Jim Dowling
François Taïani (Eds.)

LNCS 7891

Distributed Applications and Interoperable Systems

13th IFIP WG 6.1 International Conference, DAIS 2013
Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013
Florence, Italy, June 2013, Proceedings

ifip

Springer

Jim Dowling   François Taïani (Eds.)

Distributed Applications and Interoperable Systems

13th IFIP WG 6.1 International Conference, DAIS 2013
Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013
Florence, Italy, June 3-5, 2013
Proceedings

Springer

Volume Editors

Jim Dowling
Royal Institute of Technology (KTH)
Isafjordsgatan 39, 16440 Kista, Sweden
E-mail: jdowling@kth.se

François Taïani
Université de Rennes 1, IRISA
263 Avenue du Général Leclerc, Bât. 12, 35042 Rennes, France
E-mail: francois.taiani@irisa.fr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-38540-7 e-ISBN 978-3-642-38541-4
DOI 10.1007/978-3-642-38541-4
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013938373

CR Subject Classification (1998): C.2, C.4, H.4, D.2, D.4, C.3, K.6

LNCS Sublibrary: SL 5 – Computer Communication Networks
and Telecommunications

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)

Foreword

In 2013 the 8th International Federated Conference on Distributed Comput- ing Techniques (DisCoTec) took place in Florence, Italy, during June 3–6. It was hosted and organized by Università di Firenze. The DisCoTec series of federated conferences, one of the major events sponsored by the International Federation for Information processing (IFIP), included three conferences:

- The 15th International Conference on Coordination Models and Languages (Coordination)
- The 13th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)
- The 2013 IFIP Joint International Conference on Formal Techniques for Distributed Systems (33rd FORTE/15th FMOODS)

Together, these conferences cover the complete spectrum of distributed computing subjects ranging from theoretical foundations to formal specification techniques to systems research issues.

Each of the first three days of the federated event began with a plenary speaker nominated by one of the conferences. The three invited speakers were: Tevfik Bultan, Department of Computer Science at the University of California, Santa Barbara, UCSB; Gian Pietro Picco, Department of Information Engineering and Computer Science at the University of Trento, Italy; and Roberto Baldoni, Department of Computer, Control and Management Engineering "Antonio Ruberti", Università degli Studi di Roma "La Sapienza", Italy. In addition, on the second day, there was a joint technical session consisting of one paper from each of the conferences. There were also three satellite events:

1. The 4th International Workshop on Interactions Between Computer Science and Biology (CS2BIO) with keynote talks by Giuseppe Longo (ENS Paris, France) and Mario Rasetti (ISI Foundation, Italy)
2. The 6th Workshop on Interaction and Concurrency Experience (ICE) with keynote lectures by Davide Sangiorgi (Università di Bologna, Italy) and Damien Pous (ENS Lyon, France)
3. The 9th International Workshop on Automated Specification and Verification of Web Systems (WWV) with keynote talks by Gerhard Friedrich (Universität Klagenfurt, Austria) and François Taïani (Université de Rennes 1, France)

I believe that this program offered each participant an interesting and stimulating event. I would like to thank the Program Committee Chairs of each conference and workshop for their effort. Moreover, organizing DisCoTec 2013 was only possible thanks to the dedicated work of the Publicity Chair Francesco Tiezzi (IMT Lucca, Italy), the Workshop Chair Rosario Pugliese (Università di Firenze, Italy), and members of the Organizing Committee from Università di Firenze: Luca Cesari, Andrea Margheri, Massimiliano Masi, Simona Rinaldi and Betti Venneri. To conclude I want to thank the International Federation for Information Processing (IFIP) and Università di Firenze for their sponsorship.

June 2013 Michele Loreti

Preface

This volume contains the proceedings of DAIS 2013, the 13th IFIP International Conference on Distributed Applications and Interoperable Systems, sponsored by IFIP (International Federation for Information Processing) and organized by the IFIP Working Group 6.1.

DAIS was held during June 3–5, 2013 in Florence, Italy, as part of the DisCoTec (Distributed Computing Techniques) federated conference, together with the International Conference on Formal Techniques for Distributed Systems (FMOODS & FORTE) and the International Conference on Coordination Models and Languages (COORDINATION). There were 42 submissions for DAIS. Each submission was reviewed by at least 3, and on average 3.9, Program Committee members. The committee decided to accept 12 full papers and six short papers, giving an acceptance rate of 28% for full research papers.

The conference program presented state-of-the-art research results and case studies in the area of distributed applications and interoperable systems. The main themes of this year's conference were cloud computing, replicated storage, and peer-to-peer computing.

In the area of cloud computing, there are papers on security, adaptive replicated services, network forensics for the cloud, autnomously adapting applications, a benchmark-as-a-service, and building an ambient cloud for mobile ad-hoc networks. A significant number of the papers cover replicated storage, including providing SQL support for NoSQL databases, strengthening consistency for the Cassandra key-value store, using application-level knowledge to improve replication consistency models, improving transaction processing throughput for optimistic concurrency control through adaptive scheduling, and a study of the cost of consistency models in distributed filesystems. Two papers are on peer-to-peer computing, including algorithms for generating scale-free overlay topologies, and a model for a peer-to-peer-based virtual microscope. We also had papers on deploying experiments on smartphones, bandwidth prediction, asynchronous protocol gateways, and decentralized workflow scheduling.

