Application behaviour*. In the *EUTravelAgency* implementation, users can independently select a flight, a room and a car, whereas in the *USTravelAgency* implementation all of them are selected through a package.
- *Middleware data format*. The data exchanged in the *EUTravelAgency* implementation are encapsulated in a SOAP message, while the input data of the *USTravelAgency* are passed through a URL and the output data are formatted using JSON.
- *Middleware behaviour*: REST and RPC-SOAP are different architectural styles and induce heterogeneous control and communication models.

The travel agency ontology. The first step of the experiment of interoperability between *EUTravelAgency* and *USTravelAgency* was to create the domain-specific ontology associated with the travel agency scenario (Fig. 4 illustrates an excerpt of this ontology). The ontology shows the relations holding among the various concepts defined in the interfaces of the two travel agencies. Note that the application-specific

[8] http://www.json.org/

ontology not only describes the semantics and relationships related to data but also the semantics of the operations performed on data, such as `FindTrip`, `SelectFlight`, `SelectHotel`, and `SelectCar`.

In the general case, the application ontology is not defined by the application developers but by domain experts, to reflect shared knowledge about a specific domain. Many ontologies have been developed for specific domains, e.g., Sinica BOW[9] (Bilingual Ontological Wordnet) for English-Chinese integration. In addition, work on ontology alignment enables dealing with possible usage of distinct ontologies in the modelling of different networked systems from the same domain, as illustrated by the W3C Linking Open Data project[10].

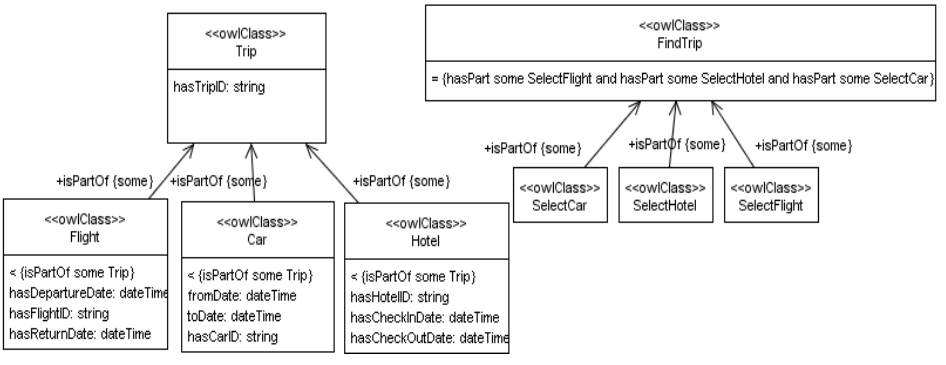


Fig. 4. The travel agency ontology

Dealing with application-level heterogeneity. The travel agency ontology indicates how the *Flight*, *Hotel* and *Car* concepts are related to the *Trip* concept, including their individual attributes. Moreover, we can also use standard ontologies for translation, e.g., OWL-Time[11] can be used to resolve the time difference between GMT and PST.

Solving the application data mismatches is not sufficient. We also need to coordinate the actions of the networked systems in order to make them interoperate. Ontologies help establishing the correspondence between actions. As illustrated in Fig. 4, *FindTrip* is defined as equivalent to the composition of the three operations *SelectFlight, SelectHotel,* and *SelectCar*. A mediator that ensures the coordination between the above operations can then be synthesized based on the semantic (subsumption) relations between them and the behaviour of the two networked systems. Moreover, since the *SelectFlight*, *SelectHotel*, and *SelectCar* can be executed concurrently, we need to check all possible executions. Therefore, we rely on model checking further extended with ontology reasoning capabilities, in order to exhaustively explore all the state space and systematically guarantee the correctness of the synthesized mapping rules. As illustrated in Fig. 5, the mediator translates the

[9] `http://BOW.sinica.edu.tw/`
[10] `http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData`
[11] `http://www.w3.org/TR/owl-time/`

FindTrip action to the concurrent execution of the *SelectFlight, SelectHotel* and *SelectCar* actions, and the *MakeReservation* action to the *ConfirmTrip* action. This translation is further refined according to the underlying middleware of each networked system as illustrated next.

Dealing with middleware-level heterogeneity. To reason about the behavioural matching of middleware, we have defined a middleware ontology that identifies where sequences of protocol messages execute similar functionality. For example, the request-response message sequence of CORBA is clearly equivalent to that of SOAP. Yet, there may be cases where the relationship is semantically deeper, e.g., subscription in a publish-subscribe protocol may be equivalent to a RPC invocation (but only when they are performing similar application behaviour) [28].

In the travel agency scenario, the operations are implemented atop SOAP and HTTP REST. The ontology specifies SOAP as a request followed by a synchronous response. Similarly, REST is specified as four alternative synchronous message sends and responses (Get, Post, Put, Delete). The ontology defines that a SOAP operation in the general case is semantically equivalent to all four REST behaviours. Therefore, to reason about interoperability, the application matching must be considered in tandem. For example, in the *FindTrip* operation case, the protocol mediation is from SOAP to Get, whereas in the *ConfirmTrip* case the protocol mediation is from SOAP to Post.

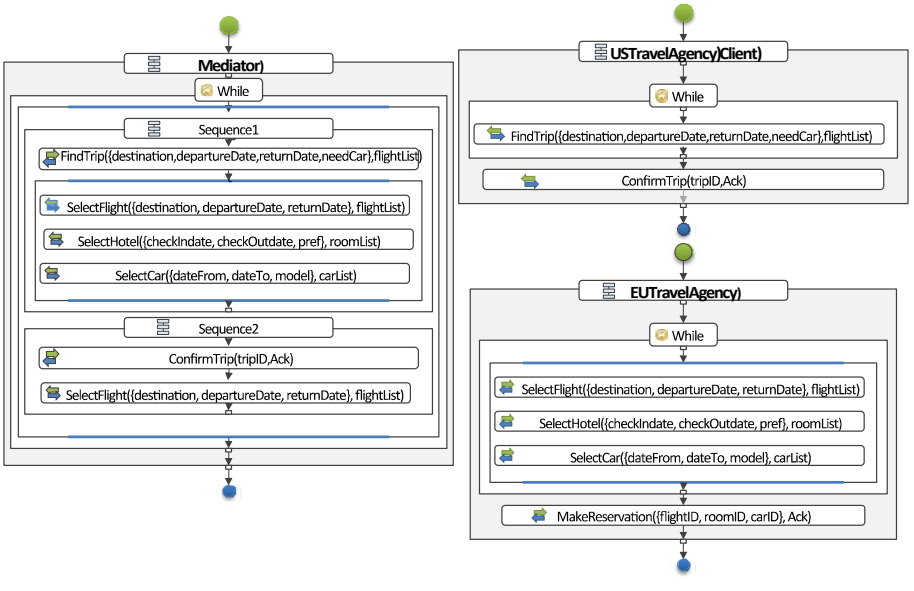


Fig. 5. Behavioural specification of the two travel agencies and the mediator

Another fundamental difference at the middleware level is in the heterogeneity of messages, i.e., the complexity of the translation of SOAP data content to REST data content while message formats are different. We investigate the use of ontologies to reason about this important problem in the second experiment.

4.2 Reasoning about Interoperability at the Network Layer

Devising solutions at the application and middleware level to enable any two systems to interoperate does not suffice if they cannot properly exchange network messages. It is imperative to understand and reason about the heterogeneous message formats of protocols in such a way that message-level interoperability can be achieved on a broader scale. We need systematic ways to dynamically capture the underlying differences of network packets to then generate the mapping between them. Ontologies provide the methodology to identify these semantic similarities and differences in order to automatically identify the translation between messages.

This experiment focuses on using ontologies to map between heterogeneous Vehicular Ad Hoc Network (VANET) protocols; this domain was chosen because the protocol behaviour is common (i.e., routing of messages to a destination), but there is a high-level of heterogeneity at the packet level. A number of VANET protocols exist that fall into different routing strategies: broadcast, position-based forwarding, trajectory-based forwarding, restricted directional flooding, content-based forwarding, cluster-based forwarding, and opportunistic forwarding. Hence, these exhibit highly heterogeneous message formats owing to the vast array of routing strategies.

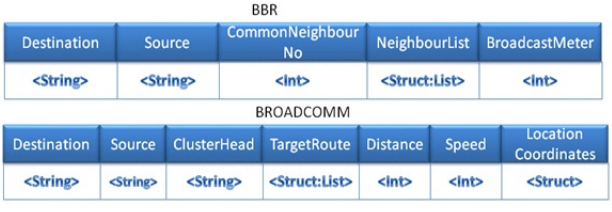


Fig. 6. Packet formats of BBR and Broadcomm packets

Interoperability between BBR and Broadcom. The BBR protocol [34] is a broadcast routing protocol that keeps track of neighbouring nodes and broadcasts the packet at a set rate. The node, lying on the border of the transmission range, is designated to forward the packet further away in the network. This is determined by the number of common neighbours this node has with the source node. This value is represented as a *CommonNeighbourNo* field in the packet. Fig. 6 shows the format of BBR and Broadcomm packets. Broadcomm [35] is a position-based routing protocol, which keeps track of nodes through their geographical locations. This protocol divides the network into clusters and allocates one node in each cluster to be the cluster head. The latter is responsible to forward messages to the cluster members and forward them to the nearest neighbour found outside the cluster. We can say that the behaviour of Broadcomm matches with that of BBR to a certain degree in the sense that both designate a node to disseminate messages further into the network. But both differ in the way their messages are formulated, especially with the use of geographical coordinates in one protocol and not in the other.

Applying Ontologies. Given the differences in their message formats, any sort of interoperation does not seem to be valid if Broadcomm and BBR try to interoperate.

However, if we can interpret both message formats and deduce their meaning, it is possible to find a basis for comparison. As a result, we create a vehicular ontology for the various routing strategies used in VANETs together with a definition of known packet formats. The main idea is to use this ontology to classify packet formats under the appropriate routing scheme and deduce how to enable this packet to interoperate with another packet, i.e., construct the mappings that are part of the synthesized mediator in the emergent middleware architecture. The presence of a reasoner engine enables us to infer the meaning of a packet (as we discover middleware knowledge of the networked system). As a result, the packet gets classified under the most appropriate routing strategy. This classification is an important step as it helps to establish a ground for comparison between packets belonging to different routing categories. Part of the inferred ontology is displayed in Fig. 7, where the BBR packet (BBRPacket) is ranked under *IdentifiedPacket* and *MFRBroadcastPacket* classes. The requirements for *MFRBroadcastPacket* are the fields: *CommonNeighbourNo* and *NeighbourList*. Since these fields form part of *BBRPacket*, the reasoner is able to classify the latter under *MFRBroadcastPacket*. The *IdentifiedPacket* class denotes that the packet contains known fields. In this way, incoming packets can be classified by the ontology and be compared with existing packet formats. For example, assume the incoming packet to be *Broadcomm* and the existing packet to be BBR.

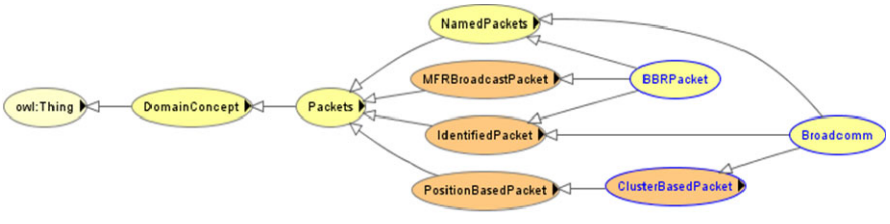


Fig. 7. Inferred Vehicular Ontology

Field Matching. Once both packets are classified, they can be compared to each other through an intuitive mechanism embedded in the ontology, which is the use of SWRL rules and SQWRL[12] query rules. These mechanisms add further reasoning to the classification process to enable field matching. As an example, the following SQWRL rule retrieves the fields from BBR and Broadcomm packets and tries to find the differences between them. To do so, it creates a collection of the fields of each packet using the SQWRL *makeBag* function and identifies the differences with the SQWRL *difference* function. The SQWRL clause is introduced within the SWRL rule by a separator character °. The SQWRL clause handles construction and manipulation operators required to execute SQWRL-based rules. As can be seen in the example below, the ° separator character enables a SWRL rule to include a SQWRL query.

[12] `http://protege.cim3.net/cgi-bin/wiki.pl?SWRLTab` and `…/wiki.pl?SQWRL`

BBRPacket(?b) ∧ hasFields(?b, ?f) ∧ Broadcomm(?p) ∧ hasFields(?p, ?pf) ˚ sqwrl:makeBag(?bag, ?f) ∧ sqwrl:makeBag(?bagt, ?pf) ˚ sqwrl:difference(?diff, ?bagt, ?bag) ∧ sqwrl:element(?e, ?diff) ⇒ sqwrl:selectDistinct(?p, ?e)

The result of this query gives the fields required for BBR to function as Broadcomm and vice versa; the fields lacking in BBR would be *LocationCoordinates*, *TargetRoute* and *ClusterHead*. Moreover, further classification is also possible through the use of SWRL rules to reason about the data types of the fields. As an example, suppose we have a field x in BBR packet of type <int> and a corresponding field y in Broadcomm of type <String>. In this case, we can make use of a SWRL rule to suggest a mapping between these two fields:

hasFields(BBR, ?x) ^ hasType(?x, <int>) ^ hasFields(Broadcomm, ?y) ^ hasType(?y,<String>) ⇒swrlb:MapIntToString(?x, ?y)

The OWL language enhanced with the use of SWRL and SQWRL enables comparison of two packets. The ontology can hence interpret the packet formats through matching and suggest a possible mapping between them. For example, the ontology can suggest that BBR lacks geographical coordinates in order to operate as Broadcomm. This information is fundamental in determining how to enable mapping between these two different types of packets. This is in itself a step forward towards interoperability between different network packets; however, further research is required into how ontologies can be used to generally identify mapping solutions that resolve the differences between packets. Further details about the use of ontologies within the domain of message-level heterogeneity are presented in [7].

5 Overall Reflections

Interoperability remains a fundamental problem for distributed systems due to the increasing level of heterogeneity and dynamism of the networking environment. In this paper, we have argued for a new approach to interoperability, i.e., *emergent middleware* that is synthesized on the fly according to the behaviour of the associated networked systems. A central element of our approach is the use of ontologies in the middleware design so that middleware may dynamically emerge based on semantic knowledge about the environment. Hence, while interoperability in the past has been about making concessions, e.g., pre-defined standards and design decisions, emergent middleware builds on the ability of machines to themselves reason about and tackle the heterogeneity they encounter. Further, acknowledging that interoperability is, as with many features of distributed systems, an end-to-end problem [5], emergent middleware emphasizes that interoperability can only be achieved through a coordinated approach involving application, middleware and network levels.

This paper has introduced the core elements of the emergent middleware vision, i.e., ontologies and related Enablers to reason about and implement interoperability on the fly. The architecture of Enablers outlined in Section 3 has provided a view of how

emergent middleware can be realised, where associated technologies becoming available through the CONNECT project. This architecture illustrates the important roles of discovery, learning and synthesis in achieving our goals. The most notable feature of the architecture is that ontologies have a cross-cutting role. The experimental work reported in Section 4 has further illustrated the central role of ontologies in supporting meaning and reasoning in distributed systems, not just at the application level but also in the underlying distributed systems substrate, for achieving interoperability in the highly heterogeneous and dynamic style of today's distributed systems. However, despite the latest advances in Enablers for emergent middleware, significant challenges remain ahead as discussed below.

While emergent middleware relieves the burden of interoperability from the middleware designers and developers, and fosters future-proof interoperability, its general applicability is dependent upon the effectiveness of the supporting Enablers. The latest results of CONNECT are encouraging in that they introduce base building blocks for the Enablers, spanning automated support for discovery, learning and synthesis. Small-scale experiments further demonstrate that Enablers may adequately be combined. Still, applicability to real-scale experiments is area for future work.

Realizing the central role of ontologies to allow machines to tackle interoperability across time raises the issue of how large, comprehensive ontologies may be deployed for interoperability in practice. At first sight, this basically depends on the development of supporting ontologies by domain experts and hence on the requirements of a given domain in terms of interoperability. For instance, it is expected that the Internet of Things will lead to major ontology development. Another consideration is the cost of processing large ontologies and, more specifically, the efficacy of semantic tools, which keep improving over time given research in the area. There is also considerable potential for core research on ontologies concerning the role of fuzziness in supporting richer forms of reasoning [21], the possibility of learning new ontological information and merging it with existing information as it becomes available, and also dealing with heterogeneity in the ontologies themselves.

We have so far concentrated on the synthesis of mediators from scratch, while the construction of mediators by composing existing ones would enable more efficient synthesis and support self-adaptive emergent middleware. Ongoing CONNECT research on an algebra for mediators will provide us the required foundations [4].

The inherent openness and probability of failure in emergent middleware solutions raise important challenges. If the solution is to be deployed at Internet scale, then it must be reliably able to produce correct mediators and also be secure against malicious threats. Hence, dependability is a central research question; this has to overcome the partial knowledge about systems as well as security concerns arising. A related concern is that of dealing with interoperability between fault tolerant systems and in general with the heterogeneity of non-functional properties across systems. Dedicated solutions are being investigated within CONNECT.

Furthermore, failing to generate emergent middleware in a specific context is not only dependent on the reliability of our solution, but also, most importantly in the open target environments, on the degree of incompatibility between candidate systems. For example, semantic matching may indicate that the semantic distance

between the application features of two systems is too great to be bridged. Precisely evaluating the limitations of our approach in producing a result is an area of future work; we are already studying aspects of this important issue within CONNECT.

Another interesting research direction for emergent middleware is that of involving end-users in the synthesis process to inform the automated approach. For example, end-users can assist semantic matching where ontology heterogeneity may lead automated reasoning to ambiguous results. This raises various challenges, including how to provide user-friendly interfaces to the emergent middleware internals.

In summary, this paper has argued that, given the increasing complexity of contemporary distributed systems, both in terms of increasing heterogeneity and dynamism, there is a need for a fundamental rethink of approaches to even the most basic of problems, that is, interoperability. We advocate a new approach to middleware, that of emergent middleware. This paper has looked at one key aspect of emergent middleware, namely, the role of ontologies in supporting core underlying middleware functions related to achieving interoperability. This leads to a fascinating set of research challenges both in terms of understanding a given deployment environment and also dynamically creating appropriate connectivity solutions. We hope this paper has given a flavour of the potential of this approach and also some real experimental evidence that the approach can work in selected aspects of distributed systems. As a final comment, while CONNECT is addressing a number of the ongoing challenges, this is a vast and largely unchartered territory and we invite other researchers to join in the quest for suitable solutions for emergent middleware.

Acknowledgements. This work is carried out in the CONNECT project, a European collaboration funded under the Framework 7 Future and Emerging Technologies Programme (Proactive Theme on ICT Forever Yours): http://www.connect-forever.eu.

References

1. Maier, M.W.: Architecting Principles for System of Systems. Systems Engineering 1(4), 267–284 (1998)
2. Van Steen, M., Tanenbaum, A.: Distributed Systems: Principles and Paradigms. Prentice-Hall (2001)
3. Bennaceur, A., Blair, G., Chauvel, F., Huang, G., Georgantas, N., Grace, P., Howar, F., Inverardi, P., Issarny, V., Paolucci, M., Pathak, A., Spalazzese, R., Steffen, B., Souville, B.: Towards an Architecture for Runtime Interoperability. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 206–220. Springer, Heidelberg (2010)
4. Autili, M., Chilton, C., Inverardi, P., Kwiatkowska, M., Tivoli, M.: Towards a Connector Algebra. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 278–292. Springer, Heidelberg (2010)
5. Saltzer, H., Reed, D.P., Clark, D.D.: End-to-end arguments in system design. ACM Trans. Comput. Syst. 2(4), 277–288 (1984)
6. Object Management Group, COM/CORBA Interworking Spec. Part A & B (1997)

7. Nundloll, V., Grace, P., Blair, G.S.: The Role of Ontologies in Enabling Dynamic Interoperability. In: Felber, P., Rouvoy, R. (eds.) DAIS 2011. LNCS, vol. 6723, pp. 179–193. Springer, Heidelberg (2011)
8. Bromberg, Y.-D., Issarny, V.: INDISS: Interoperable Discovery System for Networked Services. In: Alonso, G. (ed.) Middleware 2005. LNCS, vol. 3790, pp. 164–183. Springer, Heidelberg (2005)
9. Nakazawa, J., Tokuda, H., Edwards, W., Ramachandran, U.: A Bridging Framework for Universal Interoperability in Pervasive Systems. In: Proceedings of 26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006), Lisbon, Portuga (2006)
10. Cortes, C., Grace, P., Blair, G.: SeDiM: A Middleware Framework for Interoperable Service Discovery in Heterogeneous Networks. ACM Transactions on Autonomous and Adaptive Systems 6(1), Article 6:1-8 (2011)
11. Grace, P., Blair, G., Samuel, S.: A Reflective Framework for Discovery and Interaction in Heterogeneous Mobile Environments. ACM SIGMOBILE Mobile Computing and Communications Review 9(1), 2–14 (2005)
12. Duftler, M., Mukhi, N., Slominski, S., Weerawarana, S.: Web Services Invocation Framework (WSIF). In: Proceedings of OOPSLA 2001 Workshop on Object Oriented Web Services, Tampa, Florida (2001)
13. Masuoka, R., Parsia, B., Labrou, Y.: Task Computing – The Semantic Web Meets Pervasive Computing. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 866–881. Springer, Heidelberg (2003)
14. Singh, S., Puradkar, S., Lee, Y.: Ubiquitous Computing: Connecting Pervasive Computing Through Semantic Web. Information Systems and e-Business Management Journal (2005)
15. Ben Mokhtar, S., Preuveneers, D., Georgantas, N., Issarny, V., Berbers, Y.: EASY: Efficient SemAntic Service Discovery in Pervasive Computing Environments with QoS and Context Support. Journal of Systems and Software 8(5), 785–808 (2008)
16. Bromberg, Y., Grace, P., Reveillere, L.: Starlink: runtime intereoperability between heterogeneous middleware protocols. In: Proceedings of the 31st IEEE International Conference on Distributed Computing Systems, Minneapolis, USA (June 2011)
17. Martin, D., Burstein, M., Mcdermott, D., Mcilraith, S., Paolucci, M., Sycara, K., Mcguinness, D.L., Sirin, E., Srinivasan, N.: Bringing semantics to web services with OWL-S. World Wide Web Journal 10, 243–277 (2007)
18. Athanasopoulos, D., Zarras, A.: Fine-Grained Metrics of Cohesion Lack for Service Interfaces. In: Proc. of ICWS 2011 (to appear, 2011)
19. Bennaceur, A., Johansson, R., Moschitti, A., Spalazzese, R., Sykes, D., Saadi, R., Issarny, V.: Inferring affordances using learning techniques. In: International Workshop on Eternal Systems, EternalS 2011 (2011)
20. Mokhtar, S.B., Raverdy, P.-G., Urbieta, A., Cardoso, R.S.: Interoperable semantic and syntactic service discovery for ambient computing environments. IJACI 2(4), 13–32 (2010)
21. Straccia, U.: A Fuzzy Description Logic for the Semantic Web. In: Sanchez, E. (ed.) Fuzzy Logic and the Semantic Web, Capturing Intelligence, ch. 4, pp. 73–90. Elsevier (2006)
22. Heß, A., Kushmerick, N.: Learning to Attach Semantic Metadata to Web Services. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 258–273. Springer, Heidelberg (2003)
23. Krka, I., Brun, Y., Popescu, D., Garcia, J., Medvidovic, N.: Using dynamic execution traces and program invariants to enhance behavioral model inference. In: ICSE (2), pp. 179–182 (2010)

24. Bertolino, A., Inverardi, P., Pelliccione, P., Tivoli, M.: Automatic synthesis of behavior protocols for composable web-services. In: ESEC/SIGSOFT FSE, pp. 141–150 (2009)
25. Caporuscio, M., Raverdy, P.-G., Moungla, H., Issarny, V.: Ubisoap: A service oriented middleware for seamless networking. In: ICSOC, pp. 195–209 (2008)
26. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook. Cambridge University Press (2003)
27. Merten, M., Steffen, B., Howar, F., Margaria, T.: Next Generation LearnLib. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 220–223. Springer, Heidelberg (2011)
28. Issarny, V., Bennaceur, A., Bromberg, Y.-D.: Middleware-Layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 217–255. Springer, Heidelberg (2011)
29. Drummond, N., Rector, A.L., Stevens, R., Moulton, G., Horridge, M., Wang, H., Seidenberg, J.: Putting OWL in order: Patterns for sequences in OWL. In: OWLED (2006)
30. Vaculin, R., Sycara, K.P.: Towards automatic mediation of OWL-S process models. In: Proceedings of ICWS (2007)
31. Williams, S.K., Battle, S.A., Cuadrado, J.E.: Protocol Mediation for Adaptation in Semantic Web Services. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 635–649. Springer, Heidelberg (2006)
32. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
33. Howar, F., Jonsson, B., Merten, M., Steffen, B., Cassel, S.: On Handling Data in Automata Learning - Considerations from the Connect Perspective. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 221–235. Springer, Heidelberg (2010)
34. Zhang, M., Wolf, R.: Border Node Based Routing Protocol for VANETs in Sparse and Rural Areas. In: IEEE Globecom Autonet Workshop, Washington, pp. 1–7 (November 2007)
35. Durresi, M., Durresi, A., Barolli, L.: Emergency Broadcast Protocol for Inter-Vehicle Communications. In: Proc. 11th International ICPADS Conference Workshops, pp. 402–406 (2005)

Co-managing Software and Hardware Modules through the Juggle Middleware

Jan S. Rellermeyer[1,⋆] and Ramon Küpfer[2]

[1] IBM Austin Research Laboratory,
Austin, TX
rellermeyer@us.ibm.com
[2] Department of Computer Science, ETH Zurich,
Zurich, Switzerland
rkuepfer@student.ethz.ch

Abstract. Reprogrammable hardware like Field-Programmable Gate Arrays (FPGAs) is becoming increasingly powerful and affordable. Modern FPGA chips can be reprogrammed at runtime and with low latency which makes them attractive to be used as a dynamic resource in systems. For instance, on mobile devices FPGAs can help to accelerate the performance of critical tasks and at the same time increase the energy-efficiency of the device. The integration of FPGA resources into commodity software, however, is a highly involved task. On the one hand, there is an impedance mismatch between the hardware description languages in which FPGAs are programmed and the high-level languages in which many mobile applications are nowadays developed. On the other hand, the FPGA is a limited and shared resource and as such requires explicit resource management. In this paper, we present the Juggle middleware which leverages the ideas of modularity and service-orientation to facilitate a seamless exchange of hardware and software implementations at runtime. Juggle is built around the well-established OSGi standard for software modules in Java and extends it with support for services implemented in reprogrammable hardware, thereby leveraging the same level of management for both worlds. We show that hardware-accelerated services implemented with Juggle can help to increase the performance of applications and reduce power consumption on mobile devices without requiring any changes to existing program code.

Keywords: OSGi, FPGA, Hardware Acceleration.

1 Introduction

The increasing degree of dynamism in modern systems design and the resulting need for more flexible software becomes particularly apparent in mobile devices. Traditionally, mobile devices implement much of their performance-critical

⋆ The work was done while the author was at ETH Zurich, Switzerland. Part of this work was funded by the Swiss National Science Foundation SNF ProDoc program and by the Microsoft ICES initiative.

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 431–450, 2011.

functionality in *ASICs*, application-specific integrated circuits with a fixed implementation. Once manufactured and implemented in a mobile device the ASIC cannot be changed, e.g., for extending the functionality of the device or for applying critical updates. *FPGAs* (field-programmable gate array), in contrast, are known for their reconfiguration support and their ability to change the implementation of their functionality. With the technological advances in FPGAs, partially reconfigurable chips have been developed which can alter parts of their fabric at runtime.

An example of a mobile device that already makes use of reprogrammable hardware for on-demand acceleration is the Sony Playstation Portable and its Virtual Mobile Engine [15]. The main challenge, however, is the integration of FPGAs into applications. Whereas FPGAs are programmed in low-level hardware description languages like VHDL [6] or Verilog [7], the application software on mobile devices is often developed in high-productivity languages like Java, JavaScript, or Objective C. Bridging the gap between these two worlds is far away from being trivial, especially designing the communication interfaces between applications and FPGAs and effectively managing both the reprogrammable hardware and the software side of applications. Furthermore, in practice the design of the device mandates specific patterns of interaction.

Mobile phones, for instance, are naturally constrained in the way humans can interact with them due to their form factor. For a systems design, however, this means that most of the time a mobile phone is used for exactly one interactive (foreground) task whereas the remaining tasks are running in the background and have a lower priority. For example, when the user receives a phone call, the web browser functionality of the device becomes secondary. Ideally, a system could exploit this interaction pattern by using the reconfigurable hardware for always accelerating the interactive task. In the example of the phone call, this would be the audio encoding and decoding. Since the hardware is reconfigurable, the system can keep the invariant of accelerating the most critical interactive task even when the user switches from one application to another.

Implementing such systems requires the developer to overcome the impedance mismatch between hardware and software and an active handling of the inherent dynamism of the problem, which typically results in ad-hoc solutions. The contribution of this paper is the approach of creating an equivalence between software and hardware functionality by treating both as modules—running on and being managed by a common middleware platform. In order to do so, several concrete challenges need be solved:

- handling and managing both software and *hardware modules* where the latter are the different binary images (*bitstreams*) used to reconfigure the FPGA hardware.
- the ability to substitute one implementation of a functionality with another, e.g., a software module with an accelerating hardware module and back.
- making decisions when to do substitutions, given that the hardware resource is constrained so that typically not all tasks can run in hardware at the same time.

We provide a solution to these challenges with our implementation, *Juggle*, which takes advantage of the widely-used OSGi [12] standard for dealing with the life-cycle of software modules and extends it with support for functionality implemented in reconfigurable hardware. In contrast to approaches like, e.g., Liquid Metal [2], Juggle does not attempt to apply a unified design strategy for hardware and software in the small but instead focuses on the composition and interaction in the large. Juggle then takes care of the co-existence of software and hardware implementations and provides a unified model of communication through loosely-coupled services. Based on application-dependent policies, the system can thus dynamically switch between hardware and software implementation to accelerate most critical tasks without interrupting the system. As we show in the paper, the latency for switching is low enough to allow for dynamic replacement while the achieved acceleration for the evaluated use case of an encryption service reaches a factor of 20. The amount of energy consumed can be reduced by more than 97% compared to the same encryption done in Java and 59% when comparing to an implementation in C.

2 Background

Reprogrammable hardware like Field-Programmable Gate Arrays (FPGAs) is increasingly becoming powerful and affordable which makes them attractive to be used as a dynamic resource in systems. The following sections provide background information about FPGAs and their reconfiguration and discusses OSGi for managing software modules in Java.

2.1 Field-Programmable Gate Arrays (FPGAs)

Traditional integrated circuits are the result of a manufacturing process; once they are manufactured they cannot be altered any more. FPGAs, in contrast, are integrated circuits with the ability to be reconfigured after manufacturing either by the designer itself or the customer. This advantage especially comes into effect when the implemented functionality undergoes changes—one-time changes as in product line customization or continuous changes as through periodic upgrades.

Internally, an FPGA is structured into three main parts: a set of configuration logic blocks (CLB), a programmable interconnection network between the blocks, and a set of input and output cells around the device. The actual implementation of a configuration logic block (or basic block) can vary and depends on the concrete FPGA chip used.

Juggle has been prototyped on a Virtex-II Pro chip, which consists of groups of four *slices*, each containing two actual basic blocks. A basic block consists of a lookup table (LUT) with 4 inputs and an output, a set of multiplexers, arithmetic logic and a storage element. The LUT is a group of memory cells which contain all the possible results of a given function for a given set of input values. It can therefore implement arbitrary mappings between input and output ports. Altering the content of a LUT through a configuration consequently changes the behavior of the basic block.

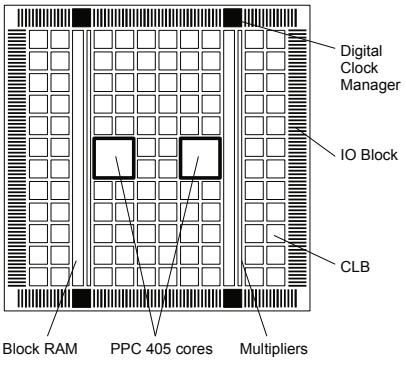


Fig. 1. Virtex-II Pro chip layout

The FPGA chip is a 2-dimensional array with the CLB as the smallest element. The precise layout of the FPGA structure such as the arrangement of the logic blocks and the interconnection paradigm of the logic blocks is vendor-dependent. Figure 1 shows a schematic picture of the Virtex-II Pro FPGA. The interconnect fabric is generally a network of vertical and horizontal wires arranged in a mesh topology. At the intersection points are programmable multiplexers facilitating the routing inside the FPGA fabric. Around the periphery of the FPGA chip are the I/O components used for communication with off-chip components. Those I/O components are programmable just like the CLBs and can use as input, output, or bidirectional gates.

The programming of an FPGA typically starts with a design of the required functionality in a hardware description language like VHDL or Verilog. This design can be considered as an abstract description without taking a particular technology into account. It operates on the register transfer level (RTL), a behavioral description in terms of signal flows between hardware registers. The mapping to a concrete technology is the task of an electronic design automation (EDA) tool, which synthesizes a *netlist* from the HDL code. This netlist now describes concrete gates and could be implemented in actual hardware. However, netlists only describe instances of gates, ports, and the wiring in between but not a concrete topology. It is the task of a place-and-route tool to create an instance of the template-like netlist which resembles a concrete layout of a digital circuit. In the case of an FPGA, it represents a configuration of the FPGA fabric that implements the designed functionality.

2.2 Reconfiguration

The data to configure an FPGA is called a *bitstream*. Bitstreams can be downloaded to the device via several configuration ports, e.g., a JTAG interface or a USB cable. This has the effect that the LUTs and the routing fabric of the FPGA is changed and hence the behavior is altered. Full reconfiguration—the process

of rewriting the complete design of the FPGA chip—requires a reconfiguration-time linear to the size of the bitstream to write. In addition, the entire chip is inoperable during the reconfiguration and running processes are interrupted.

Many chips therefore support the rewriting of only a part of the fabric. Specific regions on the chip are marked as reconfigurable and can at runtime be reconfigures through a *partial bitstream* while the remainder of the fabric can continue to operate. In order to support partial reconfiguration in a design, the FPGA fabric is partitioned in into a static region holding the functionality critical for the running of the system—e.g., the bus systems—and one or more regions that are partially reconfigurable (*PRRs*).

Functional tasks (reconfigurable modules or *PRMs*) can be mapped into individual PRRs (space multiplexing). If the tasks are mutually independent, they can also be mapped into the same PRMs (time multiplexing) to reduce the required FPGA real estate but at the same time introducing reconfiguration latency into the system. However, partial reconfiguration does not automatically mean that the board continues operation during reconfiguration. Depending on the hardware it can be the case that the reconfiguration requires the board to be in an inactive state. The ability of a chip to be reconfigured during runtime without interruption of the system is called dynamic partial reconfiguration.

One example of such a chip is the Xilinx Virtex-II Pro which was used to prototype Juggle. A complete discussion of the prototype system follows in Section 5. Partial reconfiguration requires the designer of a system to explicitly mark areas of the chip as reconfigurable. The place-and-route software has to take care that no signal lines are crossing these areas so that dynamic reconfiguration becomes possible. Otherwise the static part of the chip could encounter malfunctions during reconfiguration or even be short-circuit.

2.3 OSGi

In the domain of software modules, OSGi is a widely used middleware system for running and managing dynamic modules in Java. Historically, OSGi has its origins in embedded systems and mobile devices. Due to its flexibility and agility, it has recently been widely adopted in the latest generation of Java enterprise application servers. OSGi describes a runtime system that sits atop the Java virtual machine and provides primitives for controlling the life-cycle of software modules. At runtime, new modules can be installed and modules no longer needed can be completely removed from the system. Furthermore, OSGi supports consistent updates of modules. The unit of modularity in OSGi is the *Bundle*. From a technical perspective, a bundle is nothing but an ordinary JAR file—a compressed filesystem with a manifest as commonly used in Java—but enriched with additional meta-data. Most importantly, bundles have to declare their dependencies explicitly. The default case in OSGi is that bundles do not share any code but run in complete isolation. Sharing is possible when corresponding Java packages contained in a bundle are declared to get exported and consequently are imported by another bundle. This indeed creates a tight

coupling between bundles since the importing bundle cannot be resolved without an exporter already installed.

Orthogonal to the module layer, OSGi provides applications with a service layer to facilitate a loose coupling among components. Every bundle can register any Java object with the runtime system under one or more service interfaces. The OSGi runtime maintains a central service registry through which bundles can search for services. Consuming bundles are typically only tightly coupled to the service interface but no longer to the service implementation with all its transitive dependencies. When a service is acquired by a bundle, it gets the actual Java service object so that no further overhead other than the initial interaction with the runtime can be observed.

An important difference between traditional application design and the OSGi world is the handling of dynamism. Whereas usually software is assumed to be a static and unchangeable unit, in OSGi a module should never make assumptions about the permanent availability of any other module or service. An operator could at any time unload a module or stop it, which causes the removal of all registered services. Hence, OSGi bundles typically register listeners to get informed about changes in the topology and react accordingly. As a further consequence, OSGi bundles are usually written with a high degree of locality so that exchanging one service implementation with another can often be done seamlessly.

3 Management and Substitution of Modules

Introducing an FPGA into a mobile or embedded system enables applications to implement parts of their performance-critical functionality in hardware. These hardware modules are physically handled as bitstream files. Reprogramming the device requires the writing of a bitstream to a reconfiguration device embedded into the system. The first step in making FPGAs easier to use in applications is to provide management of the bitstreams of the same quality as for software modules. In a system like OSGi, this gives both the application itself and an external operator the possibility to explicitly control the composition of an application and the life-cycle of the individual components. However, this alone does not solve the integration problem. Whereas software modules can be seamlessly used in the programming languages (e.g., in OSGi through package imports), FPGAs constitute hardware components and have much more low-level communication interfaces like memory-mapped I/O ports, registers, or interrupts. In order to preserve the full flexibility of modularity, the interfaces between a software and the corresponding hardware module have the be uniform. In practice, this means that the representation of the FPGA requires the co-design of a *device driver* in software which embeds it into the host programming language.

3.1 Hardware-Accelerated Services

A hardware module in the first place consists of a partial bitstream designed to configure the core functionality of the service into a partial configuration region

of the FPGA. The bitstream is embedded into the OSGi bundle as a file. Once this bitstream is applied to the FPGA, the hardware is ready to be used but still not accessible from Java. The virtual machine approach prevents Java code from accessing the underlying physical machine. Hence, the device driver is typically coded in C and makes use of the Java Native Interface (JNI) to bridge between Java and the hardware. From the point of view of the Java OSGi application, the device driver is represented through a Java class in which all critical methods map to JNI native code methods.

When loaded, the JNI code initializes by mapping the hardware addresses into the virtual memory of the JVM process as well as registering handlers for interrupts or initializing DMA. For each service method, there is a piece of code turning the service call into one or more interactions with the FPGA. Usually, this involves a mangling of the arguments and selective stores and loads of portions of the arguments into memory, waiting for a result to become available and then preparing the return value for caller. Even though writing the driver is still a challenge that requires a skilled programmer, most of this can be done in a more declarative way that takes full advantage of having a clear specification of the hardware interface on the one hand and the high-level service interface on the other. For instance, the driver code could be generated from the domain-specific language (like, e.g., in Devil [9]).

The pair of driver and bitstream is the foundation for the hardware service and dual to the Java software service implementation of the same service. When the hardware service implements the same interface it can replace the latter on demand. There might, however, be cases in which a certain consumer of a service should get accelerated by a hardware implementation while others should continue to run against a software implementation. Such a pattern of interaction is far away from being trivial to implement in OSGi since in general the application chooses the service and not the service the application.

In order to still support for such use cases we introduce *Co-Modules*, which are modules providing both a software and hardware implementation of the service at the same time. Such modules can register a common proxy service as an indirection in between the service exposed to applications and the back-end implementation. As a result, co-modules can seamlessly switch between either of the two implementations (if the hardware resource is available) and provide seamless dynamic acceleration. If the service is implemented as an OSGi *ServiceFactory*, it can even selectively accelerate the service only for certain consumers and serve requests from other bundles through the software implementation. Figure 2 shows a structural overview of both a hardware module (a module containing only a hardware implementation of a service) and a co-module.

3.2 The FPGA Bundle Extender

The basic unit of modularity in OSGi is the bundle. Even though there are very few requirements for a bundle to participate in an OSGi application, in practice there is a small piece of code in the bundle which interacts with the runtime and registers or consumes services. This code is specific to OSGi whereas most

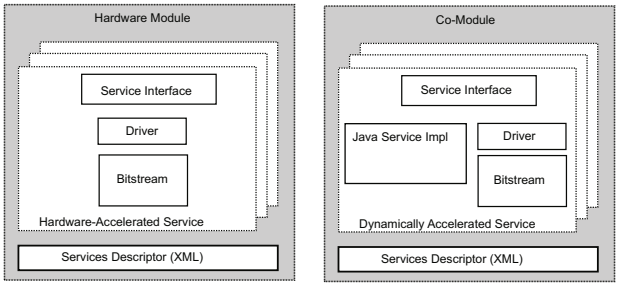


Fig. 2. Structure of hardware-accelerated OSGi services

of the remaining bundle code is standard Java. For some applications, however, this OSGi-specific code is a liability. For instance, considering a web application server based on OSGi, every web application is preferably a bundle and registers its servlets as services. In traditional Java EE, however, web applications are packaged in WAR files, which are JAR files with a set of specific XML configuration files. The requirement to write the boilerplate code so that the servlets are discovered from the `web.xml` file and registered as services so that the server engine becomes aware of their existence is a burden for the adaption of the OSGi model for web applications. The solution to the problem is the extender pattern.

In the extender pattern, there is a singleton entity—called extender—in the system that listens for newly installed bundles. Whenever a new bundle is installed, it scans the content of the bundle for the existence of a specific configuration file. If such file is present, the extender interprets this file and extends the bundle by, for instance, registering services on behalf of the bundle. Thereby, in principle plain WAR files can be used within an OSGi deployment; the extender takes care of integrating the content of the file into the application server.

For Juggle, a similar approach is taken. An FPGA extender listens for new bundles containing a configuration file for hardware-accelerated services and then registers the service on behalf of the bundle. Listing 3 shows an example of a configuration file. Each bundle can contain arbitrarily many hardware-accelerated services. As for traditional OSGi services, properties can be attached to the service on which clients can filter their requests. Instead of selecting either the Java or the FPGA-based service, the FPGA extender generates a service proxy from the service interface. The purpose of the proxy is to provide the system with an interception point located between the caller and the service. This enables the system to seamlessly switch between a software service and a hardware service as well as tracing service invocations to derive performance information.

4 Juggling Software and Hardware-Accelerated Services

Not only is the FPGA a singleton entity in the system, the resources of the FPGA in terms of logic gates are also limited and permit—depending on the complexity of the service—just one or a small number of hardware-accelerated

```
<?xml version="1.0"?>
<co-module xmlns="http://flowsgi.inf.ethz.ch/comodules">
  <accelerated-services>
    <accelerated-service interface="math.AddService">
      <java-service>math.MathAddImpl</java-service>
      <fpga-service>
        <driver>math.MathAddDriver</driver>
        <bitstream>opb_prr_0_adder_partial.bit</bitstream>
      </fpga-service>
      <service-properties>
        <entry key="version" value="1.0.0"/>
        <entry key="foo" value="bar"/>
      </service-properties>
    </accelerated-service>
  </accelerated-services>
</co-module>
```

Listing 3. Juggle service descriptor example

services to co-exist at any given time. It is hence inevitable to make resource scheduling decisions and set priorities. For this purpose, Juggle continuously traces service invocations and assembles statistical data to make decisions which services can run in software and which can profit from hardware acceleration.

Deciding which service of a single application to swap into hardware is a policy decision and can thus be best made by the application itself. On an OSGi runtime and particularly on mobile devices, however, it is not unusual to run multiple applications simultaneously. Deciding in favor of a specific application hence requires coordination. However, global knowledge about the setup is against the principle of modularity; a module should only reason locally and not require knowledge about other modules installed on the same system beyond declared or loosely-coupled dependencies.

Juggle deliberately avoids implementing policies and instead expects the platform to implement a *controller* defining the criteria to be used for reconfiguration. For instance, such a policy could be that the application currently running in the foreground and having the focus of the user is prioritized over the background applications. Which service to prioritize could therefore be determined by the window manager, which is by definition an entity with full knowledge of all modules currently using its services. What the system has to provide is access to the basic collected performance data of each service, such as invocation frequency, average duration of the invocations, etc. and a simple imperative command interface to turn a software into a hardware-accelerated service and vice versa. This command interface consists of a single primitive: the *juggle* operation. As arguments, the juggle operation takes the service id of a service to turn into a hardware-accelerated service as well as the ids of previously hardware-accelerated service to turn back into software services.

4.1 Reprogramming the FPGA

When the controller has issued a juggle operation, the system first does a sanity check, e.g., if the freed slots are adjacent and if the reclaimed FPGA space is sufficient to accommodate the new service. If the check passes, the hardware can be reprogrammed. The Xilinx FPGAs are able to perform a *glitchless* dynamic reconfiguration. This means, if a resource on the chip—despite being in the reconfigured ares—is not affected by the reconfiguration it can be accessed without interruption. There can, however, be problems in the design phase of hardware service implementations. If the place-and-route tool does not have to meet any communication constraints between two hardware services the signal will much likely cross the boundaries of the partial reconfiguration area where the best timing can be achieved. The routing can be different for every hardware service implementation.

The solution is the usage of bus macros [3] which can be seen as fixed data paths for signals going between PRRs. Bus macros serve the purpose of a socket where the corresponding hardware service can be plugged into the system. Hence, they provide the means of locking the routing between hardware services and the static part, making the modules pin compatible with the base design. In addition to locking the routing path, bus macros also serve as switches to enable and disable the transmission of signals. The signal propagation has to be disabled during reconfiguration, and enabled after, to avoid bus congestion or even a corruption of the bus during the reconfiguration process.

In the Juggle design, the bus macros for PRRs are encapsulated into their own IP core (a reusable unit in the hardware design process), the socket bridge. In our prototype system, the socket bridge is controlled over the Device Control Register (DCR) bus. This bus bypasses the standard memory bus and bus controller for low latency and implements a daisy-chain architecture propagating the signals to all attached cores. The communication between the runtime system and this IP core happens through a kernel-driver in the operating system. In our prototype system, we use Linux and have developed a driver which registers a character device to accept control words for opening or closing the socket bridge. When reading from the character device, the current status of the bridge can be retrieved. Since character devices in Linux are represented through ordinary file descriptors, the Java VM can access them as random access files.

The actual reconfiguration happens through the internal configuration access port (*ICAP*) interface. The ICAP device is supported under Linux by a driver in the patched Xilinx kernel and can therefore also be accessed from Java. The system has to open the socket bridge for the RPP, retrieve the partial bitstream from the bundle, write it to the `/dev/icap` character device, and close the socket bridge again. Subsequently, the JNI driver for the hardware service is loaded. If the service is hardware only, the driver is now registered as an OSGi service under the designated service interfaces. For co-modules, there is already an existing service proxy which needs to be altered to redirect calls to the JNI driver and hence to the hardware service. In order to perform juggling from the software to the hardware service in a consistent manner, all pending service method calls

that are still accessing the software implementation will run to completion in software whereas all newly method calls use the hardware. This is consistent with the behavior of dynamic code modifications in the JVM like the *RedefineClasses* function in the Java Virtual Machine Tool Interface (JVMTI) which is frequently used in runtime debugging tools or for runtime aspect-oriented programming (AOP) support.

5 Juggle Prototype System

Our prototype system uses the Xilinx XUPV2P development board [19] containing a Virtex-II Pro FPGA with a total number of 30,816 programmable logic cells. In addition, the FPGA chip contains two embedded PowerPC 450 cores running up to 300 MHz. The cores can have an instruction and data cache with up to 16 KB each and a MMU. Xilinx provides a patched Linux kernel tree that runs on the PowerPC cores.

The system boots off a flash device containing the system ACE file which initially configures the PPC cores as well as programming the static parts of the FPGA required to connect the PPC cores to the peripheral hardware. The PowerPC405 program counter is set to the starting address of the Linux kernel also contained in the ACE file. During and after the boot process the kernel can use the flash card as a secondary storage device for its root file system.

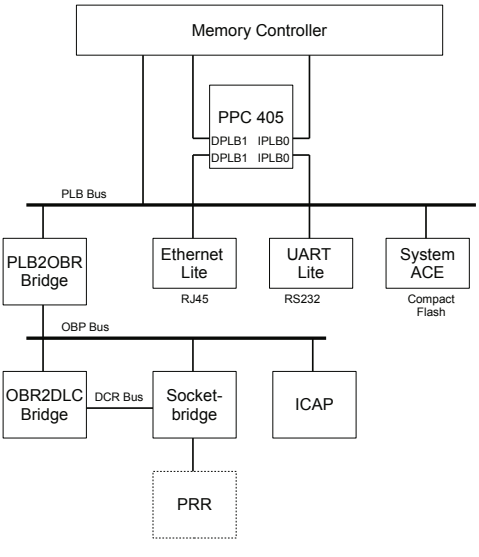


Fig. 4. Base design of the prototype system

Figure 4 shows a block diagram of the base system design used for the prototype system. A single PowerPC core is attached to a Processor Local Bus (PLB)

which is part of the IBM CoreConnect Bus Architecture specification [4] and in this system serves as a communication backbone. An Ethernet connector, a serial port, and the compact flash connector for the card holding the system ACE file are attached to this bus through their controller logic cores. In addition, there is a bridge which connects to a second bus, the On-Chip Peripheral Bus (OPB). Even though this bus type is deprecated in recent versions of the Xilinx tools, the Virtex-II Pro internal configuration access port (ICAP) is only capable of communicating through the OPB. Later versions of Virtex chips feature ICAP devices that can be directly attached to the PLB. Our prototype contains only a single PRR, for the proof of concept, which is attached to the OPB through a socket bridge. This has the consequence that only a single hardware-accelerated service can run at any time.

The figure shows the logical structure of the base system. Physically, the PR region of the system has been placed at the right edge of the chip. The reason is that the entire memory bank is connected to the left side of the chip so that choosing the right side for the PRR keeps the the number of static routes crossing the module boundaries low. As a consequence, the RP region can cover almost the full height of the device except for four rows of IOB and IOI at the top and bottom. The width of the PR region spans 8 CLBs, leading to a total size of almost 16% of the FPGA fabric (Table 1):

Table 1. Physical resources of the FPGA chip and the PRR

	Slice	Mult	Ram16	TBUF
Entire FPGA	13696	136	136	6848
PRR	2240 (16.35%)	20 (14.7%)	20 (14.70%)	6848 (16.35%)

The prototype runs the PowerPC core at 300 MHz and features 256 MB of external DDR SDRAM. As an operating system, it uses the patched Xilinx Linux kernel based on version 2.6.35 and a Java virtual machine (three different VMs have been successfully tested). After the system has booted, about 190 MB remain available for user-space programs such as the JVM and Juggle. Due to the constrained resources, Juggle relies on an updated version of the highly optimized Concierge [13] OSGi technology implementation. The OSGi framework is enhanced with support for hardware-accelerated OSGi bundles through an FPGA extender. The prototype system does not feature an autonomic controller for juggling software and hardware implementations of services. Instead, it registers an extension service for the Concierge shell so that the juggling can be triggered on demand by the user of the system.

6 Evaluation

The use case for evaluating Juggle is an application that requires encryption. This can, e.g., be the encryption of data on the internal storage of the device

or a secure protocol which encrypts the data before transmission. Normally the encryption functionality would either be implemented in software or, if performance critical, in hardware as an ASIC. Security, however, is one of the areas that require a constant update of the technology used due to bugs in implementations and exploits through weaknesses in the algorithms. For instance, if the mobile device was shipped with an ASIC accelerating the encryption of data with the Data Encryption Standard [10] (DES), the de-facto standard until 2004, it would be obsolete by now since with todays possibilities the DES encryption cannot be considered secure. If, however, the encryption is implemented as a hardware-accelerated service, the device becomes much more flexible and future-proof. First, the encryption algorithm can be exchanged at any time, e.g., with a Triple-DES encryption [11], even at runtime. Second, encryption can selectively run hardware-accelerated, e.g., when the performance of the interactive process is limited by the encryption of data. An example would be a user decrypting an larger email message. When the user switches the foreground task, e.g., to the music player, the audio decoding becomes the hardware-accelerated service and any encryption happening in the background runs through the software implementation of Triple-DES.

6.1 DES and Triple-DES as Hardware-Accelerated Services

A DES and a Triple-DES encryption service have been implemented as hardware-accelerated OSGi services for Juggle. Both services are implemented in software—using either the Java Cryptography Extension (JCE) provider shipped with the VM or the BouncyCastle [16] Java library—and in hardware in the form of a partial bitstream for the PR region in the base system. Listing 5 shows the common service interface of both DES and Triple-DES so that one can be easily exchanged with the other.

```
public interface EncryptionService {
      void loadKey (byte [] key);
      int encrypt(ByteBuffer data, int size);
      int decrypt(ByteBuffer data, int size);
}
```

Listing 5. Interface of the EncryptionService

The bitstreams for hardware-accelerated services for a given PRR have the identical size (≈145 kB in case of our prototype system) since the entire PRR is reconfigured in either case. In practice, tools for partial reconfiguration like Xilinx PlanAhead create compressed bitstreams so that there can be slight variations in the size depending on the complexity of a design. Table 2 shows a detailed resource consumption of the two hardware implementations when programmed into the FPGA. As reference points we have added two simple

hardware-accelerated services we used during the development of the system, an adder service and a multiplication service, which both take two Java primitive type integers as input and return the result of the arithmetic operation as a Java integer. The Triple-DES implementation in fact contains three instances of DES and hence consumes about three times more chip real-estate than the DES implementation. Both hardware-accelerated services leave enough space so that potentially other services could run in parallel if the base system was designed to support this.