Finally, we would like to take this opportunity to thank the many people whose work made this conference possible. We wish to express our deepest gratitude to the authors of submitted papers, to all PC members for their active participation in the paper review process, and to all external reviewers for their help in evaluating submissions. We would like to thank the Steering Committee of DAIS, and in particular the Chair Rui Oliviera, for their advice and help. We would also like to thank Roberto Baldoni, our invited keynote speaker. Further thanks go to the University of Florence for hosting the event in Florence, to the past DAIS Chairs Karl Göschka and Seif Haridi for their useful advice and documentation, and to Michele Loreti for acting as a General Chair of the joint event.

April 2013

Jim Dowling
François Taïani

Organization

Steering Committee

Rui Oliveira (Chair)	Universidade do Minho, Portugal
Frank Eliassen	University of Oslo, Norway
Pascal Felber	Université de Neuchâtel, Switzerland
Karl Göschka	Vienna University of Technology, Austria
Seif Haridi	KTH, Sweden
Rüdiger Kapitza	Technical University of Braunschweig, Germany
Romain Rouvoy	University of Lille 1, France

Program Chairs

Jim Dowling	KTH, Sweden
François Taïani	Université de Rennes 1/IRISA, France

Program Committee

Jean Bacon	University of Cambridge, UK
Carlos Baquero	Universidade do Minho, Portugal
Thais Batista	Federal University of Rio Grande do Norte, Brazil
Gordon Blair	Lancaster University, UK
Yerom-David Bromberg	LabRI, France
Denis Conan	Institut Télécom; Télécom SudParis/CNRS UMR SAMOVAR, France
Domenico Cotroneo	University of Naples Federico II, Italy
Wolfgang De Meuter	Vrije Universiteit Brussel, Belgium
Tudor Dumitras	Symantec Research Labs, USA
Paulo Ferreira	INESC ID/Technical University of Lisbon, Portugal
Davide Frey	INRIA, France
Kurt Geihs	Universität Kassel, Germany
Paul Grace	Lancaster University, UK
Yanbo Han	Institute of Computing Technology, CAS, China
Franz J. Hauck	Ulm University, Germany
Peter Herrmann	NTNU Trondheim, Norway
Matti Hiltunen	AT&T Labs Research, USA

Mark Jelasity	University of Szeged, Hungary
Wouter Joosen	Katholieke Universiteit Leuven, Belgium
Boris Koldehofe	University of Stuttgart, Germany
Reinhold Kroeger	Wiesbaden University of Applied Sciences, Germany
Edmundo Madeira	IC/UNICAMP, Brazil
Kostas Magoutis	ICS-FORTH, Greece
Rene Meier	Lucerne University of Applied Sciences, Switzerland
Hein Meling	University of Stavanger, Norway
Pietro Michiardi	EURECOM, France
Alberto Montresor	University of Trento, Italy
Nearchos Paspallis	UCLan Cyprus, Cyprus
Amir Payberah	KTH - Royal Institute of Technology, Sweden
Jose Pereira	University of Minho, Portugal
Guillaume Pierre	Université de Rennes 1, France
Peter Pietzuch	Imperial College London, UK
Etienne Rivière	University of Neuchatel, Switzerland
Giovanni Russello	The University of Auckland, New Zealand
Marc Sánchez-Artigas	Rovira i Virgili University, Spain
Lionel Seinturier	Univ. Lille 1 and IUF - LIFL and Inria ADAM, France
Luís Veiga	Instituto Superior Técnico - UTL/INESC-ID Lisboa, Portugal
Spyros Voulgaris	VU University, The Netherlands
Roland Yap	National University of Singapore, Singapore

Additional Reviewers

Azab, Abdulrahman	Jehl, Leander	Nzekwa, Russel
Azevedo, Paulo J.	Kambona, Kennedy	Onica, Emanuel
Baraki, Harun	Kapitza, Ruediger	Opfer, Stephan
Bieniusa, Annette	Kloudas, Konstantinos	Pecchia, Antonio
Bittencourt, Luiz	Kächele, Steffen	Philips, Eline
Bovenzi, Antonio	Liang, Zhenkai	Rouvoy, Romain
Cacho, Nelio	Lopes, Frederico	Schaefer, Jan
Christophe, Laurent	Lopes, Nuno	Scholliers, Christophe
Crain, Tyler	Maerien, Jef	Senna, Carlos
Dell'Amico, Matteo	Malheiros, Neumar	Spann, Christian
Domaschka, Jörg	Merle, Philippe	Textor, Andreas
Esteves, Sérgio	Meyer, Fabian	Tudor, Bogdan Marius
Frattini, Flavio	Morandat, Floréal	Vukolić, Marko
Haque, Tareq	Nakai, Alan	
Hoste, Lode	Natella, Roberto	

Table of Contents

Work-in-Progress Papers

Semantically Aware Contention Management for Distributed Applications

Matthew Brook, Craig Sharp, and Graham Morgan

School of Computing Science, Newcastle University
{m.j.brook1,craig.sharp,graham.morgan}@ncl.ac.uk

Abstract. Distributed applications that allow replicated state to deviate in favour of increasing throughput still need to ensure such state achieves consistency at some point. This is achieved via compensating conflicting updates or undoing some updates to resolve conflicts. When causal relationships exist across updates that must be maintained then conflicts may result in previous updates also needing to be undone or compensated for. Therefore, an ability to manage contention across the distributed domain to pre-emptively lower conflicts as a result of causal infringements without hindering the throughput achieved from weaker consistency is desirable. In this paper we present such a system. We exploit the causality inherent in the application domain to improve overall system performance. We demonstrate the effectiveness of our approach with simulated benchmarked performance results.