Table 2. Physical resources used by different hardware-accelerated services

	PRR	add	mul	DES	Triple-DES
LUT	4480	90 (2.01%)	58 (1.29%)	1081 (24.13%)	3008 (67.14%)
Flipflop	4480	176 (3.93%)	144 (3.21%)	513 (11.45%)	1527 (34.08%)
Slice	2240	108 (4.82%)	88 (3.93%)	660 (29.46%)	1835 (81.92%)
Mult	20	0	1 (5.00%)	0	0

Since the reconfigurable area is always entirely overwritten, the reconfiguration time is solely a function of the target service and not of the service previously located in the PRR. Hence, the time depends on the number of elements to be reconfigured. Table 3 shows the exact sizes of the example bitstreams and the reconfiguration times. The static full bitstream used to boot the system is given as a reference point in terms of bitstream size but it indeed cannot be used for reconfiguration. The reconfiguration time for our examples varies between 11.2 and 24.9 milliseconds. Values reported in the literature indicate that in general the reconfiguration time of the Virtex-II Pro is between 10 and 35 milliseconds. There is an additional overhead involved in switching the socket bridge, which adds on average about three milliseconds. In total, the time to juggle a service is hence between 15 and 30 milliseconds for our examples. This indicates that an on-demand reconfiguration is feasible given the latency requirements of applications typically found on mobile devices.

The time to create the initial hardware service is in range of 100 milliseconds for our DES and Triple-DES example and includes the time to create the service proxy and the time to load the JNI driver for the hardware. Loading either of the

Table 3. Size of the bitstream and reconfiguration time

bitstream	size in bytes	reconfiguration time
static full.bit	*1448817 (100%)*	—
add	123249 (8.50%)	11.159 msec
mul	128222 (8.85%)	24.896 msec
des	149087 (10.29%)	13.441 msec
tdes	149088 (10.29%)	13.374 msec

two encryption bundles, which are implemented as co-bundles and also register a software service, takes less than a second. The startup time of Juggle, which includes the startup time of the Java virtual machine, Concierge with the basic bundles, and the load time of the FPGA extender bundle, is on average 6 seconds in the prototype setup.

6.2 Acceleration through Juggle

Dynamic juggling of software and hardware services has shown to be feasible for a large class of applications. What remains to be shown is that hardware services have in fact a significant potential for speeding-up Java programs. For this purpose, we have evaluated different ways of performing TripleDES encryption on our prototype board using different Java virtual machines.

For the PowerPC architecture, there is no implementation of the original Sun Java HotSpot Virtual Machine. However, since the sources were released to the open source community, the IcedTea project [5] has implemented a portable version of the OpenJDK which is largely free of assembly code (IcedTea Zero) and has been successfully ported to the PowerPC and other architectures. The main caveat of the IcedTea Zero VM is that it is purely interpreting and does not feature just-in-time compilation. Hence, the performance of applications running on the IcedTea Zero VM is significantly lower than on a JIT-enabled virtual machine. For comparison, we have taken a version of the IcedTea VM enhanced with the Cacao [8] just-in-time compiler. Both versions are based on the same Java 6 version 1.8.2 build 18 of IcedTea. The Zero VM is version 14.0-b16, the Cacao JIT corresponds to the released version 0.99.4. The third virtual machine used is the IBM J9 VM for PowerPC. This virtual machine has JIT and is typically used in production servers of the PSeries but also runs on the PPC 405. The version used is J2RE 1.6.0 IBM J9 2.4 build pxp3260-20071123_01.

As a first reference point, we have measured the performance of different Java TripleDES implementations on the three virtual machines running on our evaluation prototype system. In general, cryptography for Java applications is supported through the Java Cryptography Extensions, an API for data encryption, authentication, and key management. All three virtual machines used in the experiments ship with a JCE provider. The providers of the two IcedTea flavors are identical while the IBM implementation is based upon a different code-base. For further comparison, we have used the open-source JCE provider *BouncyCastle* [16] in its latest version 1.38 for Java 6. This library can uniformly run on all three virtual machines.

The experimental setup for this and all following experiments is the encryption of a buffer filled with random bytes, using TripleDES with the same fixed key. The size of the buffer is varied in the experiments to get an impression of the overall performance characteristics of the corresponding implementations. Figure 6 summarizes the experimental results for the various Java implementations. As a baseline of comparison, the graph additionally shows the performance of a C program using the Triple-DES implementation from Eric Young's *libdes*.

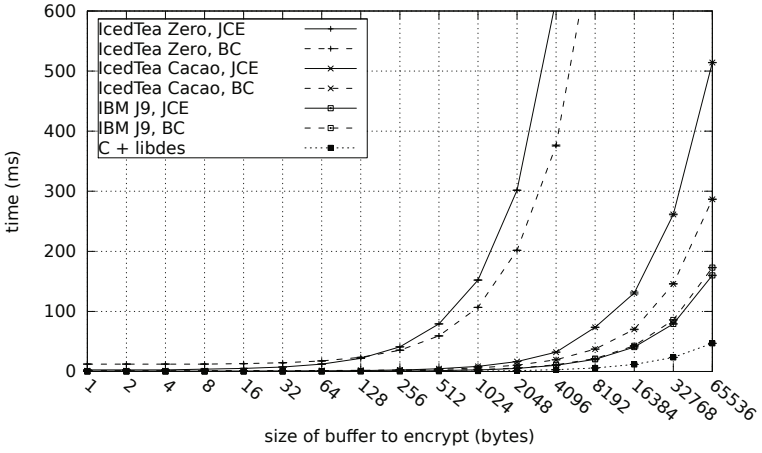


Fig. 6. Performance of Triple-DES encryption in Java

The first conclusion to be drawn from the results is that the interpreting VM has a significantly lower performance than the other two VMs. This can be expected since cryptography is computation-intensive and can hence profit to a large extent from just-in-time compilation. The IBM VM performs better with its own JCE provider whereas on Cacao BouncyCastle performs better than the built-in OpenJDK provider. Overall, however, even this implementation is still almost a factor of three slower than the C implementation, which is surprising given its JIT but it is possibly constrained by the available systems resources.

The next experiment compares the performance of the two software implementations (IBM J9 JCE and C+libdes) with the performance of a hardware-accelerated Juggle service running on the different VMs. The hardware-accelerated service consists of a Java interface, a JNI driver, and the corresponding Triple-DES logic in the FPGA.

The JNI driver exchanges data with Java through ByteBuffers and shuffles data to the FPGA TDES core by writing to software registers. Triple-DES is a block cypher. For the encryption of each block (of 8 bytes), the JNI driver writes the data into two 32 bit registers of the TDES core and then alters a status register to indicate that the data is ready and the requested operation is an encryption. When the core has encrypted the data, it sets the status register to a success value. The JNI driver busy-waits on the content of the status register and then reads back the encrypted result from two 32 bit registers. This design has been mainly chosen for simplicity, more sophisticated implementations might further improve the performance of the hardware. The TDES core runs with the bus clock speed, which is 100 MHz in the prototype system and therefore a factor of three lower than the CPU clock.

Figure 7 shows the measured results and, as a baseline, the performance of a C implementation using the same FPGA TDES core for acceleration. Hardware-accelerated encryption in Java can provide a performance equal to using the

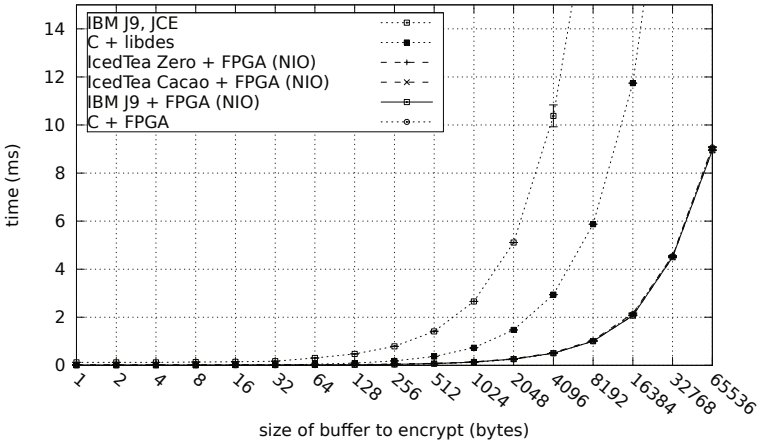


Fig. 7. Performance of Triple-DES encryption through a hardware-accelerated service

FPGA directly from C since the static overhead becomes irrelevant for realistic buffer sizes. When encrypting buffer of at least 64 bytes size, a Juggle hardware service accelerates the encryption by a factor of almost 20 compared to the best Java software service. Furthermore, when accelerating the performance-critical code through hardware implementations, the interpreting VM can reach almost the performance of a JIT-enabled VM. This is an interesting design option for highly resource-constrained systems that cannot afford the memory and storage footprint of a just-in-time compiling VM.

6.3 Power Consumption

Besides performance, power consumption is a major issue for mobile and battery-powered embedded devices. Therefore, we evaluated the power consumption of three different implementations of TDES by using a wattmeter introduced between the power supply of the prototype board and the power outlet. Hence, the values measured determine the consumption in the primary circuit and correspond to the de-facto consumption observed by an operator of the device. For benchmarking, we used the TDES encryption of a buffer of 1024 bytes in 30 runs of loops of 1000 encryptions and measured the power consumption for the OpenJDK IcedTea VM with the Cacao JIT and the JCE encryption provider, for the C implementation using the libdes library, and for the same OpenJDK IcedTea/Cacao VM but using the FPGA for the encryption. Figure 8 shows the results for the three different implementations. The power consumption of the board in the idle state is 8.75 W, illustrated by the dashed horizontal line. The two Java implementations have a peak consumption in the first six seconds, which is the time that the JVM takes to start. After this startup time, the software implementation has a relatively stable power consumption of 9.1 W whereas

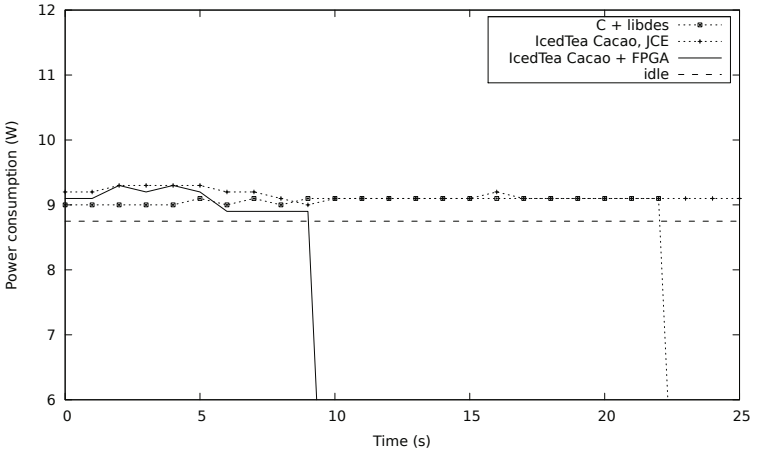


Fig. 8. Power consumption of the different TDES implementations

the FPGA-based implementation uses only 8.9 W. The C implementation has a constant consumption of 9.1 W.

Integrating the power consumption over the runtime of the test run gives the total energy required for performing the encryption task. The pure Java implementation, which has a total runtime of 346 seconds (the figure does not show the entire runtime for Java/JCE), consumes 3151 Joule. With the C implementation, the device consumes a total of 199.6 Joule, only slightly more than 6% of the energy for Java/JCE, mainly due to the significantly lower runtime. When accelerated through Juggle, however, the encryption can be performed in Java using only 81.7 Joule, which is about 41% of the energy spent with the C implementation.

7 Related Work

Juggle is not the first attempt to interface between a high level language like Java and reconfigurable hardware. JBits [1] is a set of Java libraries that can read bitstreams either generated by the toolchains or from a currently running FPGA device. It provides an API to modify a configuration bitstream and use it to reprogram the device. Unfortunately, JBits provides little abstraction over a hardware description language. Hence, it is highly platform-dependent and requires the using application to explicitly deal with low-level details such as the routing.

Liquid Metal [2] features the Lime language which is based on Java but extended with a special and more restricted type system amenable to bit-level analysis. Lime can be compiled both into Verilog and successively into FPGA bitstreams as well as to Java byte-code. The target domain of Liquid Metal is similar to Juggle as both systems target devices with both a conventional CPU

and an FPGA as an additional resource for dynamic acceleration. The major difference is the level on which the systems operate. Liquid Metal attempts to create a unified language for both the software and the hardware design. Juggle in turn focuses on the integration of the two worlds through composition of modules.

A different approach has been taken with JOP [14], a Java optimized processor implemented on top of an FPGA. The motivation of JOP is to enable the use of high-level productivity languages like Java for programming FPGA chips. The authors point out that the programming languages usually used on top of systems on a chip like C and Assembler provide poor abstractions to the programmer. The result of this consideration is a JVM implemented as a processor on an FPGA. The original Java byte-code is translated by the processor into an address in the own microcode format of the FPGA-driven JVM.

Ullmann et al. [17] present a complete approach to a module based architecture for automotive control devices. Todays automobile classes contains up to 100 control devices which quickly obsolete and decreases the product life cycle from 5 to 2 years. The adaptivity of reconfigurable devices can increase the product life cycle while reducing the cost and risk for development and later maintenance.

A high-level approach for using the Xilinx FPGA reconfiguration is explained in the work of Williams et.al. [18]. They present a modular platform for RSoC called Egret designed around the idea that complex systems can and should be designed by composition. The specification of an assembled hardware module stack is given to a software tool that constructs the appropriate FPGA configuration, as well as software infrastructure such as device drivers.

8 Summary and Discussion

Juggle shows that a modular and loosely-coupled approach to integrating software and reprogrammable hardware facilitates a flexible and dynamic coexistence between software and hardware services. Applying the same management facilities to both worlds simplifies the development of such systems and at the same time gives the maintainer the opportunity to alter the setup even at runtime. OSGi is extensible enough to be used for this purpose and the extender pattern avoids large parts of the boilerplate code required for registering services. The latency of reprogramming reconfigurable areas is low enough to seamlessly switch between the software and hardware and accelerate the most critical tasks at any time. Due to the common interface that Juggle applies to both software and hardware services, the substitutability principle of modularity ensures that existing applications do not need to be modified for Juggle. Hence, their performance can gradually be improved through introducing hardware services. In practice, the degree of acceleration can be significant, as shown with TripleDES where we reached a speedup of 20 despite the unoptimized hardware design.

References

1. Guccione, S., Levi, D., Sundararajan, P.: JBits: Java based interface for reconfigurable computing. In: MAPLD 1999 (1999)
2. Huang, S.S., Hormati, A., Bacon, D.F., Rabbah, R.: Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 76–103. Springer, Heidelberg (2008)
3. Huebner, M., Becker, T., Becker, J.: Real-time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration. In: SBCCI 2004: Proceedings of the 17th Symposium on Integrated Circuits and System Design, pp. 28–32 (2004)
4. IBM Microelectronics: CoreConnect Bus Architecture (1999), https://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture
5. IcedTea Project: OpenJDK IcedTea (2006), http://icedtea.classpath.org
6. IEEE: Standard VHDL Language Reference Manual. ANSI/IEEE Std 1076-1993 (1994)
7. IEEE: Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language. IEEE STD 1800-2009 (2009)
8. Krall, A., Grafl, R.: CACAO - A 64-bit JavaVM Just-in-time Compiler. Concurrency: Practice and Experience 9, 1017–1030 (1997)
9. Mérillon, F., Réveillère, L., Consel, C., Marlet, R., Muller, G.: Devil: an IDL for hardware programming. In: OSDI 2000 (2000)
10. National Institute of Standards and Technology: Data Encryption Standard (DES). FIPS Publication 46-2 (1993)
11. National Institute of Standards and Technology: Data Encryption Standard (DES). FIPS Publication 46-3 (1999)
12. OSGi Alliance: OSGi Service Platform, Core Specification Release 4, Version 4.2, Draft (2009)
13. Rellermeyer, J.S., Alonso, G.: Concierge: A Service Platform for Resource-Constrained Devices. In: EuroSys 2007: Proc. of the ACM SIGOPS/EuroSys European Conference on Computer Systems, pp. 245–258. ACM (2007)
14. Schoeberl, M.: JOP: A Java Optimized Processor. In: Meersman, R. (ed.) OTM-WS 2003. LNCS, vol. 2889, pp. 346–359. Springer, Heidelberg (2003)
15. Sony Corporation: Virtual Mobile Engine (VME). http://www.sony.net/Products/SC-HP/cx_news/vol42/pdf/sideview42.pdf (2002)
16. The Legion of the Bouncy Castle: Bouncy Castle Java Cryptography APIs (2000), http://www.bouncycastle.org/java.html
17. Ullmann, M., Huebner, M., Grimm, B., Becker, J.: An FPGA Run-Time System for Dynamical On-Demand Reconfiguration. International Parallel and Distributed Processing Symposium 4, 135a (2004)
18. Williams, J.W., Bergmann, N.W.: Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-on-Chip. In: ERSA, pp. 163–169 (2004)
19. Xilinx Inc.: Xilinx University Program, Virtex-II Pro Development System, Hardware Reference Manual (2009), http://www.xilinx.com/univ/XUPV2P/Documentation/ug069.pdf

A Generic Solution for Agile Run-Time Inspection Middleware

Wouter De Borger, Bert Lagaisse, and Wouter Joosen

Distrinet, Department of Computer Science, KULeuven, Belgium
{wouter.deborger,bert.lagaisse,wouter.joosen}@cs.kuleuven.be

Abstract. Contemporary middleware offers powerful abstractions to construct distributed software systems. However, when inspecting the software at run-time, these abstractions are no longer visible. While inspection, monitoring and management are increasingly important in our always-online world, they are often only possible in terms of the lower-level abstraction of the underlying platform. Due to the complexity of current programming languages and middleware, this low-level information is too complex to handle or understand.

This paper presents a run-time inspection system based on dynamic model transformation capabilities that extends run-time entities with higher-level abstract views, in order to enable inspection in terms of the original and most relevant abstractions. Our solution is lightweight in terms of performance overhead and agile in the sense that it can selectively (and on-demand) generate these high-level views.

Our prototype implementation has been applied to inspect distributed applications using RMI. In this case study, we inspect the distributed RMI system using our integrated overview over the collection of distributed objects that interact using remote method invocation.

1 Introduction

Run-time analysis and run-time inspection of software is required at various stages of the software engineering life cycle: from the early prototyping phases, over the debugging phases that are inherently present when preparing for release, to the deployment phases when the software is exploited in a production environment where profiling and monitoring are important for management purposes.

Many distributed software systems, for example based on the service oriented computing paradigm, or built using web service technology cause major and ineffective efforts to enable dynamic and run-time analysis. In such systems, the operational code is the result of composing and translating many building blocks, developed using different technologies at different abstraction layers. For example, in contemporary distributed systems, applications are represented at different layers of abstractions, ranging from business process management (BPM) support such as the Business Process Execution Language (BPEL [10]) over web services and enterprise component models, to plain object oriented

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 451–470, 2011.

artifacts and possibly native code. The run-time application typically consists of byte code and data structures that cannot be used to observe higher level abstractions that the BPM developer, system operator or web service integrator might understand. In summary, the run-time representation of such a program is a complex synthetic structure that is not suitable for run-time monitoring by the system operator and that is foreign even to the original developer.

This trend is further evolving due to the versatile modeling and programming languages that can be combined in a single system. Meanwhile platforms and operating environments (cloud based systems, Internet of things etc) tend to become more heterogeneous. The need for run-time inspection and dynamic program analysis increases rapidly. Various stakeholders (developers, operators etc.) should be capable of building dynamic program analysis features that represent the abstraction and concepts that match their understanding of the software.

To enable inspection of such complex composed systems, we present an approach to run-time inspection, based on dynamic model transformation capabilities to extend run-time artifacts with higher-level abstract views, in order to enable inspection in terms of the relevant abstractions. Our solution is lightweight in terms of performance overhead and agile in the sense that it can selectively (and on- demand) generate these high-level views. We combine model transformation and reflective technology to enable a declarative specification of the relation between abstractions. This declarative specification is automatically converted into a mirror-based inspection system that reconstructs representations of higher-level abstractions [1].

The core element of our approach is a generator that is capable of converting the declarative specification of relationships between views on a system (at various abstraction levels) into an actual system that consumes information from lower-level reflective interfaces and implements the higher-level interface. We have developed a prototype implementation of such a system that is capable of translating the relationship between programming models into mirroring systems.

Our generator is validated in a middleware case-study. The generated inspection system automatically collects information from multiple machines, to offer a view on a collection of distributed objects that interact using remote method invocation (RMI hereafter). To enable intuitive inspection, distribution is made transparent, enabling navigation through remote relations with the same ease as local relations. Also distributed stack traces, that span multiple VMs, are represented as if they are local.

The remainder of this paper is structured as follows. The next section elaborates on the problem of inspection of middleware based applications. Section three discusses the requirements for middleware inspection. Section four gives a detailed overview of our solution. In section five, we validate our solution with four inspection cases and evaluate the performance overhead. Section six describes the related work and section seven concludes.

2 Problem Illustration

Application developers and integrators use middleware as development platform and abstraction layer. They use the advanced functionality provided by the middleware, but they are unaware of its inner workings. Middlewares and programming languages offer powerful abstractions, that allow programmers to focus on functionality while making abstraction of technical complexity.

However, when a middleware based software system is deployed, the middleware abstractions are no longer visible. The middleware and the application are composed together into a single synthetic system. The abstractions provided by the middleware are are no longer visible in the run-time structure. When inspecting the run-time structure, the programmer is faced with its full complexity.

When performing detailed inspections, with a monitoring or debugging tool, the developer or operator is faced with information in terms of the language abstraction. The middleware is no longer an abstraction layer, but a complex synthetic structure. The application is no longer represented in terms of middleware abstractions, but entangled in a complex low-level system.

For distributed middleware, the situation is further complicated by distribution. The run-time structure is not only hard to understand, but it is also scattered over different machines. When the middleware is capable of automatic deployment, without human interaction, it is not even known up-front which information is located where. Only the middleware itself knows where the various components have been deployed.

As an example, we will look at remote method invocations in Java. RMI creates a notion of distributed objects and distributed threads. While this is a very basic middleware feature, its run-time structure is already hard to understand without tool support. A distributed object consists of a stub and a proxy. The stub listens on a network port for remote calls. When it receives a remote call, it passes them on to an actual local object. A proxy acts as a local object, but sends all calls it receives over the network to its stub. To support remote invocations, proxy objects are passed on between the different hosts. When a method is invoked on a proxy, the stub on the remote host is contacted and a thread is created on the remote side of the call. This thread receives the request and invokes the correct local method. The caller thread is blocked until the remote call is complete. As such RMI creates many threads, on different machines. What looks like a single thread of execution to a programmer is actually a collection of many different threads on different machines.

In the current state of the art, several tools exist to examine the state of RMI applications. Tools such as JMX for example can provide general overview information, such as the number of threads, memory usage and garbage collector information. A Java debugger can be used for detailed inspection of the individual nodes.

However, there is currently no tool capable of presenting a detailed overview. When an RMI application exhibits undesired behavior, there is no convenient way to inspect it. Current tools don't support distributed objects or distributed

logical threads of execution through the various machines. In practice it would require manually decoding all stack traces on all machines, to find the various local threads that make up the distributed stack trace. Conversely, there is no convenient way to find a stub for a specific proxy.

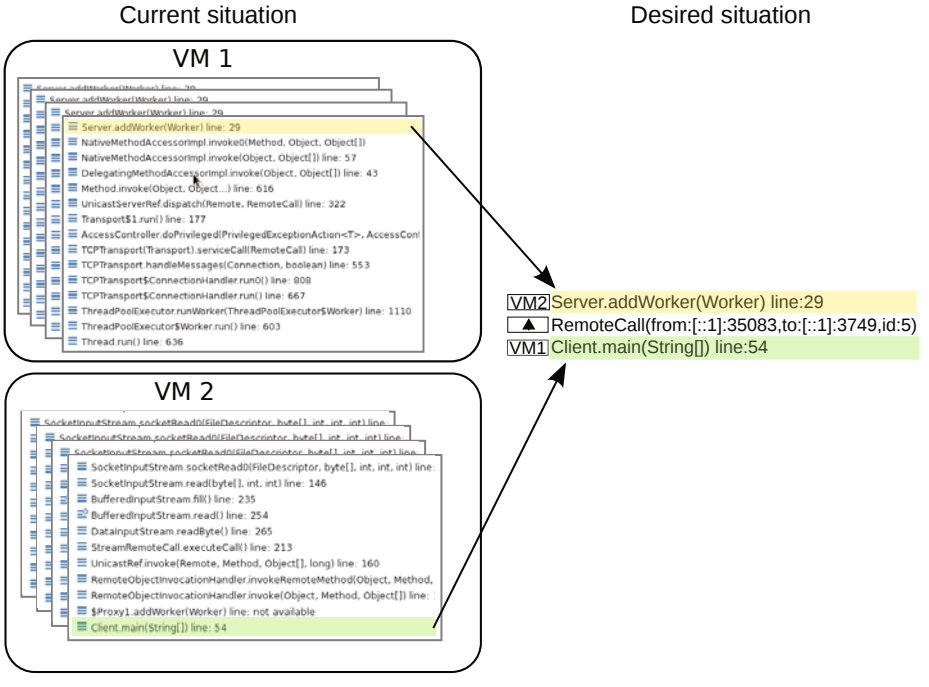


Fig. 1. example of an RMI stack trace

Consider Figure 1. On the right is the conceptual overview: a client performs a remote call to a server. On the left we see the view current tools can offer: on each virtual machine, there is a number of threads. Even when the correct pair of threads is isolated, the view is still polluted with synthetic code. In this case, each of the stack traces contains only one line of actual application code. Furthermore, the information of where the call comes from and where it goes to is not apparent from these synthetic stack traces.

3 Requirements and Approach

When inspecting middleware based applications, the abstraction offered by the middleware should be maintained. For RMI in particular, this means that remote objects and logical distributed threads should be visible.

However, it is also important to acknowledge that openness to inspection is not the core functionality of middleware. Inspection tools should not place a

burden on the normal structure and operations of the middleware. As such, the inspection tools should not overly influence the middleware's core design or operations.

As such we propose three requirements:

1. *Information should be extracted from existing sources.* By reusing existing sources of inspection information, we make sure the middleware's execution structure can be optimized for concerns other then inspection. It enables creation of inspection tools for middlewares that are already in production. No extensive instrumentation or modification is required to support inspection.
2. *What is to be inspected should be specified independent of how it should be inspected.* Middleware experts are not necessarily inspection experts. As such, a middleware expert should be capable of conveying his knowledge about the middleware abstractions without being concerned with the technicalities of distributed inspection. When a middleware expert writes an independent specification of the abstractions, he has the freedom of expressing all facts he knows, without having to think about the efficiency of the resulting inspection tool.
3. *Lazy inspection should be supported.* Inspection should interfere with normal operations as little as possible. Inspection tools should support precise scoping, in which only the required information is extracted. Detailed inspection often focuses on a specific part of the system and explores from there outwards. To support such a restricted focus, the system should support dynamic, on-demand inspection. It should only inspect the part of the system that is requested by the user.

To fulfill these requirements we propose the use of model transformations to build abstract views on top of existing inspection tools. Model transformations enable declarative specification of the relation between the existing inspection interfaces and the desired inspection interfaces in a declarative and natural way. In our solution, these specifications can be automatically converted to middleware components that support dynamic inspection.

As such, our approach for presenting the run-time state leverages on a declarative specification of the transformation that restores the middleware abstractions. This specification is purely declarative and free of technical details about run-time inspection. It describes how the run-time structure, that can be observed through existing inspection interfaces, relates to the conceptual structure, that we want to observe. The declarative specification is automatically converted into an implementation of the high-level reflective interface as a component that consumes a lower-level reflective interface and provides a high-level reflective interface.

Our approach can be divided into four steps.

1. **Modeling:** the existing inspection interfaces and the desired high-level inspection interface are represented by models. For the low-level interface this is usually a trivial conversion of the existing interface into a textual model. Modeling the desired high-level interface requires some design efforts, as they

define which information should be presented to operators or developers. For more information about the design of such interfaces, we refer to Bracha et al [1].
2. **Intermodeling:** the relations between the low-level source models and the high-level target model are specified as a model-to-model transformation. This requires a very precise understanding of the run-time structure of the middleware. However, it requires no special knowledge about reflective systems.
3. **Generation:** the model-to-model transformation is automatically converted into an inspection component that consumes low-level inspection information and produces the higher level information.
4. **Deployment:** the generated component is connected to the actual system.

In the intermodeling phase, a model-to-model transformation language is used to relate the existing structure to the desired structure. Generic model transformation systems exist in the form of rule engines and model-to-model transformers [19,21,4,19,11]. However, both types of systems have no support for lazy execution. These existing systems fundamentally assume that the entire system must be transformed at once. These systems apply an eager strategy, that makes lazy evaluation impossible. Due to the size of the run-time state of software systems, this eager strategy is too slow to support effective reflective transformations. *As such, we base the syntax and semantics of our transformation on the existing QVT-r language[19], but provide an alternate, lazy execution strategy.* We named this QVT-r dialect dynamic QVT-r or QVT-dr.

4 Detailed Solution

This section explains the technical details of our solution. First the declarative description of model-to-model transformations is presented. Then, we describe how such a declarative specification can be converted to an executable form. Finally, we discuss the advantages of this approach.

4.1 Declarative Specification of Model to Model Transformations

In general, a model-to-model transformation expresses the relations between a source model and a target model. The model transformation defines how entities in the source model are related to entities in the target model and vice versa (See Figure 2). For our approach, the source model is an existing inspection system. The source meta-model is the interface of this inspection system. In analogy, the target model is the inspection infrastructure we wish to provide. Its interface is described by the target meta-model. The transformation definition describes the relation between the two interfaces, while the transformation engine is the component we generate, which implements the target model by consuming the source model.

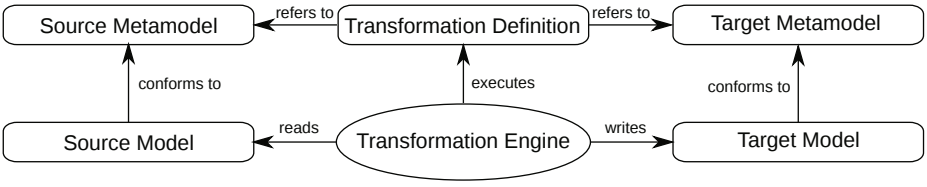


Fig. 2. Overview of model to model transformations. Based on [4]

Meta-modeling. More concretely, for RMI, the *source meta-model* is the Java Debugging Interface, JDI [27,1]. It contains entities such as classes, objects, threads and stack frames. Each of these entities has properties. Classes for example have a name, instances and a reference to their virtual machine. The *source model* is an instance of JDI, that connects to an actual running Java virtual machine (JVM). It provides instances of the types defined in the meta-model, which represent the actual classes and the actual objects present in that JVM. The *target meta-model* is the remote java debugging interface (RJDI), which contains all entities present in JDI, but also all RMI abstractions, such as stubs, proxies and distributed logical threads. Like in the source-meta model, the entities have properties. Proxies for example have an associated stub, instances and a reference to their virtual machine. The *target model* is provided by our generated inspection component: it implements the RJDI interface, based on the JDI interface. The model transformation itself defines how the RJDI interface can be implemented.

Model Transformations. A declarative model transformation describes all relations between elements in the source and target model. Any QVT-(d)r transformation consists of a set of relations, where each relation models the relation between two specific entities. A relation defines the conditions an entity in the source model must fulfill to be transformed into a specific entity in the target model. Listing 1.1 shows the relation between Java (JDI) classes and RMI (RJDI) proxy-types.

To relate both entities, all their properties are bound to a set of shared variables. All variables bound in the target model can be derived from the variables bound in the source model. The relation also defines a set of preconditions to which the source entity must comply. These preconditions are demarcated with the keyword `when`. When an entity of the correct type is found in the source model for which all preconditions hold, we say the relation holds.

If the relation holds, the target model must contain the target entity. Furthermore, the relation can also define a set of post-conditions (demarcated with the keyword `where`). If the relation holds, all post-conditions must hold.

As such, Listing 1.1 defines a single relation, relating two entities. It states that if an entity of type `ClassType` exists in Java (JDI), and it has `$Proxy` in its name and its superclass has as name `java.lang.reflect.Proxy`, then this entity corresponds to an `RmiProxyType` in RMI (RJDI), which belongs to the

```
1  relation ObjectReferenceTypeToProxyType{
2    domain JDI in:ClassType{
3     instances = instI;
4    };
5    domain RJDI out:RmiProxyType{
6     instances = instO;
7     //other properties omitted
8    };
9    when{
10     VmtoVm(in.virtualMachine,out.virtualMachine);
11     in.name.contains("$Proxy");
12     in.superclass.name = "java.lang.reflect.Proxy";
13     //more complex preconditions omitted
14   }
15   where{
16     ObjectReferenceToProxy(instI,instO);
17     //additional post-conditions omitted
18   }
19 }
```

Listing 1.1. Concrete example of a model transformation relating RMI-proxies to their Java equivalents

corresponding virtual machine. The instances of the `RmiProxyType` can be derived through the `ObjectReferenceToProxy` relation.

In this example, we omitted the more complex pre- and post-conditions that are used to extract more information from the middleware, such as how to find the stub associated with this proxy. These parts of the pattern are analogous to what is already presented, but require a more intimate knowledge of the internals of RMI.

Using this approach, a description of the run-time structure of the most important RMI concepts is created. Stubs and proxies can be found based on these patterns both on the heap and on the stack. They are transformed into a representation that hides their internal complexity but exposes their internal state.

4.2 Dynamic Execution of Declarative Model Transformations

Such declarative model transformations are not directly executable. The next section describes how the declarative specifications can be converted to an executable form. It describes the internal mechanism of the generator that transforms the declarative specification to an inspection component that dynamically and lazily transforms low-level information.

The model transformation defines how information flows between the source and target model. As such, the main task of the generator is to infer an implementation for all operations of the target model, based on the operations in the source model, in such a way that lazy evaluation is supported. The generator is based on a four step process.

1. **Parsing.** The model transformation definition and related meta-models are compiled into a data-flow graph and type-checked. The data-flow graph directly represents all relations defined in the transformation definition.
2. **Inference of the control flow.** Directions are added to the flow of information. Given the fact that the input node is known and the characteristics of all other nodes, a sat solver is used to compute all valid flows of information through the model. If multiple alternatives exist, a heuristic is used to choose the most optimal.
3. **Model partitioning.** The graph is partitioned into three parts: a part containing all preconditions, a part containing all postconditions and a part containing the rest.
4. **Code generation.** Based on the partitioned information flow graph, code is generated.

The remainder of this section provides more information about these steps.

Parsing and Checking. The declarative model transformation definition is parsed into an abstract semantic graph (ASG). The ASG is a typed and labeled graph (e.g. Figure 3), representing the structure of the source meta-model, target meta-model and the transformation between them. First we discuss the general structure of the ASG and then illustrate it based on the ASG segment in Figure 3.

In general, for each relation (such as the one in Listing 1.1), each entity presented in it becomes a pattern node in the ASG. Each pattern node is also bound to a node in the source or the target meta model that indicates its type. All relations between pattern nodes become edges in the ASG. The type of the edge indicates the type of the relation. Entities bound to a variable have their pattern node bound to a variable node.

For example, consider figure 3. It represents the following part of Listing 1.1:

```
in:ClassType{virtualMachine=temp1,instances=instances}.
```

The variable `temp1` is implicit in Listing 1.1. In this pattern, the variable `in` is bound to a pattern node of type `ClassType`. This node is indicated with a bold border. The type `ClassType` has three operations: `virtualMachine`, `instances` and `name`, with as types respectively `VirtualMachine`, `ObjectReference` and `String`. The pattern binds the operations `virtualMachine` and `instances` to pattern nodes that are bound to the named variables `temp1` and `instances`.

Inference of the Control Flow. To support the generation of an implementation, the direction of the information flow through the pattern must be inferred. When considering individual statements, information can flow in either direction. For example: in Listing 1.1 line 3 and 6 are identical statements but information flows through them in the opposite direction. Line 3 assigns the value of `in.instances` to the variable `instI` while line 6 provides a result for the operation `out.instances` through the value of `instO`.

As such, the information flowing through any node depends on the flow through any other node in the same relation. To derive a valid information

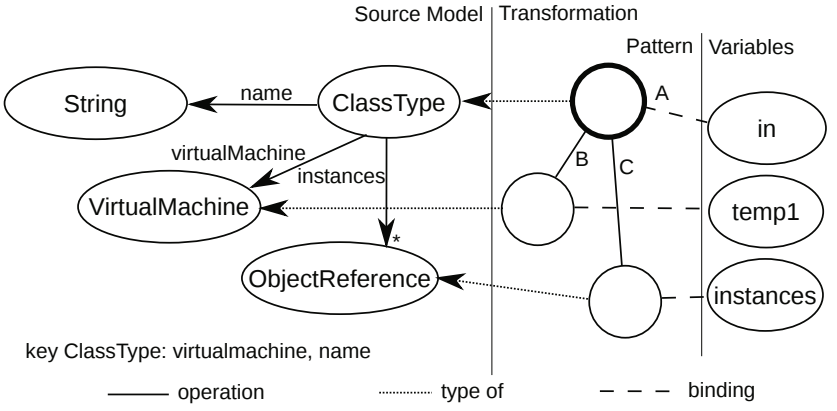


Fig. 3. Part of the ASG

flow, the flow analysis adds a direction to each edge, indicating the flow of information. The analysis is based on a model that defines when a node in the pattern has sufficient information flowing in to calculate all other edges.

While information flow is best explained in terms of graphs, it is more convenient to use a SAT solver to efficiently derive an optimal information flow. Therefore the ASG model is translated into logic predicates. Each edge becomes a boolean variable indicating the flow of information. Each node becomes a list of predicates, defining when the node has sufficient incoming edges to calculate all other edges.

For example, take the pattern from Figure 3. The bold node is defined if either 1) information is flowing in from the variable `in` (i.e. the entity is defined elsewhere) or 2) all it's operations (`virtualMachine`, `name` and `instances`) are defined (i.e. a complete definition of the entity is present and it can be constructed here). However, in this case the operation `name` is not bound in the pattern. As such the overall logical predicate becomes $(false \wedge B \wedge C) \vee A$, when we assume *true* means *the edge is incoming*. As a consequence A must be true and the corresponding edge must be incoming. The bold node thus receives information from the variable `in`.

For this simple example pattern, there is only one possible information flow (A is incoming). However, in general, each node can have multiple solutions. This require a more global analysis process, taking into account all nodes in the relation. By converting all nodes and edges into predicates, a SAT solver can be used to derive all valid information flows through the relation. A search heuristic can then be used to select one solution.

Model Partitioning. In the information flow graph, some nodes in the pattern have more incoming edges then strictly required. We call these nodes overconstrained. For example, on line 11, `in.superclass.name` is defined because the variable `in` is defined. It is also defined because it is bound to a constant. When information flow is overconstrained, pattern matching may fail on that node. i.e.

If both sources of information produce a different value, the relation doesn't hold. Any overconstrained node forms a condition that must be checked to determine if the relation holds or not.

For code generation, overconstrained nodes must be identified. Therefore, the ASG is partitioned. First it is divided into conditional nodes and non-conditional nodes. Conditional nodes are either overconstrained or they directly depend on an operation that may fail, such as a call to another relation. The conditional nodes are then partitioned again into nodes that have been marked as assertions and others.

In this way, there are three partitions: the guard nodes (conditions that are not assertions), the assertion nodes and the non-conditional nodes. The guard nodes are all the nodes that must be checked to see if the pattern matches. These nodes (and all nodes providing information to them) must always be evaluated eagerly (and thus not lazy).

The assertion nodes can be discarded, as the assertion should always hold. However, they can also be passed to the code generator, to produce more robust code, that checks all (or some) assertions.

Code Generation. In the implementation (see Listing 1.2) each relation becomes a method, that takes as an argument a source-model entity. When the method is called, all conditions in the guard partition are checked. If all conditions hold, the relation holds. Then an object of the desired target type is constructed. Each method of this object corresponds to an operation in the target model. Each operation contains the part of the pattern that provides the operation according to the inferred control flow. In practice, most operations use other patterns to create other target-model entities. As such, when a first target-model entity has been created, the rest of the target model can be explored using its operations. Each operation will lazily collect entities from the underlying inspection interface, as required by the patterns.

4.3 Discussion

Our approach enables automatic generation of an inspection component out of a declarative specification. Apart from the earlier mentioned requirements, it has two important advantages:

1. Overdetermined specifications don't result in a slower system. When a model transformation defines many ways of deriving any given operation, this doesn't make the execution of the pattern less efficient. The generator can choose any sufficient implementation, while discarding redundant information. At the other hand, the generator can also be configured to check all assertions. As such, the generator can create either more robust or more efficient code, without any manual rewriting.
2. Different tools can reuse the same transformation. As in any compiler, the use of a central intermediate representation decouples three roles: language user, optimization writer and back-end developer. The important consequence is

```
rmi.RmiProxyType objectReferenceTypeToProxyType(jdi.ClassType in){
 rmi.VirtualMachine temp1 = vmToVm(in.virtualMachine);
 if(temp1 == null) return null;
 if(!in.name().contains("$Proxy")) return null;
 if(!in.superclass().name().equals("java.lang.reflect.Proxy"))
  return null;
 //more complex guards omitted
 return new ObjectReferenceTypeToProxyType(in,temp1);
}
class ObjectReferenceTypeToProxyType implements rmi.RmiProxyType{
 private jdi.ClassType in;
 private rmi.VirtualMachine virtualMachine;
 private List<rmi.ObjectReference> instances;

 ObjectReferenceTypeToProxyType(jdi.ClassType in,
   rmi.VirtualMachine virtualMachine){
       this.in = in;
       this.virtualMachine = virtualMachine;
 }

 public rmi.VirtualMachine virtualMachine(){
  return virtualMachine;
 }

 public List<rmi.ObjectReference> instances(){
  if(instances != null)
    return instances;
  instances = ObjectReferenceTypeToProxyType(in.instances());
  return instances;
 }
 //other properties omitted
}
```

Listing 1.2. Implementation of the pattern in Listing 1.1 without assertions

that replacing the back-end stages yields a different type of inspection infrastructure. Inspection can be used in-program (reflective), but also for debugging or even post-mortem debugging (debugging of systems that have already crashed). Each of these styles requires a very different tool, but suffers from the same abstraction gap. If tools are required to support N languages and M styles, it is no longer necessary to build M*N tools, but N specifications and M back-ends. By decoupling the back-end and front-end the complexity of the problem has been reduced from multiplicative to additive. Furthermore, as any component is reused more often, it will mature faster.

5 Evaluation

To validate the use of such a high level inspection tool, we use an example application and compare inspecting it with existing tools against our solution. First, a number of use cases are discussed, then we evaluate the performance overhead of our solution.

The application is a work scheduling server. Jobs, consisting of several tasks are queued on the server. The server then schedules the tasks on worker nodes. The server also passes a callback remote object to the worker, by which the worker can report its progress. When a worker node has completed its task, it

signals the server through the callback. The server then schedules the next task in the job on a worker.

On the application, we test four inspection scenarios.

1. **A control flow problem:** when a worker signals a task is done, the next task is scheduled from within that thread. This causes all tasks in the job to be in the same logical thread of execution. The logical thread starts from the server, then goes to the first worker node and then back to the server, then back to the next worker and so on. This means that each job consumes 2n+1 threads, with n the number of tasks in the job. This has two side effects: the server and worker consume a massive amount of threads and network sockets and, when one worker node fails, all jobs that used this worker before crash on completion.
2. **An information flow problem:** when a worker receives a monitor callback, it exports the callback. When a job is done, it returns the callback to the server. The server then hands this callback to the next worker. Conceptually, this is the same callback-reference. However, because the callback has been exported on the worker, all calls to the remote object are routed over the worker node. This makes message propagation slow and very sensitive to failure of worker nodes.
3. **A deployment problem:** a worker node has been deployed to a wrong host. This causes two worker nodes to share the same virtual machine. At application level the workers look different. RMI makes abstraction of distribution, so the server doesn't know they are on the same host.
4. **The cost of inspection:** different work scheduling servers are working for different organizations. They share the same infrastructure, but for security reasons, different servers should never share a worker node. Regular audits must ensure this.

The control flow problem has no apparent symptoms, until a worker node is taken out of the schedule and powered down. After some time, jobs start to fail unexpectedly. When connecting a local debugger to the server and browsing through all threads present, we find that many threads have a similar structure. On closer examination, we find that they share the segment depicted in Listing 1.3. On the worker nodes, a similar stack trace is found (Listing 1.4). A very experienced RMI developer may conclude from this information that the control flow is going back and forth between client and server. This conclusion is however far from obvious.

When using our generated inspection tool we can investigate the distributed logical stack trace from one of the jobs (Listing 1.5). This immediately shows that the distributed stack trace spans many different nodes. The individual frames can be inspected to trace this control flow and learn to understand what causes this behavior.

Similar to the first problem, the second problem has no apparent symptoms, until a worker node is taken out of service. However, in this case the local stack trace provide no clues. In the previous example, the long stack traces had a long lifetime. This made sure many abnormal stack traces were present on any

```
... (8 more frames)
RemoteObjectInvocationHandler.invoke()
$Proxy1.run()
MonitorImpl.runOn()
MonitorImpl.done()
GeneratedMethodAccessor5.invoke()
DelegatingMethodAccessorImpl.invoke()
 Method.invoke()
UnicastServerRef.dispatch()
... (13 more frames)
```

Listing 1.3. Server side stack trace caused by the control flow problem

```
... (8 more frames)
RemoteObjectInvocationHandler.invoke()
$Proxy2.done()
Client.run()
NativeMethodAccessorImpl.invoke0()
... (13 more frames)
```

Listing 1.4. Client side stack trace caused by the control flow problem

machine, making it easy to find them. However, in this case they are very short lived. When inspecting any local node, one is unlikely to find any abnormal stack traces. Also, these stack traces contain no application code. Placing break points in the application doesn't help in finding abnormal stack traces. Without appropriate tools, the only possibility of finding the root cause is source code analysis.

With our tool, when plotting all remote objects present in the system, it is immediately clear that most worker nodes are not directly referring to the server, but referring to other worker nodes (Figure 4). This can only be caused by a reexport of the remote reference on the worker nodes.

```
vm3 Client.run()
vm3 STUB 192.168.150.32:38175 57769 -7666749065146411512
vm1 PROXY 192.168.150.32:38175 57769 -7666749065146411512
vm1 MonitorImpl.runOn()
vm1 MonitorImpl.done()
vm1 STUB 192.168.150.31:39728 36651 -2921635400086109349
vm2 PROXY 192.168.150.33:39728 36651 -2921635400086109349
vm2 Client.run()
vm2 STUB 192.168.150.33:60307 52965 -2670173772187310440
vm1 PROXY 192.168.150.33:60307 52965 -2670173772187310440
vm1 MonitorImpl.runOn()
vm1 MonitorImpl.run()
vm1 MonitorImpl.run()
```

Listing 1.5. Stack trace of failing application

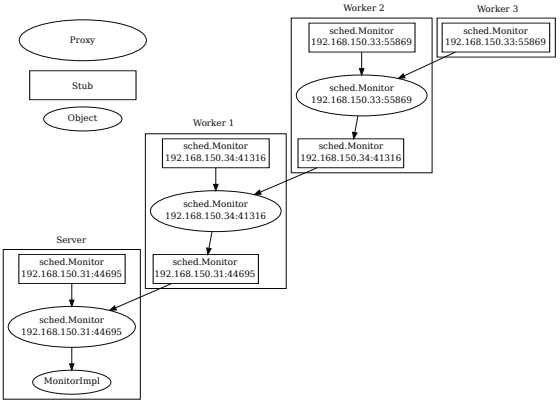


Fig. 4. Remote objects with information flow problem

The third bug has no immediately apparent symptoms. When looking at the load on the different hosts, it will be clear that one host is less heavily loaded. When looking at the performance of all hosts, two hosts will be underperforming. Putting these two facts together one may conclude that a node has been deployed on the wrong host.

When plotting the remote objects with our inspection component (Figure 5), it is immediately clear the server references two workers on the same host. It is also clear on which host the nodes are deployed.

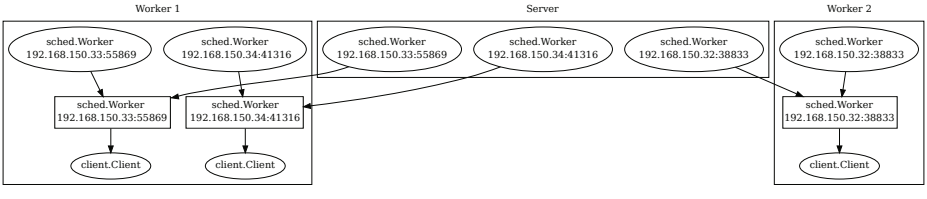


Fig. 5. Remote objects with deployment problem

The fourth problem is not a bug as such. It is an auditing requirement. When using ordinary inspection techniques, it is not cost efficient for an auditor to regularly dump the state of the application and start digging through in the hope of finding an irregularity in the communications pattern. Currently, two alternate solutions exist. The first solution would be to physically separate both infrastructures. This is more expensive, due to a lack of resource sharing. Alternatively, the server could be adapted to maintain an explicit list of hosts on which its workers are located. However, as RMI is location transparent, this would preclude the use of RMI. When using our inspection component, the physical placement and relation between hosts can be audited easily, as shown in the previous example.

To evaluate the performance overhead, we set up the following worst case scenario: worker nodes with a CPU intensive workload were subjected to continuous monitoring. Our set up consisted of one server with three worker nodes – all Pentium III single core machines. The inspection server continuously searched for all proxies and stubs on all machines and retrieved all their attributes. Once all information was retrieved, the cache was cleared and the search restarted. With all nodes executing CPU intensive tasks, the performance overhead was 34% compared to the case with no inspection. When the worker nodes were given less CPU intensive tasks, the performance overhead dropped under 1%.

6 Related Work

This work is founded in the reflection and model driven development (MDD) communities. The related work around MDD has already been briefly highlighted in Section 4. In this section, we highlight some influential work from the field of *reflection* and compare our approach to existing inspection approaches for *monitoring*, *debugging* and *reverse engineering*.

Reflection. Reflection is the ability of software systems to reason about and act on themselves. It encompasses inspection, but also self-modification and metaprogramming. Our approach is inspired on the design principles for reflective systems, defined by Bracha et al [1]. These principles define that a reflective system should have ontological correspondence to the system it reflects on and that reflective systems should encapsulate their implementation.

Ontological correspondence means that the reflective interface should be structured according to the abstractions of the system it reflects on. This is also one of our key requirements for middleware inspection: all the middleware abstractions should be maintained when inspecting distributed software systems.

We also choose for strong encapsulation of the implementation. Our approach requires no modification of the underlying middleware and puts no constraints on the middleware's execution structure. The implementation of the inspection component is the model transformation, which is declarative and completely separated from the middleware. However, if the middleware has no interface or fixed internal structure, the inspection component may break when the middleware evolves. As such our approach doesn't require a stable interface, but it is more stable when a stable interface exists.

Reflective middleware systems [14,31,3] have the capability of reflecting on the middleware structure itself. This reflection goes beyond inspection and supports run-time adaptation of the middleware. In such middleware, the reflective infrastructure is always present. This adds a constant overhead to the execution. Reflective middleware systems can serve as sources of information for our approach. They can be used in the way we used JDI in this paper.

Monitoring. Monitoring systems keep track of a limited set of inspection targets over a long period of time. Monitoring consists of two main activities: information extraction and information aggregation.

The most common way of extracting information is *built-in monitoring*. The system is modified by hand to emit events that signal important changes [26]. The advantage of this approach is its simplicity, while the disadvantage is that the monitoring system always incurs an overhead – as it is part of the system. It can not evolve independently or be adapted at run-time. As such, this approach is used to expose small volumes of high-level information. When the middleware actively supports the emitting of events, such as in Google's Dapper [24], the event streams of different hosts can be put together to create a distributed trace. The information extracted from such monitoring probes can also be aggregated. For statistical aggregation of monitoring data, collection systems are already widely deployed [7,34].

A second way of extracting monitoring information is *instrumentation*. Instrumentation systems automatically modify a program so that it emits events. This enables dynamic fine-tuning of the monitoring and its associated overhead. However, dynamic deployment of monitoring probes is a technically complex operation, that requires support of the underlying platform. For many languages instrumentation support exists [8,18,20].

An advanced proponent of the instrumentation approach is described in [17]. This monitoring system dynamically instruments code ahead of the flow of control. It has properties comparable to our approach. Monitoring components are developed separately and deployed on demand. Also the performance overhead seems to be comparable. The major difference is that our system only supports structural inspection while Mirgorodskiy et al. only support event based inspection. As such, their system is capable of tracing change very efficiently, but incapable of inspecting state that doesn't change. Our system has the inverse properties: it can inspect existing state in great detail, but is incapable of perceiving rapid change. However, in the future we aim to integrate events into our model transformation approach.

Persistent query systems are another possible form of information aggregation. A query system is a reflective component that allows external systems to query its own state. A *persistent* query system is a query system capable of keeping the results of its queries up-to-date when the underlying system changes. It provides a form of continuous reporting [22,13,29].

Debugging. Debugging means searching for and remedying of software faults [35]. Interactive inspection is an important component of debugging, but not the only one. It also encompasses methodology, tools for automatic and semiautomatic detection, prevention and removal of faults as well as edit-and-continue technology. [23,35,8,33,32,12,25]

The current generation of inspection tools for debugging consists of two categories: debuggers for languages with a custom VM and debuggers for compiled languages. Debuggers for languages with a custom VM or languages with a strong reflective system provide debugging facilities by exposing their internal data structures. This provides a view of the running program in terms of the abstractions supported by the VM. Without need for transformations, the VM natively supports all abstractions.