Keywords: replication, contention management, causal ordering.

1 Introduction

A popular technique to reduce shared data access latency across computer networks requires clients to replicate state locally; data access actions (reads and/or writes) become quicker as no network latency will be involved. An added benefit of such an approach is the ability to allow clients to continue processing when disconnected from a server. This is of importance in the domains of mobile networks and rich Internet clients where lack of connectivity may otherwise inhibit progress.

State shared at the client side still requires a level of consistency to ensure correct execution. This level is usually guaranteed by protocols implementing *eventual consistency*. In such protocols, reconciling conflicting actions that are a result of clients operating on out-of-date replicas must be achieved. In this paper we assume a strict case of conflict that includes out-of-date reads. Such scenarios are typical for rich Internet clients where eventual agreement regarding data provenance during runtime can be important.

Common approaches to reconciliation advocate compensation or undoing previous actions. Unfortunately, the impact of either of these reconciliation techniques has the potential to invalidate the causality at the application level within clients (semantic causality): all tentative actions not yet committed to shared state but carried out on the local replica may have assumed previous actions were successful, but now require reconciliation. This requires tentative actions to be rolled back. For applications requiring this level of client-local causality, the impact of rolling back tentative actions has a significant impact on performance; they must rollback execution to where the conflict was discovered.

Eventually consistent applications exhibiting strong semantic causality that need to rollback in the presence of conflicts are similar in nature to transactions. Transactions, although offering stronger guarantees, abort (rollback state changes) if they can't be committed. In recent years transactional approaches have been used for regulating multi-threaded accesses to shared data objects. A contention manager has been shown to improve performance in the presence of semantic causality across a multi-threaded execution. A contention manager determines which transactions should abort based on some defined strategy relating to the execution environment.

In this paper we present a contention management scheme for distributed applications where maintaining semantic causality is important. We extend our initial idea [10] by dynamically adapting to possible changes in semantic causality at the application layer. In addition, we extend our initial idea of a single server based approach to encompass n-tier architectures more typical of current server side architectures.

In section 2 we describe background and related work, highlighting the notion of borrowing techniques from transactional memory to benefit distributed applications. In section 3 we describe the design of our client/server interaction scenario. In section 4 we describe our contention management approach with enhanced configurability properties. In section 5 we present results from our simulation demonstrating the benefits our approach can bring to the system described in section 3.

2 Background and Related Work

Our target application is typically a rich Internet client that maintains replicas of shared states at the client side and wishes to maintain semantic causality. Such an application could relate to e-commerce, collaborative document editing or any application where the provenance of interaction must be accurately captured.

2.1 Optimistic Replication

Optimistic protocols allow for a deviation in replica state to promote overall system throughput. They are ideal for those applications that can tolerate inconsistent state in favour of instant information retrieval (e.g., search engines, messaging services). The guarantee afforded to the shared state is eventual consistency [3], [4].

Popular optimistic solutions such as Cassandra [5] and Dynamo [6] may be capable of recognising causal infringement, but do not provision rollback schemes to enforce semantic causality requirements at the application layer within their design. They are primarily designed, and work best, for scalable deployment over large clusters of servers. They are not designed for distributed clients to maintain replication of shared state. However, earlier academic work did consider client-based replication. Bayou [7] and Icecube [8] [9] do attempt to maintain a degree of causality, but only at the application programmer's discretion. In such systems the application programmer may specify the extent of causality, preventing a total rollback and restart. This has the advantage of exploiting application domains to improve availability and timeliness, but does complicate the programming of such systems as the boundary between protocol and application overlap. In addition, the programmer may not be able to predict the semantic causality accurately.

2.2 Transactions

Transactions offer a general platform to build techniques for maintaining causality within replication accesses at the client side that does not require tailoring based on application semantics. Unfortunately, they impose a high overhead to an application that negates the scalable performance expected from optimistic replication: maintain ordering guarantees for accesses at the server and clients complete with persistent fault-tolerance.

Transactional memory [12] approaches found in multi-threaded programming demonstrate fewer of such guarantees (e.g., persistence). In addition, unlike typical transactions in database processing multi-threaded programs present a high degree of semantic causality (threads executing and repeatedly sharing state). Therefore, it is no surprise to learn that they have been shown to help improve overall system throughput by judicial use of a contention manager [1] [2] [13]. Although no single contention manager works for all application types [14], dynamism can be used to vary the contention strategy.

2.3 Contribution

In our previous work we successfully borrowed the concept of contention management from transactional memory and described a contention manager that may satisfy our rich Internet client setting [10]. We derived our own approach based on probability of data object accesses by clients. Unfortunately, the approach had limitations: (1) it was static and could not react to changes in application behaviour (i.e., probability of object accesses changing); (2) it worked on a centralised server (i.e., we could not utilise scalability at the server side). In this paper we propose solutions to both these problems and present a complete description of our eventually consistent protocol (which we didn't present earlier) with the dynamic version of our semantically aware contention manager.

3 System Design

Our contention management protocol is deployed within a three-tier server side architecture as illustrated in Fig. 1. The load balancer initially determines which application server to direct client requests to. A sticky session is then established such that all messages from one client are always directed to the same application server.

Communication channels exhibit FIFO qualities but message loss is possible. We model this type of communication channel to reflect a typical TCP connection with the possibility of transient connectivity of clients, a requirement in many rich Internet client applications.

Data accesses are performed locally on a replication of the database state at the client side. In our evaluation we used a complete replica, but this could be partial based on a client's ability to gain replica state on initialisation of a client. Periodically clients inform the application server of these data accesses. An application server receives these access notifications and determines whether these accesses are valid given the current state as maintained at the database. Should the updates be valid, the database state is updated to reflect the changes the client has made locally. However, if the update results in an irreconcilable conflict then the client is notified. When the client learns that a previous action was not successful, the client rolls back to the point of execution where this action took place and resumes execution from this point.

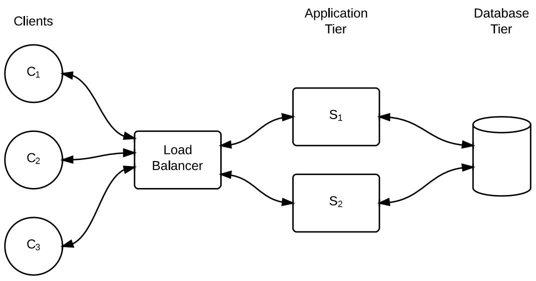


Fig. 1. System Design

3.1 Clients

Each client maintains a local replica of the data set maintained by the database. All client actions enacted on the shared data are directed to their local replica. The client uses a number of logical clocks to aid in managing their execution and rollback:

- *Client data item clock (CDI)* – exists for each data item and identifies the current version of the data item's state held by a client. The value is used to identify when a client's view of the data item is out-of-date. This value is incremented by the client when updating a data item locally or when a message is received from an application server informing the client of a conflict.
- *Client session clock (CSC)* – this value is attached to every request sent to an application server. When a client rolls back this value is incremented. This allows the application server to ignore messages belonging to out of date sessions.
- *Client action clock (CAC)* – this value is incremented each time a message is sent to an application server. This allows the application servers to recognize missing messages from clients.

The result of an action that modifies a data item in the local replicated state results in a message being sent to the application servers. This message contains the data item state, the CDI of the data item, the CSC and the CAC. An execution log is maintained and each client message is added to it. This execution log allows client rollback.

A message arriving from the application server indicates that a previous action, say A_n, was not possible or client messages are missing. All application server messages contain a session identifier. If this identifier is the same or lower than the client's CSC then the application server message is ignored (as the client has already rolled back – the application server may send out multiple copies of the rollback message). However, if the session identifier is higher than the client's CSC the client must update their own CSC to match the new value and rollback.

If the message from the application server was sent due to missing client messages then only an action clock and session identifier will be present (we call this the *missed message request*). On receiving this message type, the client should rollback to the action point using their execution log. However, if the application server sent the message because of a conflicting action then this message will contain the latest state of the data that A_n operated on and the new logical clock value (we call this the *irreconcilable message request*). On receiving such a message the client halts execution and rolls back to attempt execution from A_n.

Although a client will have to rollback when requested by the application server, the receiving of an application server message also informs the client that all their actions prior to A_n were successful. As such, the client can reduce the size of their execution log to reflect this.

3.2 Application Server

The role of an application server is to manage the causal relationship between a client's actions and ensure a client's local replica is eventually consistent. The application server manages three types of logical clock to inform the client when to rollback:

- *Session identifier (SI)* – this is the application server's view of a client's CSC. Therefore, the application server maintains an SI for each client. This is used to disregard messages from out of date sessions from clients. The SI is incremented by one each time a client is requested to rollback.
- *Action clock (AC)* – this is the application server's view of client's CAC. Therefore, the application server maintains an AC for each client. This is used to identify missing messages from a client. Every action honoured by the application server on behalf of the client results in the AC for that client being set to the CAC belonging to the client.
- *Logical clock (LC)* – this value is stored with the data item state at the database. The value is requested by the application sever when an update message is received from a client. The application server determines if a client has operated on an out-of-date version using this value. If the action from the client was valid then the application server updates the value at the database. Requests made to the database are considered transactional; handling transactional failure is beyond the

scope of this paper (we propose the use of the technique described in [11] to handle such issues).

A message from a client, say C_1, may not be able to be honoured by the application server due to one of the following reasons:

- *Stale session* – the application server's SI belonging to C_1 is less than the CSC in C_1's message.
- *Lost message* – the CAC in C_1's message is two or more greater than the application server's AC for C_1.
- *Stale data* – the LC for the data item the client has updated is greater than the CDI in C_1's message.

When the application server has to rebut a client's access, a rollback message is sent to that client. Preparation of the rollback message depends on the state of the client as perceived by the application server. An application server can recognize a client (C_1) in one of two modes:

- *Progress* – the last message received from C_1 could be honoured.
- *Stalled* – the last message received from C_1 could not be honoured or was ignored.

If C_1 is in the progress state then the application server will create a new rollback message and increment the SI for C_1 by one. If the problem was due to a lost message then the AC value for C_1 is incremented by one (to indicate that rollback is required to just after the last successful action) and is sent together with C_1's updated SI value (this is the *missed message request* mentioned in section 3.1). If the problem was due to an irreconcilable action the message sent to the client will contain the latest LC for the data item the action attempted to access (retrieved from the database), and the application server's SI value for C_1 (this is the *irreconcilable message request* mentioned in section 3.1). The application server moves C_1 to the stalled state and records the rollback message sent to C_1 (this is called the *authoritative rollback message*).