For compiled languages, transformations are always required. The state-of-practice for such languages is to write the required transformations by hand. This lack of a disciplined approach – combined with the inherent complexity of pattern matching code – limits the capabilities of current debuggers. GDB [9], for example, is still unable to decode the heap of C programs. However, recently, efforts are being made to isolate the pattern matching into separate modules, to enable heap decoding [15].

For middleware few debuggers exist. One notable exception is a distributed debugger constructed by Mega and Kon [16]. It offers support for distributed logic threads, similar to our approach. As most debuggers, its transformation components have been built by hand, supporting both structural and event based inspection.

For a more elaborate explanation about the design trade-offs for the construction of higher level debuggers, we refer to [16,5].

Reverse Engineering. Reverse engineering (RE) tools enable dynamic rebuilding of software abstractions. RE tools rely on advanced visualization [6] and combined static and dynamic analysis [28,30,2]. From a modeling perspective, reverse engineering mostly uses containment relations. For example, instructions are grouped into blocks, blocks into methods, methods into classes. Currently, most RE tools use an event based approach for dynamic analysis. However, our generator makes state transformation components easier to build and may enable RE systems to incorporate them.

7 Conclusion

We presented an approach enabling inspection of middleware with full support for all abstractions offered by the middleware, that requires no modification of the middleware itself and is capable of limiting its overhead by dynamic, on-demand transformation.

Our generator technology can be used to construct detailed inspection systems for middleware. It decouples the role of middleware expert and inspection expert, to support modular development of inspection tools. Middleware experts can express their knowledge in a declarative way. This declarative specification is automatically converted into a usable and efficient inspection component. The generator is capable of automatically removing redundant information from the specification and can switch between generating either more robust or more efficient inspection components.

We have demonstrated the advantage of maintaining the middleware abstraction when inspecting in four use cases. We also showed that the overhead of the inspection system is acceptable, even in a worst case scenario.

In the future, we will focus on automatic generation of more complex inspection tools. We aim to add support for events in our generic inspection approach to detect run-time changes and act upon them. We will also integrate our approach into an IDE to enable broader, user-driven evaluation. Further validation and

evaluation of our approach in a broader monitoring and run-time management context will also provide more metrics about the effectiveness of lazy execution and the use of overdetermined specifications.

References

1. Bracha, G., Ungar, D.: Mirrors: design principles for meta-level facilities of object-oriented programming languages. In: Proc of OOPSLA, pp. 331–344 (2004)
2. Chan, A., Holmes, R., Murphy, G.C., Ying, A.T.T.: Scaling an object-oriented system execution visualizer through sampling. In: 11th IEEE Intl Workshop on Program Comprehension, pp. 237–244 (2003)
3. Clarke, M., Blair, G.S., Coulson, G., Parlavantzas, N.: An efficient component model for the construction of adaptive middleware. In: Proc of IFIP/ACM Intl Conf on Distributed Systems Platforms Heidelberg, pp. 160–178 (2001)
4. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–645 (2006)
5. De Borger, W., Lagaisse, B., Joosen, W.: A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. In: Proc of AOSD, pp. 173–184 (2009)
6. De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J., Yang, J.: Visualizing the Execution of Java Programs. In: Diehl, S. (ed.) Dagstuhl Seminar 2001. LNCS, vol. 2269, pp. 151–162. Springer, Heidelberg (2002)
7. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Transactions on software Engineering 30(12), 859–872 (2004)
8. Eaddy, M., Aho, A., Hu, W., McDonald, P., Burger, J.: Debugging Aspect-Enabled Programs. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 200–215. Springer, Heidelberg (2007)
9. Free Software Foundation, Inc. Gdb: The gnu project debugger (July 2009), http://www.gnu.org/software/gdb/
10. Jordan, D., and Evdemon, J. Web services business process execution language version 2.0 (April 2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
11. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
12. Ko, A.J., Myers, B.A.: Designing the whyline: a debugging interface for asking questions about program behavior. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 151–158 (2004)
13. Ko, S.Y., Yalagandula, P., Gupta, I., Talwar, V., Milojicic, D., Iyer, S.: Moara: Flexible and Scalable Group-Based Querying System. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 408–428. Springer, Heidelberg (2008)
14. Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., Magalhães, L.C., Pasquale, F.: Monitoring, Security, and Dynamic Configuration with the *dynamicTAO* Reflective ORB. In: Coulson, G., Sventek, J. (eds.) Middleware 2000. LNCS, vol. 1795, pp. 121–143. Springer, Heidelberg (2000)
15. Malcolm, D.: gdb-heap, https://fedorahosted.org/gdb-heap/ (access: Febuary 17, 2011)
16. Mega, G., Kon, F.: An Eclipse-Based Tool for Symbolic Debugging of Distributed Object Systems. In: Meersman, R. (ed.) OTM 2007, Part I. LNCS, vol. 4803, pp. 648–666. Springer, Heidelberg (2007)

17. Miller, B.P., Mirgorodskiy, A.V.: Diagnosing Distributed Systems With Self-Propelled Instrumentation. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 82–103. Springer, Heidelberg (2008)
18. Navarro, L.D.B., Douence, R., Südholt, M.: Debugging and Testing Middleware With Aspect-Based Control-Flow and Causal Patterns. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 183–202. Springer, Heidelberg (2008)
19. OMG. Meta object facility (mof) 2.0 query/view/transformation, http://www.omg.org/spec/QVT/1.0/
20. Oracle. Java instrumentation, http://download.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html (access May, 2011)
21. Proctor, M. Drools documentation library (August 2009), http://www.jboss.org/drools/documentation.html
22. Rajamani, V., Julien, C., Payton, J., Roman, G.C.: Paq: Persistent Adaptive Query Middleware for Dynamic Environments. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 226–246. Springer, Heidelberg (2009)
23. Rosenberg, J.B.: How debuggers work: algorithms, data structures, and architecture. John Wiley & Sons, Inc., New York (1996)
24. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a large-scale distributed systems tracing infrastructure. In: Google Research (2010)
25. Silva, J.: A Comparative Study of Algorithmic Debugging Strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)
26. Sun Microsystems, I. Java management extensions (August 2009), http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/
27. Sun Microsystems, I. Java platform debugger architecture (June 2009), http://java.sun.com/javase/technologies/core/toolsapis/jpda/
28. Systä, T., Koskimies, K., Müller, H.: Shimba—an environment for reverse engineering java software systems. Software: Practice and Experience 31(4), 371–394 (2001)
29. Van Renesse, R., Birman, K.P., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Transactions on Computer Systems (TOCS) 21(2), 164–206 (2003)
30. Walker, R.J., Murphy, G.C., Freeman-Benson, B., Wright, D., Swanson, D., Isaak, J.: Visualizing dynamic software system information through high-level models. ACM SIGPLAN Notices 33(10), 271–283 (1998)
31. Wangham, M.S., Lung, L.C., Westphall, C.M., Fraga, J.S.: Integrating ssl to the jacoweb security framework: project and implementation. In: Proc of IEEE/IFIP Intl. Symp. on Integrated Network Management, pp. 779–792 (2001)
32. Weiser, M.: Program slicing. In: ICSE 1981, pp. 439–449 (1981)
33. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. SIGSOFT Softw. Eng. Notes 30(2), 1–36 (2005)
34. Zanikolas, S., Sakellariou, R.: A taxonomy of grid monitoring systems. Future Gener. Comput. Syst. 21(1), 163–188 (2005)
35. Zeller, A.: Why programs fail: a guide to systematic debugging. Morgan Kaufmann (2009)

A Comparison of Secure Multi-Tenancy Architectures for Filesystem Storage Clouds

Anil Kurmus[1], Moitrayee Gupta[2,⋆], Roman Pletka[1],
Christian Cachin[1], and Robert Haas[1]

[1] IBM Research - Zurich
{kur,rap,cca,rha}@zurich.ibm.com
[2] Department of Computer Science and Engineering, UCSD
m5gupta@cs.ucsd.edu

Abstract. A filesystem-level storage cloud offers network-filesystem access to multiple customers at low cost over the Internet. In this paper, we investigate two alternative architectures for achieving multi-tenancy securely and efficiently in such storage cloud services. They isolate customers in virtual machines at the hypervisor level and through mandatory access-control checks in one shared operating-system kernel, respectively. We compare and discuss the practical security guarantees of these architectures. We have implemented both approaches and compare them using performance measurements we obtained.

1 Introduction

Storage cloud services allow the sharing of storage infrastructure among multiple customers and hence significantly reduce costs. Typically, such services provide object or filesystem access over a network to the shared distributed infrastructure. To support multiple customers or *tenants* concurrently, the network-filesystem-access services must be properly isolated with minimal performance impact.

We consider here a *filesystem storage cloud* as a public cloud storage service used by customers to mount their own filesystems remotely through well-established network filesystem protocols such as NFS and the Common Internet Filesystem (CIFS, also known as *SMB*). Such a service constitutes a highly scalable, performant, and reliable enterprise network-attached storage (NAS) accessible over the Internet that provides services to multiple tenants.

In general, a cloud service can be run at any of the following increasing levels of multi-tenancy:

- *Hardware level*: server hardware, OS, and application dedicated per client.
- *Hypervisor level*: share server hardware, and use virtualization to host dedicated OS and application per client.
- *OS level*: share server hardware and OS, and run a dedicated application per client.
- *Application level*: share server hardware, OS, and application server among clients.

⋆ Work done at IBM Research - Zurich.

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 471–490, 2011.

Intuitively, the higher the level of multi-tenancy, the easier it seems to achieve a resource-efficient design and implementation; at the same time, though, it gets harder (conceptually and in terms of development effort) to securely isolate the clients from each other.

In this paper, we investigate a *hypervisor-level* and an *OS-level* multi-tenant filesystem storage cloud architecture, and compare them in terms of performance and security. The hypervisor-level multi-tenancy approach is based on hardware virtualization (with para-virtualized drivers for improved networking performance). We refer to this architecture as the *virtualization-based multi-tenancy (VMT) architecture*. The OS-level multi-tenancy approach uses mandatory access control (MAC) in the Linux kernel and is capable of isolating customer-dedicated user-space services on the same OS. Such an architecture may also leverage, for instance, OS-level virtualization technologies such as *OpenVZ* or *Linux Containers (LXC)*. We refer to this architecture as the *operating-system-based multi-tenancy (OSMT) architecture* in the remainder of this paper.

We have implemented both approaches on real hardware in the *IBM Scale-out NAS (SONAS)* [1] and the *IBM General Parallel Filesystem (GPFS)* [2] technologies. We used open-source components such as *KVM* [3] with *virtio* networking for virtualization and *SELinux* (`http://selinuxproject.org/`) for MAC.

Section 3 describes the architecture of a filesystem storage cloud and introduces the two designs. Section 4 defines an adversary model and discusses the security of both architectures according to this model. Section 5 presents the implementation and benchmark results. Related work is discussed in Section 6.

2 Background

One can distinguish the following categories of general-purpose storage clouds (ignoring storage clouds that provide database-like structures on content):

- *Block storage clouds*, with a block-level interface, i.e., an interface that allows the writing and reading of fixed-sized blocks. Examples of such clouds include *Amazon EBS*.
- *Object storage clouds*, composed of buckets (or containers) that contain objects (or blobs). These objects are referred to by a key (or name). The API is usually very simple: typically a REST API with *create* and *remove* operations on buckets and *put*, *get*, *delete*, and *list* operations on objects. Example of such storage clouds include *Amazon S3*, *Rackspace Cloudfiles*, and *Azure Storage Blobs*.
- *Filesystem storage clouds*, with a full-fledged filesystem interface, therefore referred to also as "cloud NAS." Examples of such clouds include *Nirvanix CloudNAS*, *Azure Drive*, and *IBM Scale-Out Network Attached Storage (SONAS)*.

Application-level multi-tenancy is sometimes also referred to as native multi-tenancy. Some authors consider it the cleanest way to isolate multiple tenants [4]. However, achieving multi-tenancy securely is very challenging and therefore not common for filesystem storage clouds. The reasons lie in the complex nature of this task: unlike other types of storage clouds, filesystem storage clouds possess complex APIs that have evolved over time, which leads to large attack surfaces. The vulnerability track record of these applications seems to confirm this intuition. CIFS servers were vulnerable

to various buffer-overflows (e.g., CVE-2010-3069, CVE-2010-2063, CVE-2007-2446, CVE-2003-0085, CVE-2002-1318, see http://cve.mitre.org/), format string vulnerability leading to arbitrary code execution (CVE-2009-1886), directory traversals (CVE-2010-0926, CVE-2001-1162), while NFS servers were also vulnerable to similar classic vulnerabilities as well as more specific ones such as filehandle vulnerabilities [5]. Moreover, adding multi-tenancy support into these server applications would require significant development (e.g., in order to distinguish between different authentication servers for specific filesystem exports) which will most likely result in new vulnerabilities. We discuss in Sections 3 and 4 architectures with lower levels of multi-tenancy. They effectively restrict the impact of arbitrary code execution vulnerabilities to the realm of a single tenant: by definition, this cannot be achieved with application-level multi-tenancy.

This paper targets the IBM SONAS [1] platform, which evolved from the IBM Scale-Out File Services (SoFS) [6]. IBM SONAS provides a highly scalable network-attached storage service, and therefore serves as a typical example of a filesystem storage cloud. IBM SONAS currently contains support for hardware-level multi-tenancy according to the architectures discussed in this work. Adding a higher-level of multi-tenancy is an important step to reduce the cost of a cloud-service provider.

3 System Description

Section 3.1 gives an overview of the general architecture of a filesystem storage cloud. Section 3.2 describes the MAC policies which are used in both architectures. Sections 3.3 and 3.4 introduce the two alternatives, detailing the internals of the interface nodes, the key element of the filesystem storage cloud architecture.

3.1 General Description

Figure 1 depicts the general architecture of a filesystem storage cloud that consists of the following elements:

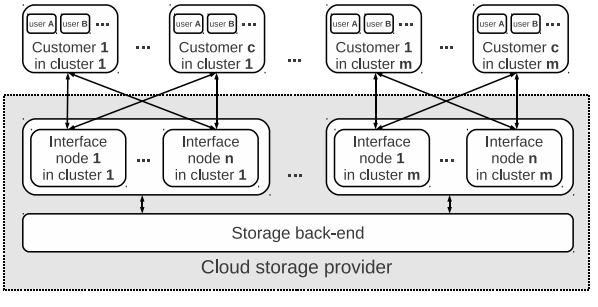


Fig. 1. General architecture of a filesystem storage cloud

- **Customers and users:** A *customer* is an entity (e.g., a company) that uses at least one network file system. A customer can have multiple individual *users*. We assume that multiple customers connect to the filesystem storage cloud and that each customer has a separate set of users. Data is separated between users from distinct customers, and user IDs are in a separate namespace for each customer. Hence two distinct customers may allocate the same user ID without any conflict on the interface nodes or in the storage back-end.
- **Interface nodes and cluster:** An interface node is a system running *filer services* such as NFS or CIFS daemons. Interface nodes administratively and physically belong to the cloud service provider and serve multiple customers. A customer connects to the filesystem storage cloud through the interface nodes and mounts its filesystems over the Internet. Multiple interface nodes together form an interface cluster, and one interface node may serve multiple customers. A customer connects only to nodes in one interface cluster.
- **Shared back-end storage:** The shared back-end storage provides block-level storage for user data. It is accessible from the interface clusters over a network using a distributed filesystem such as GPFS [2]. It is designed to be reliable, highly available, and performant. We assume that no security mechanism exists within the distributed filesystem to authenticate and authorize nodes of the cluster internally.
- **Customer boarding, unboarding, and configuration:** Typically, interface nodes must be created, configured, started, stopped, or removed when customers are boarded (added to the service) or unboarded (removed from the service). This is performed by administration nodes not shown here, which register customer accounts and configure filesystems. Ideally, boarding and unboarding should consume a minimal amount of system resources and time.

As an example, a customer registers a filesystem with a given size from the filesystem storage cloud provider, and then configures machines on the customer site that mount this filesystem. The users of the customer can then use the cloud filesystem similar to how they use a local filesystem. Customers connect to the interface cluster via a dedicated physical wide-area network link or via a dedicated VPN over the Internet, ideally with low latency. The cloud provider may limit the maximal bandwidth on a customer link.

To ensure high availability and high throughput, a customer accesses the storage cloud through the clustered interface nodes. Interface nodes have to perform synchronization tasks within their cluster and with the back-end storage, generating additional traffic. An interface node has three network interfaces: one to the customer, one to other nodes in the cluster, and one to the back-end storage.

Dimensioning. The size of a filesystem storage cloud is determined by the following parameters, which are derived from service-level agreements and from the (expected or observed) load in the system: the number of customers c assigned to an interface cluster, the number of interface nodes n in a cluster (due to synchronization overhead, this incurs a trade-off between higher availability and better performance), and the number of clusters m attached to the same storage back-end.

Customer and user authentication. Describing customer authentication would exceed the scope of this work; in practice, it can be delegated to the customer's VPN endpoint

in the premises of the service-cloud provider. The authentication of users from a given customer also requires that customers provide a directory service that will serve authentication requests made by users. Such a directory service can be physically located on the customer's premises and under its administration or as separate service in the cloud. In either case, users authenticate to an interface node, which in turn relays such requests to the authentication service of the customer.

3.2 Mandatory Access Control Policies

We use mandatory access control on the filer services. In case of their compromise, MAC provides a first layer of defense on both architectures. For practical reasons, we have used SELinux. Other popular choices include *grsecurity RBAC* [7] or *TOMOYO*. These MAC systems limit the privileges of the filer services to those required, effectively creating a sandbox environment, by enforcing policies that are essentially a list of permitted operations (e.g., open certain files, bind certain ports, fork, ...).

As an example, the policies on the interface nodes basically permit the filer services to perform the following operations: bind on their listening port and accept connections, perform all filesystem operations on the customer's data directory (which resides on a distributed filesystem), append to the filer log files, and read the relevant service configuration files.

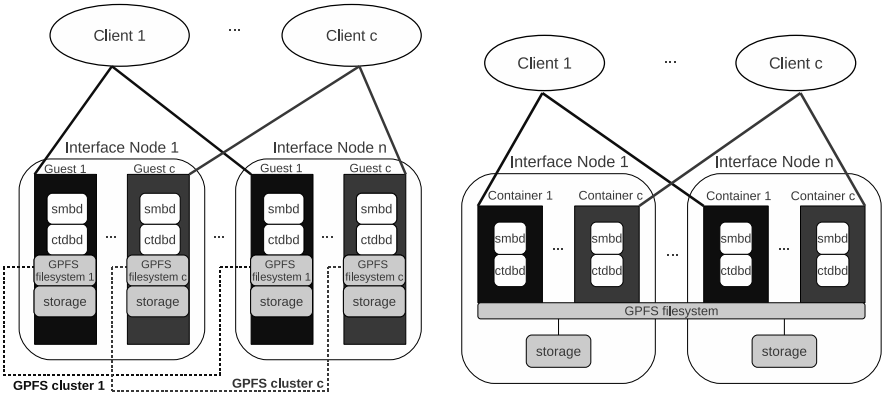
The protection provided by these policies can be defeated in two ways. One possibility is if the attacker manages to execute arbitrary code in kernel context (e.g., through a local kernel exploit), in which case it is trivial to disable any protections provided by the kernel, including MAC. The second possibility is by exploiting a hole in the SELinux policy, which would be the case, for example, if a filer service were authorized to load a kernel module.

An important example of the benefit of these policies is the restriction of accessible network interfaces to the customer and intra-cluster network interfaces only. Another example is the impossibility for processes running in the security context of the filer services to write to files they can execute, or to use `mmap()` and `mprotect()` to get around this restriction. In practice, this means, for example, that an attacker making use of a remote exploit on a filer service cannot just obtain a shell and download and execute a local kernel exploit: the attacker would have to find a way to execute the latter exploit directly within the first exploit, which, depending on the specifics of the vulnerabilities and memory protection features, can be impossible.

Note that, because of the way MAC policies are specified — that is, by white-listing the permitted operations — these examples (network interface access denied, no read and execute permission) are a consequence of the policy and do not have to be explicitly specified, which encourages policies to be built according to the least privilege principle.

3.3 VMT Architecture

We now introduce the first architecture, called the *virtualization-based multi-tenancy (VMT) architecture*. It is based on *KVM* as a hypervisor and implements multi-tenancy by running multiple virtual interface nodes as guests on the hardware of one physical interface node. Such a filesystem storage cloud has a fixed number of physical interface



(a) VMT architecture (b) OSMT architecture

Fig. 2. Two architectures for multi-tenancy, shown for one interface cluster of each architecture

nodes in every cluster, with each interface node running one guest for each customer. All guests that belong to the same customer form an interface-node cluster, which maintains the distributed filesystem with the data of the customer (labeled *GPFS cluster* in Figure 2(a), as explained below). Each virtual machine (VM) runs one instance of the required filer-service daemons, exporting the filesystems through the CIFS or NFS protocols, and has three separate network interfaces.

In terms of isolation, MAC can be applied at two levels in this architecture. The first level is inside a guest, for protecting filer services exposed to the external attackers, using the exact same policies as in the OSMT architecture[1] described below. The second level is on the host, with the idea of sandboxing guests (i.e., QEMU processes running on the host, in the case of KVM) by using multi-category security. Policies at this level do not depend on what is running inside the guests, therefore they can be applied to many virtualization scenarios. Such policies already exist and are implemented by sVirt (see `http://selinuxproject.org/page/SVirt`).

Figure 2(a) shows one of m clusters according to the VMT architecture, in which the distributed filesystem is GPFS and the daemons in each virtual machine are `smbd` (the Samba daemon, for CIFS) and `ctdbd` (the clustered trivial database daemon, used to synchronize meta-data among cluster nodes). They work together to export customer data using the CIFS protocol. The customers are also shown and connect only to their dedicated VM on each interface node. In terms of the dimension parameters from Figure 1, for every one of the m interface clusters, there are c *GPFS clusters*, each corresponding to a GPFS filesystem, and $c \cdot n$ guest virtual machines (n per customer) in this architecture.

[1] Except for the use of the multi-category security functionality (see Section 3.4): categories are not required in the VMT architecture as only one customer resides in each guest.

When a new customer is boarded, it is assigned to a cluster and a configuration script automatically starts additional guests for that customer on all the physical interface nodes within this cluster. Furthermore, a new GPFS filesystem and cluster are created for the customer on the new virtual guests. Customer data can then be copied to the filesystem and accessed by users.

3.4 OSMT Architecture

The second architecture, called the *operating-system-based multi-tenancy (OSMT) architecture*, is based on a lightweight separation of OS resources by the kernel. OS-level virtualization of this form can be achieved using *containers*, such as *LXC*, *OpenVZ* or *Zap pods* [8] for Linux, *jails* [9] for FreeBSD, and *zones* [10] for Solaris. Containers do not virtualize the underlying hardware, and thus cannot run multiple different OSes, but create new isolated namespaces for some OS resources such as filesystems, network interfaces, and processes. Processes running within a container are isolated from processes within other containers, thus they seem to be running in a separate OS. All processes share the same kernel, hence, one cannot encapsulate applications that rely on kernel threads in containers (such as the kernel NFS daemon).

In our implementation, isolation is performed using *SELinux multi-category-security (MCS)* policies [11] for shielding the processes that serve a particular customer from all others. It is then sufficient to write a single policy for each filer service that applies for all customers by simply adjusting the category of each filer service and other related components inside a container (e.g., configuration files, customer data). This ensures that no two distinct customers can access each other's resources (because they belong to different categories). In comparison to the VMT architecture, the policies in the guest that contain the filer services, and the policies in the host that isolate the customers are now combined into a single policy which achieves the same goals.

In addition, a *change-root (chroot)* environment is installed, whose only purpose is to simplify the configuration of the isolated services and the file labeling for SELinux. We refer to such a customer isolation domain as a *container* in the remainder of this work. A container dedicated to one customer on an interface node consists of a *chroot directory* in the root filesystem of the interface node, which contains all files required to access the filesystem for that customer. All required daemons accessed by the customer run within the container. Because of the chroot environment, the default path names, all configuration files, the logfile locations, and so on, are all the same or found at the same locations for every customer; this is implemented through read-only mount binds, without having to copy files or create hard links. This approach makes our container-based setup amenable to automatic maintenance through existing software distribution and packaging tools.

This form of isolation does not provide new namespaces for some critical kernel resources (process identifiers are global, for instance); it does not allow a limitation of memory and CPU usage either. However, it causes a smaller overhead for isolation than hypervisor-based virtualization does.

Figure 2(b) shows an interface cluster following the OSMT architecture, in which the distributed filesystem is GPFS, shared by all containers within a cluster. Each container runs a single instance of each of the `smbd` and `ctdbd` daemons, accessed only by the

corresponding customer. In the terms of the dimension parameters from Figure 1, for every one of the m interface clusters, there exists *one* GPFS filesystem (only one per cluster), n kernels (each kernel is shared by c customers), and $c \cdot n$ containers (n per customer).

Customer boarding is done by a script that creates an additional container on every interface node in the cluster, and a data directory for that customer in the shared distributed filesystem of the interface cluster. The daemons running inside the new containers must be configured to export the customer's data directory using the protocols selected. No changes have to be made to the configuration of the distributed filesystem on the interface nodes.

4 Security Comparison

In this section, we discuss the differences between the VMT architecture and the OSMT architecture from a security viewpoint. Because we aim to compare the two approaches, we only briefly touch on those security aspects that are equal for the two architectures. This concerns, for instance, user authentication, attacks from one user of a customer against other users of the same customer, and attacks by the service provider ("insider attacks" from its administrators). These aspects generally depend on the network filesystem and the user-authentication method chosen, as well as their implementations. They critically affect the security of the overall solution, but are not considered further here.

4.1 Security Model

We consider only attacks by a malicious customer, i.e., attacks mounted from a user assigned to one customer against the service provider or against other customers. In accordance with the traditional goals of information security, we can distinguish three types of attacks: those compromising the confidentiality, the integrity, or the availability of the service and/or of data from other customers.

Below we group attacks in two categories. First we discuss *denial-of-service (DoS) attacks* targeting service availability in Section 4.3. Second, we subsume threats against the confidentiality and integrity of data under *unauthorized data access* and discuss them in Section 4.4.

We assume that the cloud service provider is trusted by the customers. We also disregard customer-side cryptographic protection methods, such as filesystem encryption [12] and data-integrity protection [13]. These techniques would not only secure the customer's data against attacks from the provider but also protect its data from other customers. Such solutions can be implemented by the customer transparently to the service provider and generally come with their own cost (such as key management or the need for local trusted storage).

4.2 Comparison Method

An adversary may compromise a component of the system or the whole system with a certain *likelihood*, which depends on the vulnerability of the component and on properties of the adversary such as its determination, its skills, the resources it invests in an

attack and so on. This likelihood is influenced by many factors, and we refrain from assigning numerical values or probabilities to it, as it cannot be evaluated with any reasonable accuracy [14, Chap. 3–4].