If C_1 is in the stalled state all the client's messages are responded to with C_1's current authoritative rollback message. The exception is if the received message contains a CSC value equal to C_1's SI value held by the application server. If such a message is received then the CAC value contained in the message is compared with the AC value of C_1 held by the application server. If it is greater (i.e., the required message from C_1 is missing) the application server increments C_1's SI by one and constructs a new authoritative rollback message to be used in response to C_1. If the CAC value in the message is equivalent to the AC value of C_1 as held by the application server, and the application server can honour this message (logical clock values are valid), then C_1's state is moved to progress and the authoritative rollback message is discarded. If the message cannot be honoured (it is irreconcilable), then the application server increments the SI for C_1 by one and uses this together with the contents of the received message to create a fresh authoritative rollback message, sending this to the client.

3.3 Database

The database manages the master copy of the shared data set. The data set comprises of data items and their associated logical clock values. The data item reflects the state while the logical clock indicates versioning information. The logical clock value is requested by application servers to be used in determining when a client's update message is irreconcilable. The database accepts requests to retrieve logical clock values for data items or to update the state and logical clock values (as a result of a successful action as determined by an application server). We assume application servers and databases interact in standard transactional ways.

3.4 System Properties

The system design described so far can be reasoned about in the following manner:

- *Liveness* – Clients progress until an application server informs them that they must rollback (via an authoritative rollback message). If this message is lost in transit the client will continue execution, sending further access notification to the application server. The application server will continue to send the rollback message in response until the client responds appropriately. If the client message that is a direct response to the authoritative rollback message goes missing the application server will eventually realize this due to receiving client messages with the appropriate SI values but with CAC values that are too high. This will cause the application server to respond with an authoritative rollback message.
- *Causality* – A client always rolls back to where an irreconcilable action (or missing action due to message loss) was discovered by the application server. Therefore, all actions that are reconciled at the application server and removed from a client's execution log maintain the required causality. Those tentative actions in the execution log are in a state of reconciliation and may require rolling back.
- *Eventually Consistent* – If a client never receives a message from an application server then either: (i) all client requests are honoured and states are mutually consistent; or (ii) all application server or client messages are lost. Therefore, as long as sufficient connectivity between client and application servers exists, the shared data will become eventually consistent.

The system design provides opportunity for clients to progress independently of the application server in the presence of no message loss and no irreconcilable issues on the shared data. In reality, there will be a number of irreconcilable actions and as such the burden of rolling back is much more substantial than other eventually consistent optimistic approaches. This does, however, provide the benefit of not requiring any application level dependencies in the protocol itself; the application developer does not need to specify any exception handling facility to satisfy rollback.

4 Semantic Contention Management

We now describe our contention management scheme and how it is applied to the system design presented in the previous section. The aim of the contention management scheme is to attain a greater performance in the form of fewer irreconcilable differences without hindering overall throughput.

Like all contention management schemes, we exploit a degree of predictability to achieve improved performance. We assume that causality across actions is reflected in the order in which a client accesses shared data items. The diagram in Fig. 2 illustrates this assumption.

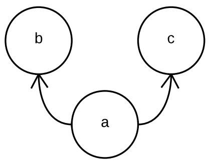


Fig. 2. Relating client actions progressing to data items

In the simple graph shown in Figure 2 we represent three data items (*a*, *b* and *c*) as vertices with two edges connecting *a* to *b* and *c*. The edges of the graph represent the causal relationship between the two connected data items. So if a client performs a successful action on data item *a* there is a higher than average probability that the focus of the next action from the same client will be either data item *b* or *c*.

Each application server manages their own graph configuration representing the data items stored within the database. Because of this graphs will diverge across application servers. This is of no concern, as an application server must reflect the in-session causality of its own clients, not the clients of others. We extend the system design described in the previous section by adding the following constructs to support the contention management framework:

- *Volatility value (VV)* – a value associated to each vertex of the graph indicating the relative popularity for the given data item. The volatility for a data item in the graph is incremented when a client's action is successful. The volatility for the data item that was the focus of the action is incremented by one and the neighbouring data items (those that are connected by outgoing arcs of the original data item) volatilities are incremented by one. Periodically, the application server will decrement these values to reflect the deterioration of the volatility for nodes that are no longer experiencing regular data access.
- *Delta queue (DQ)* – for those actions that could not be honoured by the application server due to irreconcilable conflicts (out-of-date logical clock values) a backoff period is generated as the sum of the volatility for the related data. These related data items include the original data item for which the conflict occurred along with

the data items with the highest volatilities up to three hops away in the graph. This client is now considered to be in a stalled state and is placed in the delta queue for the generated backoff period. The backoff period is measured in milliseconds given a value generated from the volatility values.

- *Enhanced authoritative rollback message* – when a backoff period expires for a client residing in the delta queue, an enhanced authoritative rollback message is sent to the client. This is an extension of the authoritative rollback message described in the system design that includes a partial state update for the client. This partial state update includes the latest state and logical clock values for the conflicting data item and the data items causally related to the original conflicting access. Based on the assumption of causality as reflected in the graph configuration, the aim here to pre-emptively update the client. As a result, future update messages will have a higher chance of being valid (this cannot be guaranteed due to simultaneous accesses made by other clients).

The approach we have taken is a server side enhancement. This decision was taken to alleviate clients from active participation in the contention management framework. The client needs only to be able to handle the enhanced authoritative rollback message that requires additional state updates to the client's local replica.

As each application server manages their graph structure representing the data items, should a single application server crash, client requests can be directed to another working application server will little loss. Clients that originally had sessions belonging to the crashed application server will require directing to a different application server and there will be some additional conflicts and overhead due to the lost session data.

4.1 Graph Reconfiguration

To satisfy the changing probabilities of causal data access over time our static graph approach requires only minor modifications.

We introduce two new values that an application server maintains for each client:

- *Happens Before Value (HBV)* – the vertex representing a data item a client last successfully accessed.
- *Happens After Value (HAV)* – the vertex representing a data item a client successfully accessed directly after HBV.

If there does not exist a link between HBV and HAV then one is created. Unfortunately, if we were to continue in this manner we may well end up with a fully connected graph, unnecessarily increasing the load in the overall system (e.g., increased sized enhanced authoritative rollback message). Therefore, to allow for the deletion of edges as well as the creation of edges we make use of an additional value to record the popularity of traversal associated to each edge in the graph:

- *Edge Popularity Value (EPV)* – The cumulative number of times, across all clients, a causal occurrence has occurred between a HBV and HAV.

If there already exists a link between HBV and HAV then the associated edge's EPV is incremented by one. This provides a scheme within which the most popular edges will maintain the highest values. However, this may not reflect the current popularity of the causal relations, therefore, the EPVs purpose is to prune the graph. Periodically, the graph is searched and EPVs below a certain threshold result in the removal of their edges. After pruning the graph all remaining edges are reset to the value 0.

Periodic pruning and resetting of EPVs provides our scheme with a basic reconfiguration process to more appropriately reflect current semantic causal popularity. We acknowledge that this process of reconfiguration will incur a performance cost relative to the number of data items (vertices) and edges present in the graph. The decision on the time between periodic reconfiguration will be based on a number of factors: (i) the relative performance cost of the reconfiguration; (ii) the number of client requests within the period. If the number of requests is low but reconfiguration too frequent then edges may be removed that are still popular. Therefore, we dynamically base our reconfiguration timings on changes in load.

An interesting observation of reconfiguration is it also presents a window of opportunity to alter the data items present. If this was a typical e-commerce site with items for sale then they may be introduced as graph reconfiguration occurs. This has two benefits: (i) introduction of items may well alter the causal relationships dramatically (e.g., timed flash sales) and so waiting for reconfiguration would not result in unnecessary overhead as graph values change significantly; (ii) one can apply some application level analysis on the effect new items have on existing data items.

5 Evaluation

Three approaches were evaluated to determine performance in terms of server side conflicts and throughput: (1) the basic protocol as described in the system design with no contention management; (2) the enhanced protocol with contention management but without graph reconfigurations; (3) the enhanced protocol with both contention management and graph reconfiguration. To create an appropriate simulation scenario we rely on a pattern of execution for rich Internet clients similar to that described in [16] (ecommerce end client sales).

5.1 Simulation Environment

We produced a discrete event simulation using the SimJava [15] framework. We modeled variable client numbers, a load balancer, three application servers and a database as processes.

Graph layouts are randomly created and client accesses are pre-generated. The initial graph layouts include vertices with and without edges. In the dynamic scenario such a vertex may at some point become connected, but not in the static graph.

In the dynamic graph periodic reconfiguration occurred every thirty seconds with a relaxed threshold of one. This period was determined over experimentation and was found to provide reasonable balance between accurate causality representation and overhead induced by reconfiguration. The relaxed threshold simply indicated edges that had shown any causal interest would be guaranteed a starting presence in the graph after reconfiguration.

We simulated message delay between client and application servers (load balancer) as a random variable with a normal distribution between 1 - 50 milliseconds. Each client performs 200 data accesses then leaves the system. Each experiment was run five times to generate the average figures presented. The arrival rate of client messages to the application server was set as ten messages per second for each client process. The simulation was modeled with a 2% message loss probability. Database read and writes were 3 and 6 microseconds respectively.