Instead we group all attacks into three sets according to the likelihood that an attack is feasible with methods known today or the likelihood of discovering an exploitable vulnerability that immediately enables the attack. We roughly estimate the relative severity of attacks and vulnerabilities according to criteria widely accepted by computer emergency readiness teams (CERTs), such as previous exploits or their attack surfaces. Our three likelihood classes are described by the terms *unlikely*, *somewhat likely* and *likely*.

In Section 4.4 we model data compromise in the filesystem storage cloud through graphical *attack trees* [15]. They describe how an attacker can reach its goal over various paths; the graphs allow a visual comparison of the security of the architectures.

More precisely, an attack tree is a directed graph, whose nodes correspond to states of the system. The initial state is shown in white (meaning that the attacker obtains an account on the storage cloud) and the exit node is colored black (meaning that the attacker gained unauthorized access to another customer's data). A state designates a component of the system (as described in the architecture) together with an indication of the security violation the attacker could have achieved or of how the attacker could have reached this state.

An edge corresponds to an attack that could be exploited by an attacker to advance the state of compromise of the system. The intermediate nodes are shown in various shades of gray, roughly corresponding to the severity of the compromise. Every attack is labeled by a likelihood (unlikely, somewhat likely, or likely), represented by the type of arrow used.

4.3 Denial-of-Service Attacks

Server crashes. An attacker can exploit software bugs causing various components of an interface node to crash, such as the filer services (e.g., the NFS or CIFS daemon) or the OS kernel serving the customer. Such crashes are relatively easy to detect and the service provider can recover from them easily by restarting the component. Usually such an attack can also be attributed to a specific customer because the service provider maintains billing information of the customer; hence the offending customer can easily be banned from the system.

Both architectures involve running dedicated copies of the filer services for each customer. Therefore, crashing a filer service only affects the customer itself. Although the attack may appear likely in our terminology, we consider it not to be a threat because of the external recovery methods available.

Note that non-malicious faults or random crashes of components are not a concern because all components are replicated inside an interface cluster, which means that the service as a whole remains available. Crashes due to malicious attacks, on the other hand, will affect all nodes in a cluster as the attacker can repeat its attack.

Furthermore, any server crash has to be carried out remotely and therefore mainly affects the network stack. It appears much easier for a local user to crash a server, in contrast. For this, the attacker must previously obtain the privilege to execute code on the interface node, most likely through an exploit in one of the filer services. However,

when attackers have obtained a local account on an interface node, they can cause much more severe problems than simply causing a crash (Section 4.4). Therefore we consider a locally mounted DoS attack as an acceptable threat.

In the *VMT architecture*, a kernel crash that occurs only inside the virtual machine dedicated to the customer does not affect other customers, which run in other guests — at least according to the generally accepted view of virtual-machine security. However, the effects on other guests depend on the kind of the DoS attack. A network attack that exploits a vulnerability in the upper part of the network stack (e.g., UDP) most likely only crashes the targeted guest. But an attack on lower-layer components of the hypervisor (e.g., network interface driver), which run in the host, can crash the host and all guests at once. Moreover, additional vulnerabilities may be introduced through the hypervisor itself.

In the *OSMT architecture*, an attacker may crash the OS kernel (through a vulnerability in the network interface driver or a bug in the network stack), which results in the crash of the entire interface node and disables also the service to all other customers. Thus, the class of DoS attacks targeted against the OS kernel has a greater effect than in the VMT architecture.

Resource exhaustion. An attacker can try to submit many filesystem requests to exhaust some resource, such as the network between the customers and the interface nodes, the network between interface nodes and the resource cluster, or the available CPU and memory resources on the interface nodes. Network-resource exhaustion attacks affect both our designs in the same way (and more generally, are a common problem in most Internet services); therefore, we do not consider them further and discuss only the exhaustion of host resources.

In the *VMT architecture*, hypervisors can impose a memory bound on a guest OS and limit the number of CPUs that a guest can use. For example, a six-CPU interface node may be shared by six customers in our setup. Limiting every guest to two CPUs means that the interface node still tolerates two malicious customers that utilize all computation power of their dedicated guests, but continues to serve the other four customers with two CPUs.

The impact of a resource-exhaustion attack with a container setup in the *OSMT architecture* depends on the container technology used and its configuration.

In our study, we use a container technology (SELinux and chroot environment) that cannot restrict the CPU used or the memory consumed by a particular customer. Given proper dimensioning of the available CPU and memory resources with respect to the expected maximal load per customer, however, a fair resource scheduler alone can be sufficient to render such attacks harmless.

With more advanced container technology, such as LXC (based on the recent cgroup process grouping feature of the Linux kernel), it is possible to impose fine-grained restrictions on these resources, analogously to a hypervisor. For instance, the number of CPUs attributed to a customer and the maximally used CPU percentage can be limited for every customer.

4.4 Unauthorized Data Access

We describe here the attack graphs in Figures 3 and 4 as explained in Section 4.2. Some attacks are common and apply to both architectures; they are described first. We then present specific attacks against the VMT and OSMT architectures. Each attack graph includes all attacks relevant for the architecture.

Common attacks. *Filer service compromise.* Various memory corruption vulnerabilities (such as buffer overflows, string format vulnerabilities, double frees) are notorious for allowing attackers to execute arbitrary code with the privileges of the filer service. However, protection measures such as address space layout randomization, non-executable pages, position-independent executables, and stack canaries, can render many attacks impossible without additional vulnerabilities (e.g., information leaks). This is especially true for remote attacks, in which the attacker has very little information (e.g., no access to `/proc/pid/`) and less control over memory contents (e.g., no possibility of attacker-supplied environment variables) than for local attacks. For these reasons, we categorize these attacks as "somewhat likely."

Complementing the aforementioned attacks that permit arbitrary code execution, confused deputy attacks [16] form a weaker class of attacks. In such an attack, the attacker lures the target application into performing an operation unauthorized to the attacker without obtaining arbitrary code execution. Directory traversal, whereby an attacker tricks the filer service into serving files from a directory that should not be accessible, is a famous example of such attacks in the context of storage services (e.g., CVE-2010-0926, CVE-2001-1162 for CIFS). Clearly, such attacks leverage the privileges of the target process: a process that has restricted privileges is not vulnerable. Therefore, they form a weaker class of attacks: preventing unauthorized data access to an attacker who has compromised the filer service through arbitrary code execution also prevents these attacks. Furthermore, confused deputy attacks are very unlikely to serve as a stepping stone for a second attack (e.g., accessing the internal network interface), which would be required to access another tenant's data in both architectures here. Consequently, we do not consider confused deputy attacks any further.

Kernel compromise. We distinguish between remote and local kernel attacks. The reasoning in the previous paragraph concerning the lack of information and memory control is essentially also valid for remote kernel exploits. However, for the kernel, the attack surface is much more restricted: typically network drivers and protocols, and usually under restrictive conditions (e.g., LAN access). Recently, Wi-Fi drivers have been found to be vulnerable (CVE-2008-4395), as well as the SCTP protocol (CVE-2009-0065) both of which would not be used in the context of a filesystem storage cloud. For these reasons, we categorize these attacks as "unlikely." In contrast to remote exploits, we categorize local kernel exploits as "somewhat likely" given the information advantage (e.g., `/proc/kallsyms`) and capabilities of a local attacker (e.g., mapping a fixed memory location). Many recently discovered local kernel vulnerabilities confirm this view.

SELinux bypass. The protection provided by SELinux can be bypassed in two ways. One of them is by leveraging a mistake in the security policy written for the

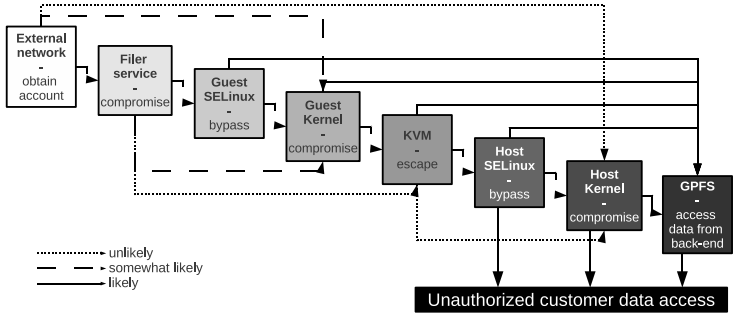


Fig. 3. Attack graph for the VMT architecture

application: if the policy is too permissive, the attacker can find ways to get around some restrictions. An example of such a policy vulnerability was found in sVirt [17]: an excessively permissive rule in the policy allowed an attacker in the hypervisor context to write directly to the physical drive, which the attacker can leverage in many ways to elevate his privileges. The second option for bypassing SELinux is by leveraging a SELinux implementation bug in the kernel. An example of such a vulnerability is the bypass of NULL pointer dereference protections. The Linux kernel performs checks when performing `mmap()` to prevent a user from mapping addresses lower than `mmap_min_addr` (which is required for exploiting kernel NULL pointer dereferences vulnerabilities). SELinux also implemented such a protection (with the additional possibility of allowing such an operation for some trusted applications). However, the SELinux access control decision in the kernel would basically override the `mmap_min_addr` check, weakening the security of the default kernel (CVE-2009-2695). For these reasons, we categorize these attacks as "somewhat likely."

Attacks against the VMT architecture. *VM escapes.* Although virtual machines are often marketed as the ultimate security isolation tool, it has been shown [18,19] that many existing hypervisors contain vulnerabilities that can be exploited to escape from a guest machine to the host. We assume these attacks are "somewhat likely".

Filer service compromise: NFS daemon and SELinux. Apart from the helper daemons, which represent a small part of the overall code (e.g., `rpc.mountd`, `rpc.statd`, `portmapd`), most of the `nfsd` code is in kernel-space. This means it is not possible to restrict the privileges of this code with a MAC mechanism in the sense that a vulnerability in this code might directly lead to arbitrary code execution in kernel mode. The authors of [20] tried to implement such a protection within the kernel but this approach cannot guarantee sufficient isolation of kernel code simply because an attacker with `ring 0` privileges can disable SELinux. We categorize this attack as "somewhat likely."

Attacks against the OSMT architecture. *Container escapes.* As mentioned in 3.4, we have implemented what we refer to as containers using a chroot environment. As

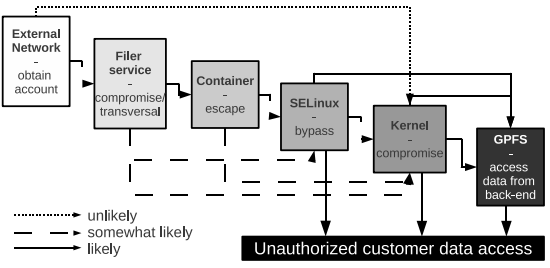


Fig. 4. Attack graph for the OSMT architecture

is widely known, a chroot environment does not effectively prevent an attacker from escaping from the environment and provides limited isolation. For completeness, we include a container-protection layer which corresponds to the chroot environment (without SELinux) in Figure 4, and marked it as "likely" to be defeated. However, containers such as LXC do implement better containment using the *cgroups* feature of Linux. While these technologies have a clean and simple design, it is still likely that some vulnerabilities allowing escapes can be found, especially because they are very recent (one such current concern regards containers mounting a `/proc` pseudo-filesystem).

4.5 Conclusion

A high-level comparison of Figures 3 and 4 shows that the VMT architecture has many more layers and could lead to the conclusion that the VMT approach provides better security. However, we also have to take into account the various attacks: most notably, it is possible that an attacker uses the internal network interface effectively for customer data access, and that this network interface is accessible from within the guest VMs (which is required, because the distributed filesystem service runs in the guest). The possibility of this attack renders other layers of protection due to VM isolation much less useful in the VMT architecture.

In other words, a likely chain of compromises that can occur for each scenario is

- for VMT:
  1. attacker compromises filer service, obtaining local unprivileged access,[2]
  2. attacker exploits a local kernel privilege escalation vulnerability that can be exploited within the MAC security context of the filer service,
  3. attacker accesses files of a customer through the distributed filesystem (assuming no authentication or authorization of nodes and no access control on blocks).
- for OSMT:
  1. attacker compromises filer service, obtaining local unprivileged access,
  2. attacker exploits a local kernel privilege escalation vulnerability that can be exploited within the MAC security context of the filer service,

[2] In the case of a kernel NFS daemon, it is possible that the attacker directly obtains `ring 0` privileges and can therefore skip the next step, however this is less likely.

3. attacker accesses files of a local co-tenant, or through the distributed filesystem for other customers.

Although it is expected that hypervisor-level multi-tenancy can, in general, be a better security design than OS-level multi-tenancy, we have seen in this section that in the case of a filesystem storage cloud and under our assumptions (i.e., that in a distributed filesystem, each individual node is trusted), no solution was clearly more secure than the other. Both solutions could be used to achieve an acceptable level of security.

5 Performance and Scalability Evaluation

In this section, we present a performance evaluation comparing the VMT and OSMT architectures for given customer workloads. Our experimental filesystem storage cloud setup is based on the IBM SONAS [1] product, on which we implemented the two multi-tenant architectures in the interface nodes, using the same physical infrastructure. We ran the benchmarks on both setups using the same customer workload based on the CIFS protocol, and measured various system metrics. Besides measuring the performance of the two architectures, the evaluation allows us to compare the scalability of the architectures.

In this section, we use the term *client* to refer to a single user belonging to a customer (each customer has one user, and we refer to it as client).

5.1 Experimental Setup

We experiment with two storage back-end configurations in our benchmark.

1. RAM-disks directly on the interface node. This allows us to observe the performance of the systems in the absence of bottlenecks due to physical disk-access limitations, which is useful for analyzing the scalability of the interface node itself.
2. Actual disk-based storage, in the form of a direct-attached DS3200 storage subsystem, which allows us to evaluate performance in a realistic setup.

The experimental setup consists of two interface nodes forming a two-node GPFS cluster, and one client node. All network connections are 1 GbE links. To measure the performance of a single interface node, we connect the client node to only one of the interface nodes. Thus, all client traffic goes to a single interface node over a single 1 GbE link. Although the second interface node receives no direct client traffic, it is included in the benchmark setup to have a realistic 2-node GPFS cluster setup.

Figure 5 shows the setup and the server configurations. All servers run RHEL Server 5.5 with kernel version 2.6.18-194. The IBM SONAS version used on the interface nodes is 1.5.3-20. On the DS3200 storage subsystem, we use a single 5+1 RAID5 array with a total capacity of 2 TB using 15k RPM SAS drives. The storage subsystem is attached to only one of the interface nodes — the same node that is connected to the client.

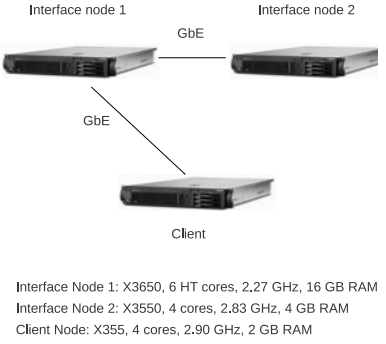


Fig. 5. Experimental setup

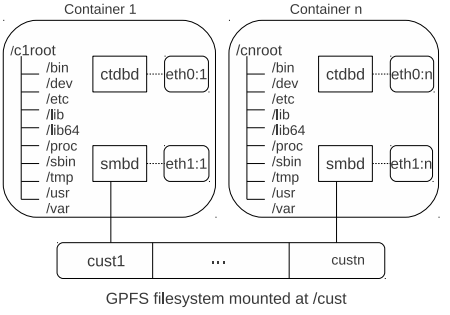


Fig. 6. Setup of containers in an interface node

In the VMT setup, we use *KVM* and *libvirt* [21] to create and manage virtual machines on the interface nodes. Each interface node has one virtual machine for each customer, and all the virtual machines belonging to a specific customer across all the interface nodes are clustered together, with a single filesystem containing that customer's data. Each virtual machine has two virtual Ethernet interfaces assigned, which are bridged to the corresponding interfaces on the host; one interface is used for GPFS-cluster synchronization traffic, and the other for client traffic. In the RAM-disk configuration, the total RAM-disk allocation on each interface node is evenly divided between all the virtual machines running on that node. Each virtual machine uses a single RAM-disk for the creation of the distributed filesystem. We make sure that the total physical memory allocation in the VMT setup is exactly the same as in the OSMT setup. In the disk-based configuration, the 2 TB SAS array is evenly divided into partitions, which are then exposed to the virtual machines as raw devices through virtio interfaces. Each virtual machine uses the raw device for the creation of the distributed filesystem.

The OSMT setup on the two interface nodes consists of dedicated "chroot directories" for each container in the root filesystems of the interface nodes. Each container is assigned two aliased Ethernet interfaces: one for GPFS-cluster synchronization traffic and the other for client traffic. All the network interfaces assigned to the containers are created as aliases of the host interfaces. We use aliases to simulate an environment in which each customer has a dedicated secure access channel to the interface cluster. In an actual customer environment, this secure channel would take the form of a VPN. Figure 6 shows the OSMT setup for a single interface node. The interface node shown here is part of a GPFS cluster. The GPFS filesystem extends to the other interface nodes that are part of the same cluster. For the RAM-disk configuration, a single RAM disk is used on each interface node and a single GPFS filesystem is created using the RAM disks from both interface nodes. This filesystem is used by all the containers, with specific data directories allotted to each customer. For the disk-based configuration, the entire disk array is used to create a single filesystem, which is then used in the same way as in the RAM-disk setup.

5.2 Tools Used in the Benchmarks

The standard fileserver benchmark used widely to evaluate the performance of Windows fileservers is *Netbench*. However, Netbench runs only on the Windows platform and requires substantial hardware for a complete benchmark setup. Under Linux, the Samba software suite provides two tools that can be used to benchmark SMB servers using Netbench-style I/O traffic, namely *dbench* and *smbtorture BENCH-NBENCH*. Both tools read a loadfile provided as input to perform the corresponding I/O operations against the fileserver being benchmarked. The dbench tool can be run against both NFS and SMB fileservers, the smbtorture tool is specific to SMB fileservers. We used the smbtorture BENCH-NBENCH tool because it offers more control over various aspects of the benchmark runs than the dbench tool, and also because our benchmark focuses solely on client access using the CIFS protocol.

The loadfile used in our benchmark consisted mainly of file creation operations, 4 KiB reads, 64 KiB writes, and some metadata queries.

To collect system metrics from the interface and client nodes during the execution of the benchmark, we used the *System Activity Report (SAR)* tool from the Red Hat *sysstat* package. The SAR tool collects, reports and saves system activity information, such as CPU and memory usage, load, network and disk usage, by reading the kernel counters at specified intervals.

5.3 Benchmark Procedure

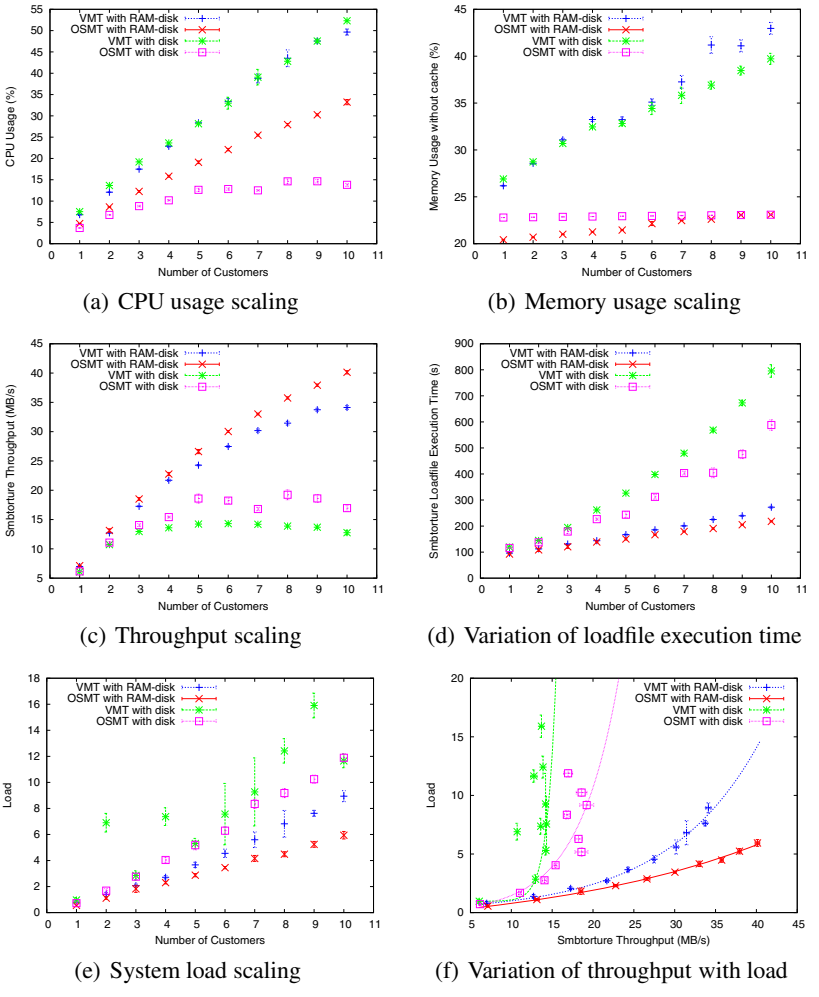
We collected and analyzed the following system metrics on the interface nodes (in the case of the VMT architecture, metrics were collected on the host):

1. *CPU usage*: We used the *%idle* values reported by SAR to compute the *%used* values in each interval.
2. *System load*: We used the one-minute system load averages reported by SAR. Note that on Linux systems, the load values reported also take into account processes that are blocked waiting on disk I/O.
3. *Memory usage*: We recorded the memory usage both with and without buffers. In both cases, we excluded the cached memory. In the VMT setups, we also excluded the cached memory on the virtual machines running on that host.

In addition to these system metrics, we also recorded the throughput and loadfile execution time reported by smbtorture on the clients. We performed 10 iterations for each benchmark run — caches were preconditioned by 2 dry runs. We then computed 95% confidence intervals using the t-distribution for each metric measured.

5.4 Results

The graphs in this section show the variation of a particular system metric on both the VMT and the OSMT architecture. For each architecture, we show the variation of the metric on the RAM-disk-based setup as well as the disk-based setup. Note that each customer is simulated with a single user (one client).



(a) CPU usage scaling

(b) Memory usage scaling

(c) Throughput scaling

(d) Variation of loadfile execution time

(e) System load scaling

(f) Variation of throughput with load

Fig. 7. Benchmark results

Figure 7(a) shows the variation of CPU usage as a function of the number of customers. The overall CPU usage is much lower in the OSMT architecture, for both the RAM-disk and the disk setup. In the OSMT architecture, the CPU usage is significantly lower when we use disk storage and flattens above 5 customers because more cycles are spent waiting on disk I/O than in the RAM-disk-based setup. Hence, the performance of the disks dominates the overall throughput. For the VMT architecture, however, the CPU usage is about the same in both setups and therefore starts impacting the overall throughput when the number of customers is higher. For 10 customers, this culminates in a difference of about 77% in CPU usage between the two architectures when disk storage is used.

The variation of memory usage with the number of customers is shown in Figure 7(b). In the OSMT architecture, the memory usage remains relatively constant irrespective of

the number of customers. In the VMT architecture, however, the memory usage grows almost linearly with the number of customers. We explain the less-than-2% discrepancy in the increase between the disk- and RAM-disk-based OSMT setup by the varying buffer size requirements with respect to the latency of the medium: in the disk setup, access time is slower and the buffers reach their maximum size already for a single customer. For 10 customers, there is a difference of about 45% in the memory usage of the two architectures when disk storage is used.

Figures 7(c) and 7(d) show the variation of throughput and loadfile execution time. As expected, the loadfile execution times are the lowest on the RAM-disk setups. The throughput reported by smbtorture is also higher on the RAM-disk setups.

Independently of the type of disks used, the VMT architecture clearly gets an additional penalty from the higher CPU load of the system which results in lower throughput and higher loadfile execution time than the OSMT architecture.

Figure 7(e) shows the variation of system load with the number of customers. In both architectures, the system load is higher when disk storage is used because of all the cycles spent waiting on disk I/O. The system load has a much higher variance in the VMT architecture. We speculate that this is because of variations in the amounts of disk activity required to maintain the state of the virtual images on disk during the different benchmark runs, which resulted in a large variation in the measured load values.

Figure 7(f) shows system load as a function of the throughput. Generally, the lower the load and the higher the throughput, the better the scalability. Clearly, throughput scales better with system load in the RAM-disk setup than in the disk-based setup. Throughput scales best in the OSMT architecture using RAM disks. In the VMT architecture, the wide variation in system load in the disk-based setup results in a relatively steep curve, whereas the curve is flatter for the RAM-disk setup. Overall it can be seen that the OSMT architecture scales better than the VMT architecture.

6 Related Work

Although the designs of some free or open-source object storage cloud solutions [22,23] are available, to the best of our knowledge no commercial cloud-storage provider has publicly documented its internal architecture. In this work, we analyze for the first time, how the technology applied for obtaining multi-tenancy impacts the security of the customer data.

In this section we discuss alternative techniques that can be leveraged to obtain similar security goals as the architectures analyzed in this work. Because we target this study at production environments, our two architectures are restricted to tools that have achieved a certain maturity and stability. Some of the isolation techniques mentioned in this section are too recent and do not yet satisfy these conditions.

Micro-kernels [24] and virtual-machine monitors are comparable to some extent [25]. In terms of security isolation, Hohmuth et al. [26] argue that the trusted computing base (TCB) is usually smaller with micro-kernels than with hypervisors. In particular, the authors suggest that extending virtualized systems with micro-kernel-like features, such as inter-process communication, can reduce the overall TCB. Although we do not use the TCB terminology to capture better the advantages of a layered security design, we

believe their main argument also applies to some extent in the context of this work. For instance, in the VMT architecture, isolating the distributed filesystem[3] from the guests running the filer services and establishing a stable and secure way of accessing the filesystem (e.g., paravirtualizing the distributed filesystem) would significantly improve the security of this architecture. To the best of our knowledge, the most mature existing technology for KVM that is close to realizing this goal is VirtFS [27].

Better isolation of the filer services and the distributed filesystem can also be achieved by improving the security of the Linux kernel, especially in the OSMT architecture. Using virtual machine introspection (VMI), Christodorescu et al. [28] present an architecture for kernel code integrity and memory protection of critical read-only sections (e.g. jump tables) to prevent most kernel-based rootkits with minimal overhead and without source code analysis. With source code analysis, Petroni and Hicks [29] prevent rootkits by ensuring control-flow integrity of the guest kernel at the hypervisor and therefore also prevent all control-flow redirection based attacks for the Linux kernel, which represents a significant security improvement.

Another approach to enhance the security of the kernel is grsecurity PaX [7], which provides a series of patches that mitigate the exploitation of some Linux kernel vulnerabilities with low performance overhead. In particular, it provides base address randomization for the kernel stack, prevents most user-space pointer dereferences by using segmentation, and prevents various information leaks which can be critical for successful exploitation of vulnerabilities. Other grsecurity patches also feature protection for the exploitation of user-space vulnerabilities.