5.2 Evaluation 1 – Irreconcilable Client Updates (Conflicts)

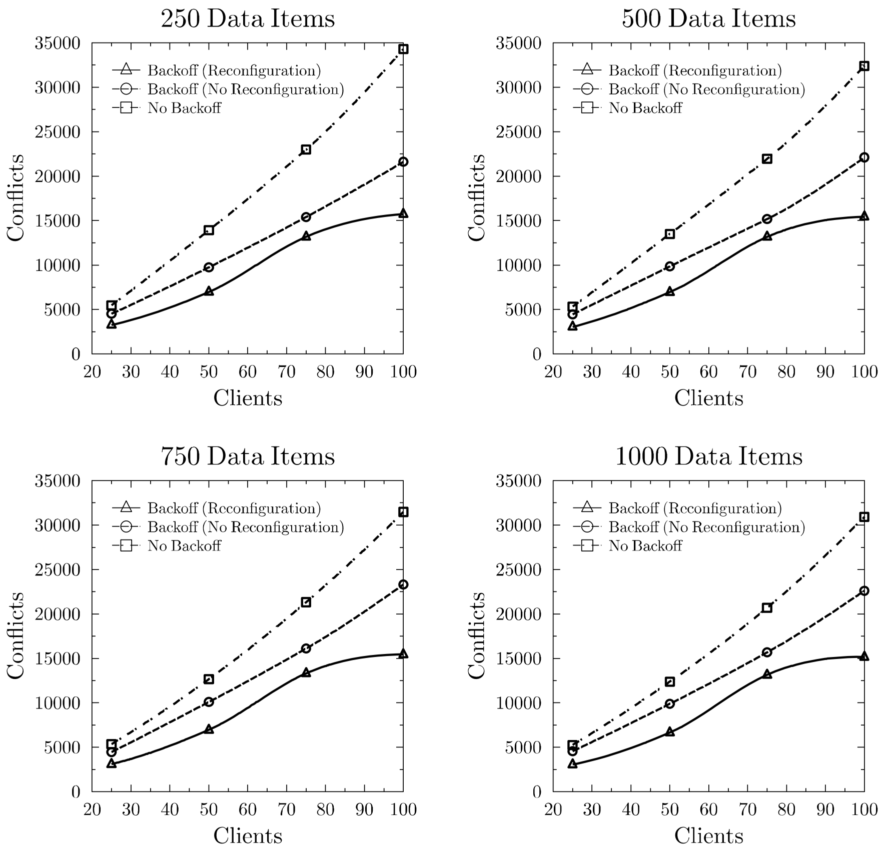


Fig. 3. Irreconcilable conflicts for varying graph sizes

The graphs in figure 3 show that the inclusion of contention management lowers conflicts. The results also show the added benefit of graph reconfiguration over a static graph. In addition, reconfiguration appears to approach a stable state as the contention increases. Reconfiguration allows for the system to adapt to the changing client interactions resulting in the graph more accurately reflecting semantic causality over time. Without reconfiguration the conflicts continue to rise rather than stabilize. What has little impact on the results is the number of data items represented in the graph. This is due to the predictability exhibited in the client accesses: if clients accessed data at random we would expect that graph size mattered, as there would naturally be less conflicts.

5.3 Evaluation 2 – Throughput of Successful Client Actions (Commits)

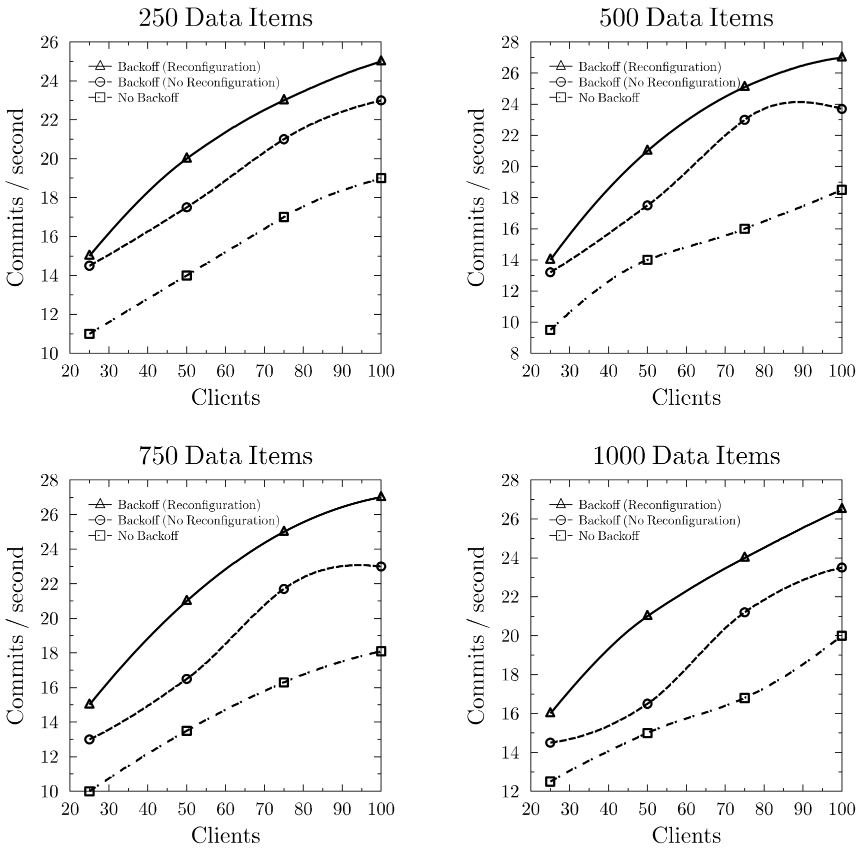


Fig. 4. Throughput measured as commits per second for varying graph sizes

Similar to the previous set of the results, the graph size plays little role given the predictive behaviour of the clients. The results show our backoff contention management providing the best throughput. Interestingly, without reconfiguration the system appears to reach saturation point early. With reconfiguration we still have improvement occurring, although it is tailoring off towards 100 clients.

The results presented here indicate that backing off clients and updating their replicas in a predictive manner actually improves performance: conflicts lowered and throughput is increased. In terms of throughput alone, this is seen as a significant 30% improvement when combined with reconfiguration. Therefore, we conclude that causality at the application layer can be exploited to improve performance for those applications where causality infringement requires rollback of local replica state.

6 Conclusion

We have described an optimistic replication scheme that makes use of dynamic contention management. We base our contention manager on the popularity of data accesses and the possible semantic causal relation this may hint at within the application layer. Our approach is wholly server based, requiring no responsibility for managing contention from the client side (apart from affording rollback). Our approach suits applications where causality is important and irreconcilable accesses of shared state may cause a client to rollback accesses tentatively carried out on a local replica. Such scenarios occur in rich Internet clients where provenance of data access is to be maintained or where actions of a client's progress must be rolled back in the context of achieving a successful client session. We describe our approach in the context of n-tier architectures, typical in application server scenarios.