7 Conclusion

We have presented in this work two alternatives for implementing a multi-tenant filesystem storage cloud, with one architecture isolating different tenants through containers in the OS and the other isolating the tenants through virtual machines in the hypervisor. Neither architecture offers strictly "better" security than the other one; rather, we view both as viable options for implementing multi-tenancy. We have observed that the overhead of the VMT architecture, due to the additional isolation layers, is significantly higher than that of the OSMT architecture as soon as multiple tenants (and not even a large number) access the same infrastructure. We conclude that, under cost constraints for a filesystem storage cloud, the OSMT architecture is a more attractive choice.

References

1. IBM Scale Out Network Attached Storage, http://www-03.ibm.com/systems/storage/network/sonas/
2. Schmuck, F., Haskin, R.: GPFS: A Shared-disk File System For Large Computing Clusters. In: Proc. File and Storage Technologies (2002)
3. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: kvm: the Linux Virtual Machine Monitor. In: Proc. Linux Symposium, vol. 1 (2007)

[3] In the sense of Hohmuth et al. [26], the distributed filesystem is in the TCB.

4. Cai, H., Reinwald, B., Wang, N., Guo, C.: SaaS Multi-Tenancy: Framework, Technology, and Case Study. International Journal of Cloud Applications and Computing (IJCAC) 1(1) (2011)
5. Traeger, A., Rai, A., Wright, C., Zadok, E.: NFS File Handle Security. In: Tech. Rep., Computer Science Department, Stony Brook University (2004)
6. Oehme, S., Deicke, J., Akelbein, J., Sahlberg, R., Tridgell, A., Haskin, R.: IBM Scale out File Services: Reinventing network-attached storage. IBM Journal of Research and Development 52(4.5) (2008)
7. grsecurity, http://grsecurity.net/
8. Osman, S., Subhraveti, D., Su, G., Nieh, J.: The Design and Implementation of Zap: A System for Migrating Computing Environments. In: ACM SIGOPS Operating Systems Review, vol. 36(SI) (2002)
9. Kamp, P., Watson, R.: Jails: Confining the omnipotent root. In: Proc. International System Administration and Network Engineering (2000)
10. Price, D., Tucker, A.: Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In: Proc. System Administration (2004)
11. McCarty, B.: SELinux: NSA's Open Source Security Enhanced Linux (2004)
12. Diesburg, S.M., Wang, A.-I.A.: A survey of confidential data storage and deletion methods. ACM Computing Surveys 43 (December 2010)
13. Sivathanu, G., Wright, C.P., Zadok, E.: Ensuring data integrity in storage: Techniques and applications. In: Proc. Storage Security and Survivability (2005)
14. Schechter, S.: Computer Security Strength & Risk: A Quantitative Approach. PhD thesis, Harvard University Cambridge, Massachusetts (2004)
15. Schneier, B.: Attack trees. Dr. Dobb's journal 24(12) (1999)
16. Hardy, N.: The Confused Deputy. ACM SIGOPS Operating Systems Review 22(4) (1988)
17. Wojtczuk, R.: Adventures with a certain Xen vulnerability (in the PVFB backend). Message Sent to Bugtraq Mailing List on October 15 (2008)
18. Ormandy, T.: An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. In: Proc. CanSecWest Applied Security Conference (2007)
19. Kortchinsky, K.: Cloudburst – Hacking 3D and Breaking out of VMware (2009)
20. Blanc, M., Guerin, K., Lalande, J., Le Port, V.: Mandatory Access Control implantation against potential NFS vulnerabilities. In: International Symposium on Collaborative Technologies and Systems (2009)
21. libvirt: The virtualization API, http://libvirt.org/index.html
22. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The Eucalyptus Open-source Cloud-computing System. In: Proc. Cluster Computing and the Grid (2009)
23. OpenStack Swift, http://swift.openstack.org/
24. Liedtke, J.: On micro-kernel construction. In: Proc. SOSP (1995)
25. Heiser, G., Uhlig, V., LeVasseur, J.: Are Virtual Machine Monitors Microkernels Done Right? ACM SIGOPS Operating Systems Review 40(1) (2006)
26. Hohmuth, M., Peter, M., Härtig, H., Shapiro, J.S.: Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In: Proc. SIGOPS European Workshop (2004)
27. Jujjuri, V., Hensbergen, E.V., Liguori, A.: VirtFS – A virtualization aware File System pass-through. In: Proc. Ottawa Linux Symposium (2010)
28. Christodorescu, M., Sailer, R., Schales, D.L., Sgandurra, D., Zamboni, D.: Cloud Security Is Not (Just) Virtualization Security: A Short Paper. In: Proc. CCSW (2009)
29. Petroni Jr, N.L., Hicks, M.: Automated Detection of Persistent Kernel Control-Flow Attacks. In: Proc. CCS (2007)

SafeWeb: A Middleware for Securing Ruby-Based Web Applications

Petr Hosek[1], Matteo Migliavacca[1], Ioannis Papagiannis[1], David M. Eyers[2], David Evans[3], Brian Shand[4], Jean Bacon[3], and Peter Pietzuch[1]

[1] Imperial College London
{ph1310,migliava, ip108,prp}@doc.ic.ac.uk
[2] University of Otago
dme@cs.otago.ac.nz
[3] University of Cambridge
{firstname.lastname}@cl.cam.ac.uk
[4] ECRIC, National Health Service
brian.shand@cbcu.nhs.uk

Abstract. Web applications in many domains such as healthcare and finance must process sensitive data, while complying with legal policies regarding the release of different classes of data to different parties. Currently, software bugs may lead to irreversible disclosure of confidential data in multi-tier web applications. An open challenge is how developers can guarantee these web applications only ever release sensitive data to authorised users without costly, recurring security audits.

Our solution is to provide a trusted middleware that acts as a "safety net" to event-based enterprise web applications by preventing harmful data disclosure before it happens. We describe the design and implementation of SafeWeb, a Ruby-based middleware that associates data with security labels and transparently tracks their propagation at different granularities across a multi-tier web architecture with storage and complex event processing. For efficiency, maintainability and ease-of-use, SafeWeb exploits the dynamic features of the Ruby programming language to achieve label propagation and data flow enforcement. We evaluate SafeWeb by reporting our experience of implementing a web-based cancer treatment application and deploying it as part of the UK National Health Service (NHS).

1 Introduction

Enterprise web applications in areas such as healthcare, financial processing and government services must selectively expose sensitive data to authorised sets of web users. For example, a cancer researcher may want to query a centralised patient database over the web for anonymised health records of patients that have a given type of cancer. The costs of inadvertently disclosing confidential data to the wrong users due to implementation errors in web applications are high—hospitals and medical practitioners in the UK are legally liable for unauthorised disclosure of patient data without prior consent. Due to new legislation introduced in 2010, organisations can be fined up to £500,000 for security

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 491–511, 2011.

breaches [25]. In this paper, we address *how to implement secure enterprise web applications that are guaranteed to comply with data protection policy.*

Enforcing a data protection policy *end-to-end*, i.e. , across an entire multi-tier web application, is challenging. An implementation error in any tier of a web application may result in unauthorised data disclosure. Developers may introduce software bugs inadvertently or based on misunderstandings of requirements. Achieving correctness is even more challenging for web applications that process different types of data from multiple domains, such as hospitals, laboratories and insurance providers, each with their own security requirements.

Current best practices include manual source code auditing for new applications, which is error-prone and costly. Tools for static analysis such as Pixy [10] validate that the implementation satisfies given invariants. Their use, however, requires expert knowledge to formalise invariants and they cannot handle large distributed web applications due to these applications' size and complexity.

We make two assumptions: (1) In terms of the threat model, we assume that the external environment is hostile, but that application code is not explicitly malicious, even if threats might be caused by bugs in the implementation. (2) We assume that stakeholders are willing to accept some performance overhead—in terms of request throughput and latency—for increased security.

Our solution is to propose a middleware that implements a "safety net" by providing a data-centric security approach that integrates well with multi-tier web applications. Our middleware is based on two key ideas: It decouples the processing of confidential data from the handling of web requests. In addition, it tracks data as it flows through the web application in order to ensure its confidentiality and integrity. This means that implementation bugs in the web request handling logic cannot cause any unauthorised confidential data to be disclosed. By tracking data propagation by means of security labels, the middleware performs automatic and appropriate compliance checks at the boundaries between application components without relying on developer support. This reduces the effort required for security audits.

We demonstrate the practicality of this approach by describing SafeWeb, a Ruby-based middleware that enforces data flow policy across web applications. SafeWeb consists of an *event-processing backend* that processes data asynchronously from a confidential data store according to application-specific business logic. Events are associated with security labels, which are tracked by SafeWeb as they propagate between event processing components. This mechanism is implemented efficiently through isolation of processing components using Ruby's safe levels. A separate *web frontend* serves processed event data in response to web requests while maintaining security labels at the variable level using Ruby's meta-programming features. Before sending data to web users, SafeWeb validates the associated security labels against user privileges, thus preventing violations of the data protection policy.

SafeWeb is designed to integrate well with existing security practices at an organisational, architectural and infrastructure level. From an organisational viewpoint we complement existing security practices, such as network

partitioning through firewalls and security code auditing, without requiring significant changes. At the infrastructure level, SafeWeb can easily be applied to production environments. In contrast with classical label-based approaches, we avoid complex changes in the language runtime. We are able to provide tracking purely at the middleware level through careful exploitation of Ruby's meta-programming and security features.

We evaluate SafeWeb in a real-world healthcare environment by developing *MDT web portal*, a web application that provides information about ongoing cancer treatment of patients to teams of medical practitioners at hospitals. We discuss the deployment of this application using SafeWeb as part of ECRIC—an organisation within the UK National Health Service (NHS) that collects relevant patient-sensitive oncological data. We show that SafeWeb can guarantee that medical records are only exposed to authorised users—even with implementation bugs in the processing logic of the MDT web portal application. It integrates well with existing information systems and introduces only minimal overheads in terms of application development effort and performance.

In summary, the main contributions of the paper are: (1) a middleware for securing web applications that uses event processing to decouple queries from sensitive data; (2) an application of information flow control techniques across all tiers of a web application to prevent non-compliant disclosure of sensitive data to users and an efficient implementation of data tracking using Ruby language mechanisms; (3) an evaluation of this approach in a healthcare environment using a realistic web application for supporting cancer treatment at hospitals.

In §2 we provide background on the security requirements of web applications dealing with confidential data. Then, we present a general data-centric security mechanism addressing these requirements, explaining why existing technologies cannot be applied in production environments (§3). The SafeWeb middleware is described in §4, focusing on the different components of its architecture. We evaluate our approach in §5 through an implementation of a web application using SafeWeb, describing its security properties and providing a performance analysis. In §6, we discuss related work, and we draw conclusions in §7.

2 Data Confidentiality in Enterprise Web Applications

Organisations in the public and private sector collect, process and analyse personal data to improve the quality of the services that they offer. Due to the sensitivity of personal data, maintaining its confidentiality is crucial. As a consequence, it is necessary to verify that applications are not vulnerable to compromise from external attacks and that confidential data is handled in compliance with the policies set by organisations.

Current best practices to maintain data confidentiality in applications are costly, error-prone and time consuming: organisations adopt a series of security measures including risk assessments, internal security code reviews and external security consultations. These measures are intertwined with project development to the point that development of new services is limited. For example,

healthcare organisations struggle to develop new medical applications that have the potential to improve patient care quickly and cost-effectively.

Middleware can be used to reduce the cost of development and deployment of new applications by moving security auditing effort from applications to reusable middleware components. In this paper, *we describe a middleware that can increase the trust placed in applications that process and provide access to confidential data by placing applications within a "safety net" that, within the constraints of a production environment, protects data from compromise and accidental disclosure*. The goal is to satisfy the following two security requirements—both of which are discussed in the context of a healthcare example in the next section:

S1 Access to confidential data by external users should be static and one-way; it should not be possible from the outside to change which confidential data items are exported from an internal network to the public Internet, or to alter existing data stored within the internal network.

S2 Confidential data should be protected end-to-end; implementation errors in an application should not result in the disclosure of confidential data and violations of the specific security policy for that application.

2.1 Case Study: A Cancer Registration System

In this work, we consider the following real-world case study. The Eastern Cancer Registry and Information Centre (ECRIC) is part of the UK National Health Service (NHS). ECRIC aims to produce a comprehensive picture of cancer cases in the East of England. It receives patient data from many sources including so-called Multidisciplinary Teams (MDTs), hospital Patient Administration Systems (PAS), pathology laboratories and the Office of National Statistics (ONS).

ECRIC's software infrastructure has recently been chosen as the national cancer registry platform for England. Most of its software systems are implemented in the Ruby programming language for ease of development and due to existing developer expertise. The main cancer registration database, hosted in a secure private network, holds structured information about patients, tumours, and associated treatments. Data are imported into the main database from different sources and processed with the help of the domain knowledge of staff. ECRIC staff can also operate off-site by using an external web application server that has been extensively audited for security.

Based on discussions with ECRIC, we identified a new application that they would like to offer: an MDT web portal that provides relevant data that can support the operation of MDTs at hospitals. MDTs treating oncological patients currently provide reports to ECRIC about their patients through secure email and paper forms. ECRIC wants to feed back both summary and detailed patient information to MDTs, letting them compare data quality with their peers and explore the underlying data to discover the cause of any discrepancies. At present, resolving discrepancies is laborious, because staff at ECRIC have to manually extract and release the relevant records for each MDT. In summary, the MDT web portal should satisfy the following functional requirements:

F1 Doctors and MDT co-ordinators that are members of an MDT can log into the MDT portal using a web browser and consult the details of patients treated by that MDT, with the option of providing feedback (handled externally, e.g. via secure NHS email).

F2 Doctors and MDT co-ordinators can consult various metrics about their patients, e.g. the level of completeness of the provided information or projected survival statistics of patients under treatment.

F3 MDT co-ordinators can put those metrics into context by comparing them with each MDT's average in the same region or with regional aggregates.

The *security policy* for the MDT web portal is as follows:

P1 Details about patients can be consulted only by members of the MDT that treats them. MDT-level aggregates can be consulted by all MDTs in the same region. Regional-level aggregates can be seen by all MDTs.

A design of the MDT web portal as a standard web application using the main ECRIC database would not be acceptable. The MDT web portal requires interactive access to patient-level data, in violation of security requirement *S1*—an implementation bug could compromise the integrity and the confidentiality of the whole ECRIC database. Furthermore, errors in the MDT web portal could violate its security policy, conflicting with *S2*.

Enforcing the MDT security policy *P1* with a traditional web application architecture is challenging. The MDT web portal has a data flow path for confidential data that involves multiple components at different layers: data must be extracted from the ECRIC database, processed in an application-specific way, and finally presented by a web front-end. Any component in the layers involved could cause unauthorised data disclosure. The security policy is difficult to enforce through a composition of local mechanisms because it is an *end-to-end* property. The large amount of source code that needs to be trusted increases the risk of defects and incurs a high code review effort.

In addition, the mechanisms used to protect from policy violations must operate at a fine data granularity. For example, the application must distinguish between confidential information at the level of single patients treated by an MDT and access to aggregated data at MDT or regional level.

In summary, we need a security mechanism that (a) is able to enforce end-to-end data flow guarantees, reducing the amount of trusted source code, and (b) allows for fine-grained, data-centric protection of confidential information. In the next section, we describe how controlling the propagation of security-labelled data through the application, at the middleware level, can achieve this.

3 Controlling Data Propagation

Traditional access control models achieve security by restricting *access to resources*: a principal, such as a user who owns data, can delegate to other principals a subset of operations. In such a model, it is easy to control information

release but difficult to control its propagation. Once a user has delegated the privilege to read data to another user, information cannot be protected anymore—e.g. the second user can write the data to a public Internet site. Thus, under a traditional (discretionary) access control model, secure data processing means that processors must be trusted—a data processor authorised to read confidential patient data must not disclose it to unauthorised parties.

The problem of unauthorised data disclosure is addressed by *information flow control* (IFC), a mandatory access control model originally developed for military systems [1,6]. IFC protects the *propagation* of data. We can model an IFC system as a set of *inputs*, *outputs* and *processing components*. An input component, acting on behalf of a principal, can attach a tamper-resistant *security label* to the data—e.g. a label can be used to protect the confidentiality of a patient medical report. The security labels can then be used to track the propagation of data through the system and middleware can enforce end-to-end restrictions on the permitted data flow. For example, if a component producing patient records labels every record, labels can prevent a mailing component from including records in emails, independently of the processing.

IFC systems can guarantee that security labels are preserved by controlling all data flow paths between components. When labelled data is copied or transformed by a component, the IFC system maintains the labels. When labelled data is processed or combined with data with different labels, the resulting labels are a composition of the previous labels, i.e. the system preserves all the data flow restrictions of the original labels. To achieve this, IFC systems require components to be "sandboxed," i.e. isolated from one another and from the external environment. Components can only communicate through primitives that are under the control of the IFC system.

To output data protected by a label, a component must have a *declassification* privilege [11]. This enables the component to remove the label from the data and use these data without restriction. The original owner of the data can restrict the data flow of, for example, a patient record by assigning declassification privileges only to components acting on behalf of treating doctors.

Labels can also be used to protect data *integrity*, which is the dual of confidentiality. The creator of an integrity label delegates to other components an *endorsement* privilege to add this label to data. Components can then trust only data that is "guaranteed" by this integrity label.

3.1 Applying Information Flow Control

IFC is a good fit for developing secure web applications because it can detect and contain the effect of application bugs which could otherwise violate a security policy (cf. security requirement *S2*). Note that we assume that application code is not intentionally malicious; this problem can be tackled by organisational safeguards such as only allowing trusted developers to develop applications. Instead, we focus on protection from unintentional software bugs.

We describe how IFC achieves our security goals in the context of the MDT web portal application from §2.1. Consider the security policy *P1* for the MDT

application. After creating a label for each patient, a component can (i) attach the label to each patient's data as it enters the system and (ii) assign the declassification privilege over the label only to components that execute on behalf of MDT principals treating the patient. Based on IFC enforcement, this guarantees that each patient record can only be accessed by the correct MDT, independently of the processing that happens between these two endpoints.

In the more complex case of MDT-level aggregate measures, which should be visible only to MDTs in the same region, label tracking is overly conservative because aggregates are considered as sensitive as the data of all involved patients, preventing access by any MDT. Therefore a component trusted with patient data must (i) remove all patient labels from aggregate data, (ii) relabel the aggregate data with an MDT-specific label and (iii) assign a declassification privilege over the MDT label to all MDTs in the region. The same mechanism can be used for regional-level aggregates by defining regional-level labels.

In summary, the security policy of the MDT application can be enforced by applying these three kinds of labels with corresponding privileges. Any component that is not policy compliant due to implementation errors cannot violate the MDT security policy. For example, a component for computing statistical aggregates would be constrained in terms of the data that it can disclose publicly. Even if its implementation is too large and complex to be audited, a software bug in, say, its logging function, which might otherwise reveal confidential patient data in externally accessible log files, would be prevented by IFC enforcement.

3.2 Practical Information Flow Control for Web Applications

In practice, applying a strong security model such as IFC to web applications is challenging. For IFC to be used, it must have low impact on developers and thus integrate with familiar architectures, programming models and languages.

Recently, researchers have proposed IFC techniques to improve the security of applications. Jif [12] extends the Java type system to include labels and checks them statically. DIFCA-J [31] rewrites Java bytecode to propagate labels in JVM operations. Trishul [13] and Laminar [19] modify the JVM to track labels. DEFCon [11] isolates threads allowing communication only through labelled data.

All of these systems adopt a "strict" tracking model that is invasive to the programmer and the system architecture. IFC tracking in these systems strives to avoid false negatives, as a single false negative if exploited could compromise the security of the whole platform. However, this strict application of IFC leads to false positives that require applications to be restructured to remove ambiguity in data flow tracking. Strict IFC also requires complex implementations: mature, industrial-strength implementations of IFC systems are currently lacking. Adoption of research prototypes in a production environment is undesirable because they are difficult to verify, requiring expert knowledge of JVMs, runtime libraries and bytecode rewriting techniques. Maintenance is also problematic when new versions are released by upstream developers.

In contrast, our IFC tracking approach is inspired by Resin [30], which only targets source code that does not actively try to evade data tracking. It is thus

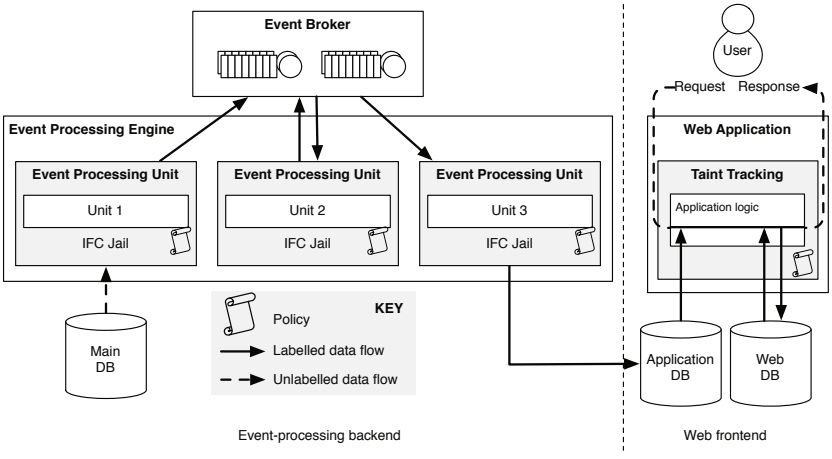


Fig. 1. Overview of the SafeWeb middleware architecture

possible to reduce the number of false positives, accepting a few false negatives which are unlikely to be exercised by non-malicious code. IFC is thus used as a safety net to catch implementation bugs, instead of acting as the primary method of security policy enforcement. This approach integrates well with current security practices adopted by organisations.

Our work differs from Resin in two key aspects. (1) We target enterprise web applications instead of stand-alone applications built directly on a specific database. In enterprise settings, it is important to support messaging services in the backend with an appropriate information processing model. To the best of our knowledge, we are the first to integrate IFC with two different processing models at different granularities (cf. §4)—a proactive event-based model in the backend with data tracking at the event level and a reactive web language model in the frontend with tracking at the variable level. (2) We manage to support IFC in the middleware without significant changes to the language runtime. By using the Ruby language support for meta-programming and its powerful security primitives, we avoid changes to the Ruby interpreter, leading to an IFC implementation that is easy to understand, verify and maintain.

4 SafeWeb Middleware Design

In this section, we describe SafeWeb, a Ruby-based middleware that separates the processing of confidential data from presentation aspects, while enforcing data flow constraints throughout the application. As shown in Figure 1, SafeWeb consists of two parts: an *event processing backend* (left), which realises the application logic, and a *web frontend* (right), which handles users' requests based on processing results. Application logic in SafeWeb is implemented in an event-based fashion through one or more processing units (i.e. application

components), which produce and/or consume events. This architecture largely decouples the processing of confidential data from the handling of web requests [29] and creates a unidirectional data flow from the backend to the frontend, in compliance with security requirement *S1*.

The event processing backend hosts the application logic for processing confidential data. *Events* are created from confidential data retrieved from a data source (illustrated as the *Main Database* in the figure) and labelled appropriately. *Event Processing Units* act as generators, filters or processors of events and exchange labelled events through an IFC-aware *Event Broker*. Units are constrained in their operation by the *Event Processing Engine*, which acts as a run-time environment for application components. Its *IFC Jail* controls communication of units with the environment and preserves labels during event communication. *Privileges* for units over specific labels are configured through a data flow *Policy*. Result events are stored with appropriate labels in an *Application Database* after processing.

The web frontend serves synchronous web requests from users by accessing the application database. State that is specific to a given web session is stored in a separate *Web Database* to isolate it from application data. Labels from the application database are propagated in the web application by SafeWeb's runtime *Taint Tracking* library and are checked when generating responses. As a result, security labels are associated with data throughout the processing pipeline and checked at boundaries between components with respect to the application policy, thus satisfying requirement *S2*.

4.1 Events with Security Labels

Event processing units communicate through events. Data models for events can vary widely [7]. For ease-of-use, we adopt one of the simplest yet popular choices: events in SafeWeb consist of a set of key-value attribute pairs and an optional data payload. The keys, values and the body are untyped strings.

SafeWeb associates a set of security labels with each event. There are two types of labels, *confidentiality* labels and *integrity* labels. Confidentiality labels prevent sensitive data from escaping a given system boundary, whereas integrity labels are used to prevent low-integrity data from entering parts of an application. Confidentiality labels are "sticky"—once they are associated with an event, all events that are derived from that event will also contain those labels. In contrast, integrity labels are "fragile"—they are applied to an event only if all the events that this event was derived from also contain the same integrity labels.

Labels are represented as URIs. For example, `label:conf:ecric.org.uk/patient/33812769` could be used as a confidentiality label to protect the data of a specific patient, while `label:int:ecric.org.uk/mdt` could act as the integrity label for all data contained within the whole MDT application. In the MDT application, an event processing unit periodically reads unlabelled patient records from the main ECRIC database and produces events which are labelled according to the encountered patient ID. This operation does not require privileges—it is always possible to add extra confidentiality labels to events. MDT-level

aggregates, such as survival statistics or measures of information completeness (cf. §2.1), are labelled with a confidentiality label specific to that MDT.

Label enforcement is managed using associated privileges. Two types of privileges are used for confidentiality labels. The *clearance* privilege is used to access information protected by a confidentiality label. The privilege to make labelled information public by removing the label is referred to as *declassification*. Analogous privileges for integrity labels exist: *clearance* to low integrity and *endorsement*. To simplify presentation, we consider only confidentiality labels and the associated privileges in the rest of the paper.

Privilege assignment and checking is performed in SafeWeb by the event processing engine and the web frontend. Privileges associated with labels are assigned directly to units (in the backend) and requests (in the frontend) through a policy specification file. For more complex policies with dynamic privileges, a label manager could delegate privileges to units at runtime.

4.2 Event Broker

Units communicate by publishing events and by subscribing to events that they are interested in. To dispatch events among units, SafeWeb uses an event broker that matches subscriptions with published events. To support fine-grained data processing, SafeWeb uses a topic-based subscription language with optional content filtering on event attributes within a topic [7].

The event broker filters events according to their security labels. This is used to restrict the set of events that units can receive: for an event to be delivered to a subscriber, the set of its confidentiality labels must be a subset of those labels for which the subscriber possesses clearance privileges.

The event broker uses a modified version of the *Streaming Text Oriented Message Protocol* (STOMP) [23]—a simple, extensible, HTTP-inspired message protocol. It is language- and platform-agnostic and an open-source implementation [24] exists for Ruby. In STOMP, each request consists of a command, such as `CONNECT`, `SEND` or `SUBSCRIBE`, a set of optional headers and an optional body. A `destination` header is used to match subscriptions with publications by topic. An optional SQL-92 selector header specifies content-based subscriptions.

The implementation of our IFC-aware event broker is based on the STOMP implementation but has been extended with SSL support at the transport layer. At the dispatching layer, we have changed the matching semantics to respect labels, which are encoded as event headers with special semantics in `SEND` and `SUBSCRIBE` messages. In addition, subscriptions include unique identifiers to simplify the handling of subscriptions issued by different units. The client side of the STOMP implementation uses the event-based EventMachine I/O library [8].

4.3 Event Processing Engine

The event processing engine in SafeWeb provides a framework to support and control unit execution. Its key functions are (1) control of unit execution by checking and tracking security labels, (2) assignment of privileges to units and

```
1  subscribe '/patient_report', 'type=cancer' do |event|
2    list = get 'patient_list'
3    list push event[:patient_id]
4    set 'patient_list', list
5  end
6  subscribe '/next_day' do |event|
7    list = get 'patient_list'
8    publish '/daily_report', list, :remove => $LABELS,
9      :add => ['label:conf:ecric.org.uk/patient_list']
10 end
```

Listing 1. Example unit

(3) restriction of access to the environment. An event processing unit is realised as one or more classes that implement the business logic of the application. The engine configures, instantiates and runs units and provides communication facilities using the event broker.

Listing 1 shows a unit that computes a daily list of patients with processed reports. The unit registers subscriptions for events published on the topics `patient_report` (line 1) and `next_day` (line 6). When a subscription is issued by a unit, the engine reads the set of labels from the unit's policy file for which the unit has *clearance* privileges. The engine then issues a subscription request to the broker with this set of labels; this set is used to check that a matching event can be processed by the unit. To support stateful units, the engine provides a unit-specific key-value store with labels associated with keys. It can be used for reading (lines 2 and 7) or storing (line 4) values, thus allowing different callbacks to communicate by exchanging state between them.

The engine prevents units from inadvertently disclosing confidential data because it controls the labelled events that they publish and isolates them from the external environment. We describe the two mechanisms for this in turn.

Label tracking. To ensure correct labelling, the engine associates a set of labels with the execution of a callback. This set, accessible to units as `$LABELS`, is initialised to the set of labels of the event being processed. When an event is published, the engine attaches all labels in `$LABELS` to the event. With each `publish` call, the unit can specify a set of labels to add or remove from the published event (lines 8 and 9), although removal is only permitted when the unit has the appropriate declassification privilege.

As values in the key-value store are labelled on a per-key basis, when a value is read from the store, `$LABELS` is updated to reflect its confidentiality—all the labels associated with the value's key are added to `$LABELS`. When writing to the store, all confidentiality labels in `$LABELS` are saved as the key's confidentiality, optionally adding and removing labels analogous to the `publish` call.

By maintaining labels from the received events to the published ones, and by labelling all datapaths through the shared key-store appropriately, confidentiality labels are preserved. However, unit callbacks could access other forms of unlabelled shared state, which would ignore label protection. In addition, they could bypass the event broker and use external APIs for console, disk or network I/O, thus disregarding labels completely. To prevent this, the engine must execute unit callbacks in isolation.

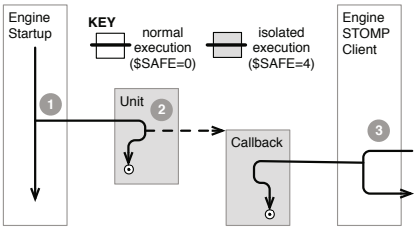


Fig. 2. Isolation of units and callbacks performed by the event processing engine

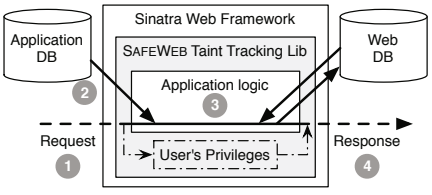


Fig. 3. Variable-level taint tracking in the web frontend

Isolation. The goals of isolation are twofold: to prevent the use of I/O operations and to prevent access to variables outside of the callback local scope, i.e. , global variables, instance variables and local variables of enclosing scopes.

To isolate units from I/O and access to global variables, we use Ruby's *safe levels*. They restrict the execution mode of Ruby code and provide different kinds of isolated environments. `$SAFE` is a thread-local global variable that controls the current safe level. When set to safe level 4, it creates the most restrictive environment with the following irreversible effects on the current thread: (i) no access to I/O operations, (ii) all new objects are marked with a flag called "taint" and (iii) no write access to any object that is not tainted.

In Figure 2, we show how safe levels are used to achieve callback isolation. The engine executes units in a new thread, after setting `$SAFE=4` to prevent the unit's initialisation code (step 1) from performing I/O operations. Units register callbacks (step 2) that execute when events arrive. When the STOMP client library that interacts with the event broker receives a matching event, it creates a new thread, sets `$SAFE=4` and executes the callback (step 3). The callback code cannot perform I/O or store events in global variables. It can only store events in the unit's key-value store that is tainted explicitly during unit allocation.

Isolation in safe level 4 still allows a callback to access variables of its enclosing scopes. To prevent this, we duplicate these variables when the callback is registered by using the meta-programming features of the Rubinius runtime [20].

Some units, however, need access to APIs that perform I/O, e.g. units that import and export events between the event engine and databases. To support this, the engine allows *privileged* units to execute without isolation at `$SAFE=0` and, thus, access I/O facilities. This effectively allows them to declassify any received event. To limit the power of privileged units, the engine prevents them from receiving events with certain labels by withholding their clearance privilege.

4.4 Web Frontend

The web frontend of SAFEWEB presents results from the backend to users and enforces IFC without requiring changes to web applications. Web developers could be not fully aware of the policy requirements of the data that they present or, more often, they may introduce implementation bugs leading to unintended

data disclosure. In the web frontend, SafeWeb's taint tracking library labels all data to reflect the confidentiality of the principals that the data correspond to. When an application that is not policy compliant attempts to return incorrect data to the client, the operation can be aborted, preventing data disclosure.

The web frontend has a traditional, database-driven architecture: a client issues an HTTP request, the request is served by the application server using the application database, and the HTTP response returns the result to the client. Since the application server handles requests from all users, it must have access to the data that *any* user may receive, i.e. all sensitive data in the application database. As a consequence, the web application would have to be trusted to remove all labels associated with data. Clearly, this would violate security requirement *S2* because any implementation error in the web application could result in inadvertent disclosure of data that should be visible only to a particular group of users, such as a given MDT.

To achieve the end-to-end security requirement, SafeWeb tracks data at a different granularity in the web frontend than in the event-processing backend. Instead of labels being attached to events, they are associated with individual *variables*. For example, when a variable n stores a patient name, n will carry a label that conveys the confidentiality of the patient name.

Labels are checked by SafeWeb when the web application returns an HTTP response to a client. For example, before the content of variable n is sent to a client, the client's privileges are validated to be a superset of the confidentiality labels associated with n. As described next, this is sufficient to provide end-to-end confidentiality guarantees without requiring a new application architecture, which would be challenging to adopt in a production environment [15].

Figure 3 shows the operation of SafeWeb's taint tracking library for Ruby. In step 1, an HTTP request arrives at the server. The request is authenticated and the confidentiality privileges of the associated user are retrieved from the web database. In step 2, the application queries the application database for the data needed to serve the request. SafeWeb's taint tracking library transparently adds the labels produced by units in the backend to the data fetched from the application database. In step 3, the application produces a response by carrying out application-specific processing of the data. SafeWeb's taint tracking library alters Ruby program statements and library methods to propagate labels correctly; e.g. , when two strings are concatenated, the resulting string receives both operands' labels. In step 4, before sending the response to the user, the response's label is compared to the user's privileges from step 1—unless the user has the required privileges, the operation is aborted.

SafeWeb implements variable taint-tracking in Ruby using labels as follows. Its taint-tracking library redefines Ruby's `String` and `Numeric` subclasses (1) to store labels within each instance and (2) to propagate labels correctly across method invocations. For example, SafeWeb's taint tracking library should propagate labels upon string concatenation. For this, it declares a new concatenation method in the `String` class called `nconcat()`. The taint tracking library then aliases the existing "`+`" method to call `nconcat()` and propagate

labels. From then onwards, the runtime transparently invokes the redefined "`+`" method when two strings are concatenated. Since we only support non-malicious code (§3.2), these changes are enough to effect label propagation.

The implementation exploits a standard meta-programming feature of Ruby and Ruby's pure object-oriented foundations. Ruby classes are open, all operators are defined as methods, and method definitions can change at any time. Implementing a similar taint-tracking library in other popular web languages, such as Java [4] or PHP [30], would require more extensive changes to the language runtime, making maintenance difficult in a production environment.

SafeWeb supports Ruby web applications running on the Rubinius runtime [20] and using the Sinatra web framework [22]. We use Rubinius due to its ability to manipulate the regular expression variables (`$~`, `$1`, etc.) directly. This is necessary to add taint tracking to Ruby's regular expression methods. Sinatra is used for its well-defined interception points of HTTP requests and responses. This allows SafeWeb's taint tracking library to intercept all communication to and from the client and attach label checks or fetch user privileges.

We do not introduce explicit features to prevent traditional Cross-Site Scripting (XSS) or SQL injection (SQLI) attacks. Ruby objects support a `taint` method that marks a given object as originating from the user. The Ruby runtime stores this information per object and propagates it when strings are processed, similar to our label propagation. In the context of web applications, this mechanism can be used to ensure that every string is sanitised before being used in a sensitive operation, such as an HTML response or an SQL query.

5 Evaluation

The goals of our experimental evaluation are to explore the effectiveness of the SafeWeb middleware in preventing unauthorised data disclosure and to measure its performance overhead. We evaluate its security properties as part of the prototype implementation of the MDT web portal application.

5.1 Case Study: MDT Web Portal Application

As shown in Figure 4, the MDT web portal application uses three units: (a) A *data producer* obtains data from the main ECRIC database, leveraging the existing ECRIC framework for data access. It reads fields from different tables, labels them appropriately according to MDT and patient ID and publishes them as events to the event broker. For the sake of simplicity, we use only MDT-level labels as these are sufficient to satisfy our security requirements. (b) A *data aggregator* continuously collects all events related to individual cancer cases and combines their data. It produces aggregated records required by the MDT web application to satisfy functional requirements *F1–F3*. Implementation errors will not disclose data because of the isolation mechanism of SafeWeb. (c) Finally, a *data storage* unit, which has declassification privileges for all MDTs, handles data persistence. It stores processed records with their security labels in the

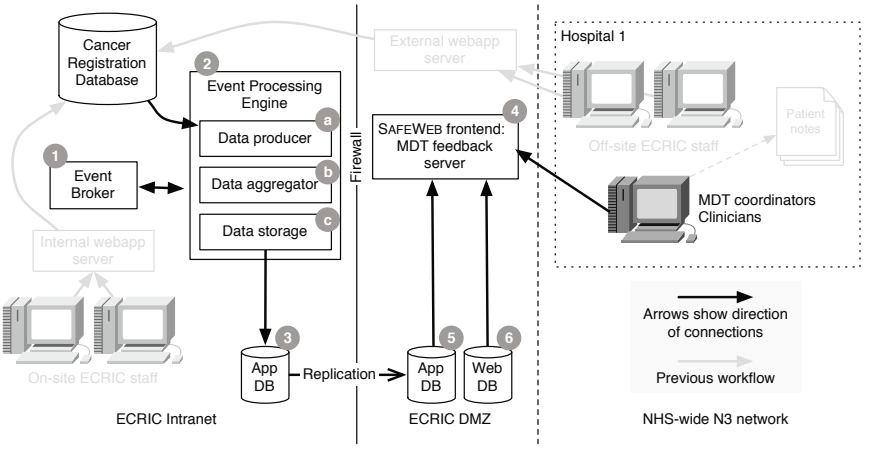


Fig. 4. Deployment of the MDT web portal application using SafeWeb within ECRIC's infrastructure

CouchDB application database. Security features of SafeWeb (i.e. IFC and strong isolation) allow the application to satisfy the security policy *P1*.

The Sinatra-based web frontend of the application uses CouchRest [5] to access CouchDB, and ERB for embedding Ruby in web pages. SafeWeb's taint tracking library enforces authentication centrally by adding hooks to all defined Sinatra rules. User accounts and their label privileges are stored in the web database. Currently, the web frontend uses HTTP basic authentication and TLS. We plan to add support for authentication using NHS smartcards in the future.

Deployment. Figure 4 shows how ECRIC's network is separated into three zones: an *Intranet*, a demilitarised zone (*DMZ*) and the NHS-wide *N3 network*. The Intranet is a restricted zone separated from the DMZ by a firewall, which permits only unidirectional connections to the DMZ. Core ECRIC infrastructure such as the main database is accessible only from within the Intranet.

The MDT web portal application is deployed within ECRIC as follows. The event broker (1) acts as a secure event bus for event processing units and is deployed within the ECRIC internal network. The units belonging to the application execute as part of the event processing engine (2). The MDT application uses a CouchDB application database (3) which contains the result data from the event-processing backend and provides result data to the web frontend (4). Because ECRIC's firewall only permits connections from the Intranet to the DMZ, we run two instances of the application database: in the Intranet (3) and in the DMZ (5). The application database is replicated periodically between the two instances using CouchDB push replication. The DMZ instance is read-only in order to prevent modifications by the web frontend, thus satisfying requirement *S1*. Data specific to the web frontend, e.g. session and usage data, is stored separately in a local web database (6) using the SQLite database engine.

```
1  require `sinatra'
2  require `safeweb−tracking'
3  get '/records/:mid' do
4    content_type :json
5    return nil if !check_privileges(params[:mid])
6    r = Records.by_mid(:key => params[:mid])
7    process r
8    r.to_json
9  end
```

Listing 2. Example of a rule in the web frontend of the MDT web portal

```
1  def check_privileges id
2    m = Measurement.find(id)
3    @is_admin or Privileges.count(
4    :conditions => {
5      :u_id => User.find_by_name(@username).id,
6      :hospital => m.hospital_id,
7      :clinic => m.type
8    }) > 0
9  end
```

Listing 3. An access control check used by the MDT web portal

5.2 Security Properties

Given the lack of third-party SafeWeb applications, we assess the security properties of SafeWeb by analysing its effectiveness in defending against known types of implementation errors. We inspected the Common Vulnerabilities and Exposures (CVE) database for vulnerabilities classified as "Information Disclosure", "Access Control" or "Design Error" and organised them into generic subcategories that share the same underlying cause. We then inject similar vulnerabilities to our MDT application and observe if SafeWeb can prevent them.

Omitted Access Checks. The most common problem that leads to information disclosure (e.g. , CVE reports 2011-0701, 2010-2353 and 2010-0752) is the omission of access control checks. To emulate this, we remove the MDT privilege check from the patient filtering routine that normally precedes the filtering of patient details (Listing 2, line 5). Without SafeWeb's taint tracking library (line 2), sensitive information disclosure occurs in line 8. However, when the taint tracking library is included and an MDT requests data they are not allowed to see, the library correctly taints the JSON string (line 8) and displays an error message.

Errors in Access Checks. Even when an access control check is present, it may not specify the correct security policy and may result in information disclosure. Often, these errors involve specially constructed input and do not manifest themselves under normal operation, making them hard to discover (e.g. , CVE reports 2011-0449, 2010-3092 and 2010-4403). To introduce such a problem, we modify the user lookup operation (listing 3, line 5) to ignore the case of the username. This may lead to two MDTs sharing privileges. To test this, we create two MDTs with usernames `mdt1` and `MDT1` but with different privileges. SafeWeb's taint tracking library, when included, successfully prevents access of the second MDT to all the patient details that only the first MDT should see.

Inappropriate Access Checks. Security policies are often complicated and developers may not fully understand them. This category of vulnerabilities covers correctly applied checks that do not enforce the intended policy (e.g. , CVE reports 2010-4775 and 2009-2431). To emulate such issues, we remove the check for clinic equality from `check_privileges` (Listing 3, line 7). This effectively

enables any MDT to see the data of all the patients in the same hospital. Again, the error does not result in information disclosure: SafeWeb's taint tracking library detects the taint of the output, generates an error and prevents access.

Design Errors. The last category captures vulnerabilities due to design errors in the application's business logic (e.g. , CVE reports 2011-0899 and 2010-3933). Such errors involve the application processing sensitive data in unexpected ways that lead to data disclosure. To emulate this, we modify the data aggregator unit to ignore the hospital of origin when matching events. As a result, the unit generates records that mix data of different MDTs. SafeWeb's event processing backend allows access to these events but requires that the output records have labels of all relevant MDTs. Thus, when the frontend attempts to display these records, access is prevented because no MDT has the necessary privileges.

Trusted Codebase. SafeWeb enforces security policies but it does not eliminate trusted code: (1) SafeWeb's taint tracking library must be trusted to correctly authenticate users, associate privileges with their requests, propagate labels and check the labels of each response. (2) The event backend must isolate non-privileged units and label their output. (3) Privileged units must label events that they publish or store in the application database correctly. (4) The policy file that specifies user privileges in the web frontend and unit privileges in the event backend, as well as the scripts that edit it, must be audited for errors.

A code audit is still required, but it can focus on the SafeWeb implementation and only includes a small application-specific part. SafeWeb's taint tracking library consists of 1943 LOC and the event processing engine has 1908 LOC. After this trusted codebase has been audited once, data confidentiality only depends on the correctness of a small part of each application. For the MDT web portal, the code that has to be audited involves the two privileged units in the backend (138 LOC) and the code that assigns privileges to new MDTs in the frontend (142 LOC). The confidentiality of patient data does not depend on the other 2841 LOC of the MDT application—no further security audit is required.

5.3 Performance Overhead

In this section, we measure the performance overhead of SafeWeb in terms of latency and throughput. All measurements are taken on an AMD Opteron 6136 2.4GHz system with 16 GiB of RAM running Ubuntu 10.04. The 95% confidence interval for each value we report extends to each side at most 5% of the value.

For the web front-end, we measure the page generation time of the MDT application's front page with and without SafeWeb's taint tracking library. We issued 1000 requests and measured the time required to render the response. With SafeWeb's taint tracking library, the page generation time increases by 14% from 158 ms to 180 ms. For the back-end, we measure the average latency of individual events from the data producer to the data storage unit during the processing of 1000 events. With SafeWeb's isolation and label checks, the latency to process a single event increases by 15% from 73 ms to 84 ms. Overall, this is an acceptable overhead for a web application with strong security requirements.

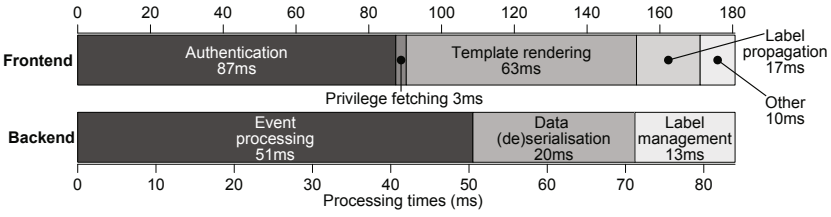


Fig. 5. Processing latency within the MDT web portal application

Figure 5 shows a break-down of the overall latency when SafeWeb is enabled. In the front-end, HTTP basic authentication takes 87 ms to which privilege fetching adds 3 ms; processing of the `ERB` template takes 63 ms, to which label propagation adds 17 ms. "Other" includes operations like network transmission and database access. For the back-end, processing an event takes 51 ms plus 20 ms for serialisation, to which SafeWeb adds 13 ms for label management including label (de)serialisation and checking.

SafeWeb provides ample throughput for the low event rates of the MDT portal. We employ a synthetic benchmark with two units, an event producer and an event consumer, to measure the throughput reduction that SafeWeb incurs. We measured the end-to-end event throughput between the two units by having the producer publishing events at the maximum sustainable rate while confirming that the memory consumption remained stable. We sampled the throughput once per second for 1000 seconds. With label tracking active, event throughput decreased (−17%) from 4455 events/second to 3817 events/second. Due to the language isolation support in Ruby, the decrease in performance is minimal and comparable to approaches that rely on low-level runtime modifications [30].

6 Related Work

The most common security problems in web applications arise from the handling of untrusted user data in the application's output. This leads to problems such as XSS and SQL injection attacks. Web application frameworks protect against such vulnerabilities (e.g. XSS, CSRF) [26]. Applications developed with SafeWeb can still benefit from this (e.g. RailsXSS [18], Rack::Csrf [17]) to avoid traditional exploits that often disclose data by hijacking user accounts. In addition, SafeWeb improves these frameworks to prevent sensitive data disclosure.

Such data disclosure is often caused by insufficient authorisation, missing access control checks or by errors in application semantics. Potential solutions include static analysis [9,10], symbolic execution [3] and runtime taint tracking [4,28,14,16,30]. Static analysis tools for dynamic web languages often have high false positive rates [9] or do not support all language features [10]. Symbolic execution explores all possible execution paths of a web application and can prove absence of certain errors [3]. However, devising assertions for symbolic

execution is a manual task and involves many of the shortcomings of manual security audits. In contrast, our approach requires minimal developer involvement.

At runtime, access control can be enforced transparently on each operation [21,2] or when sensitive events are received [27]. Nevertheless, if the application has to process sensitive data, errors in the application logic may still convey sensitive information. Taint tracking systems transparently track sensitive data and protect against inadvertent disclosure despite application errors. They have been provided, amongst others, for Java [4], C [28,14], PHP [16,30] and Ruby [2]. Simple approaches use one bit taint per string for injection attack protection [16]. In contrast, SafeWeb's taint tracking library attaches full security labels to each variable, offering end-to-end guarantees about sensitive data disclosure. Resin [30] uses pointers to user-defined policy objects, such as IFC labels; however, it requires extensive language runtime modifications.

7 Conclusion

We have designed and implemented SafeWeb, a middleware for creating secure, event-based enterprise web applications. It provides strong end-to-end security guarantees, while integrating with existing web development practices. We have demonstrated SafeWeb as part of a web application for assisting cancer treatment practices within the UK National Health Service (NHS).

The strict data security requirements across multiple interacting organisations provided us with a set of real design constraints. The sensitivity of healthcare data required careful consideration of the parts of the middleware that push and pull data. The back-end requirements suited an event-driven design, whereas the front-end is a typical web application. We showed that information flow control can be applied to both the event-processing back-end and the web front-end as part of a middleware. This gives security assurances regarding data disclosure and minimises organisations' code audits.

In future work, we plan to explore how SafeWeb could become the basis for wider deployment of healthcare applications at the national level. Scaling up will involve creating separate, independent regional instances of SafeWeb, which can interact with each other in a secure fashion. In addition, we want to investigate the use of SafeWeb for other classes of web applications.

References

1. Bell, D., LaPadula, L.: Secure computer system: Unified exposition and Multics interpretation. Technical report, MITRE Corporation (1976)
2. Burket, J., Mutchler, P., Weaver, M., Zaveri, M., Evans, D.: GuardRails: A data-centric web application security framework. In: WebApps, pp. 1–12. USENIX, Portland (2011)
3. Chaudhuri, A., Foster, J.: Symbolic security analysis of Ruby-on-Rails web applications. In: Computer and Communications Security. ACM, Chicago (2010)
4. Chin, E., Wagner, D.: Efficient character-level taint tracking for Java. In: Workshop on Secure Web Services (SWS), pp. 3–12. ACM, Chicago (2009)

5. CouchRest, http://github.com/couchrest (Accessed September 5, 2011)
6. Department of Defense. Trusted Computer System Evaluation Criteria (1983)
7. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Computing Surveys 35(2), 114–131 (2003)
8. EventMachine, http://rubyeventmachine.com (Accessed September 5, 2011)
9. Huang, Y.-W., Yu, F., et al.: Securing web application code by static analysis and runtime protection. In: World Wide Web (WWW). ACM, New York (2004)
10. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities. In: Symposium on Security and Privacy, pp. 258–263. IEEE, Berkeley (2006)
11. Miglivacca, M., Papagiannis, I., Eyers, D., Shand, B., Bacon, J., Pietzuch, P.: High-performance event processing with information security. In: USENIX Annual Technical Conference, Boston, MA (2010)
12. Myers, A., Liskov, B.: Protecting privacy using the decentralized label model. Transactions on Software Engineering and Methodology 9(4), 410–442 (2000)
13. Nair, S., Simpson, P., Crispo, B., Tanenbaum, A.: A virtual machine based information flow control system for policy enforcement. Electronic Notes in Theoretical Computer Science 197(1), 3–16 (2008)
14. Nanda, S., Lam, L.-C., Chiueh, T.-C.: Dynamic multi-process information flow tracking for web application security. In: Middleware. ACM, Toronto (2007)
15. Papagiannis, I., Migliavacca, M., Eyers, D.M., Shand, B., et al.: Enforcing user privacy in web applications using Erlang. In: W2SP, Oakland, CA (2010)
16. Pietraszek, T., Berghe, C.: Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 124–145. Springer, Heidelberg (2006)
17. Rack:Csrf, http://github.com/baldowl/rack_csrf (Accessed September 5, 2011)
18. RailsXSS, http://github.com/rails/rails_xss (Accessed September 5, 2011)
19. Roy, I., Porter, D., Bond, M., McKinley, K., Witchel, E.: Laminar: Practical fine-grained decentralized information flow control. In: PLDI, Dublin, Ireland (2009)
20. Rubinius, http://rubini.us (Accessed September 5, 2011)
21. Ryck, P.D., Desmet, L., Joosen, W.: Middleware Support for Complex and Distributed Security Services in Multi-Tier web Applications. In: Erlingsson, Ú., Wieringa, R., Zannone, N. (eds.) ESSoS 2011. LNCS, vol. 6542, pp. 114–127. Springer, Heidelberg (2011)
22. Sinatra, http://www.sinatrarb.com (Accessed September 5, 2011)
23. Stomp protocol, http://stomp.github.com (Accessed September 5, 2011)
24. StompServer, http://stompserver.rubyforge.org (Accessed September 5, 2011)
25. UK Information Commissioner's Office. Data breaches to incur up to £500,000 penalty, http://www.ico.gov.uk/ /media/documents/pressreleases/2010/PENALTIES_GUIDANCE_120110.ashx (Accessed September 5, 2011)
26. Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R., Song, D.: An empirical analysis of XSS sanitization in web application frameworks. Technical report, UC Berkeley (2011)
27. Wun, A., Jacobsen, H.-A.: A Policy Management Framework for Content-Based Publish/Subscribe Middleware. In: Cerqueira, R., Pasquale, F. (eds.) Middleware 2007. LNCS, vol. 4834, pp. 368–388. Springer, Heidelberg (2007)
28. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: Security Symposium, pp. 121–136. USENIX, Vancouver (2006)

29. Ye, C., Jacobsen, H.-A.: Event Exposure for Web Services: A Grey-Box Approach to Compose and Evolve Web Services. In: Chignell, M., Cordy, J., Ng, J., Yesha, Y. (eds.) The Smart Internet. LNCS, vol. 6400, pp. 197–215. Springer, Heidelberg (2010)
30. Yip, A., Wang, X., Zeldovich, N., Kaashoek, M.F.: Improving Application Security With Data Flow Assertions. In: SOSP. ACM, Big Sky (2009)
31. Yoshihama, S., Yoshizawa, T., Watanabe, Y., Kudoh, M., Oyanagi, K.: Dynamic Information Flow Control Architecture for Web Applications. In: Biskup, J., López, J. (eds.) ESORICS 2007. LNCS, vol. 4734, pp. 267–282. Springer, Heidelberg (2007)

Author Index