Our evaluation, via simulation, demonstrates how overall throughput is improved by reducing irreconcilable actions on shared state. In particular, we show how adapting to changes in causal relationships during runtime based solely on access patterns of clients provide greatest improvements in throughput.

This is the first time runtime adaptability of causality informed contention management has been demonstrated in a complete solution exhibiting eventual synchronous guarantees. As such, we believe that this is not only a useful contribution to the literature, but opens new avenues of research by bringing the notion of contention management to replication protocols.

We acknowledge that our approach is focussed on a particular application type: applications that always rollback to where conflict was detected. However, we believe that advocating contention management as an aid to performance for eventually consistent replicated state in general would be beneficial and worthy of future exploration.

Future work will focus on peer-to-peer based evaluation and creating contention management schemes suitable for mobile environments (where epidemic models of communication are favoured). A further opportunity of exploration will be in taking the semantic awareness properties of this work back to transactional memory systems themselves.

References

1. Scherer III, W N., Scott M.L.: Contention Management in Dynamic Software Transactional Memory. In: PODC Workshop on Concurrency and Synchronization in Java Programs, pp. 70–79 (2004)
2. Scherer III, W.N., Scott, M.L.: Advanced Contention Management for Dynamic Software Transactional Memory. In: Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing, pp. 240–248. ACM, New York (2005)
3. Saito, Y., Shapiro, M.: Optimistic Replication. ACM Computing Surveys 37, 42–81 (2005)
4. Vogels, W.: Eventually Consistent. Communications of the ACM 52, 40–44 (2009)
5. Lakshman, A., Malik, P.: Cassandra: A Decentralized Structured Storage System. ACM SIGOPS Operating Systems Review 44, 35–40 (2010)
6. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon’s Highly Available Key-Value Store. In: 21st ACM SIGOPS Symposium on Operating Systems Principles, pp. 205–220. ACM, New York (2007)
7. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In: 15th ACM Symposium on Operating Systems Principles, pp. 172–182. ACM, New York (1995)
8. Kermarrec, A., Rowstron, A., Shapiro, M., Druschel, P.: The IceCube Approach to the Reconciliation of Divergent Replicas. In: 20th Annual ACM Symposium on Principles of Distributed Computing, pp. 210–218. ACM, New York (2001)
9. Preguiça, N., Shapiro, M., Matheson, C.: Semantics-Based Reconciliation for Collaborative and Mobile Environments. In: Meersman, R., Schmidt, D.C. (eds.) CoopIS 2003, DOA 2003, and ODBASE 2003. LNCS, vol. 2888, pp. 38–55. Springer, Heidelberg (2003)
10. Abushnagh, Y., Brook, M., Sharp, C., Ushaw, G., Morgan, G.: Liana: A Framework that Utilizes Causality to Schedule Contention Management across Networked Systems. In: Meersman, R., Panetto, H., Dillon, T., Rinderle-Ma, S., Dadam, P., Zhou, X., Pearson, S., Ferscha, A., Bergamaschi, S., Cruz, I.F. (eds.) OTM 2012, Part II. LNCS, vol. 7566, pp. 871–878. Springer, Heidelberg (2012)
11. Kistijantoro, A.I., Morgan, G., Shrivastava, S.K., Little, M.C.: Enhancing an Application Server to Support Available Components. IEEE Transactions on Software Engineering 34(4), 531–545 (2008)
12. Herlihy, M., Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-free Data Structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, vol. 21(2), pp. 289–300. ACM, New York (1993)
13. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software Transactional Memory for Dynamic-sized Data Structures. In: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing, pp. 92–101. ACM, New York (2003)
14. Guerraoui, R., Herlihy, M., Pochon, B.: Polymorphic Contention Management. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 303–323. Springer, Heidelberg (2005)
15. University of Edinburgh, SimJava, `http://www.dcs.ed.ac.uk/home/hase/simjava/` (accessed: February 16, 2013)
16. Clarke, D., Morgan, G.: E-Commerce with Rich Clients and Flexible Transactions. In: First International Workshop on Software Technologies for Future Dependable Distributed Systems, pp. 73–77. IEEE (2009)

FITCH: Supporting Adaptive Replicated Services in the Cloud

Vinicius V. Cogo[1], André Nogueira[1], João Sousa[1],
Marcelo Pasin[2], Hans P. Reiser[3], and Alysson Bessani[1]

[1] University of Lisbon, Faculty of Sciences, Portugal
[2] University of Neuchatel, Faculty of Science, Switzerland
[3] University of Passau, Institute of IT-Security and Security Law, Germany

Abstract. Despite the fact that cloud computing offers a high degree of dynamism on resource provisioning, there is a general lack of support for managing dynamic adaptations of replicated services in the cloud, and, even when such support exists, it is focused mainly on elasticity by means of horizontal scalability. We analyse the benefits a replicated service may obtain from dynamic adaptations in the cloud and the requirements on the replication system. For example, adaptation can be done to increase and decrease the capacity of a service, move service replicas closer to their clients, obtain diversity in the replication (for resilience), recover compromised replicas, or rejuvenate ageing replicas. We introduce FITCH, a novel infrastructure to support dynamic adaptation of replicated services in cloud environments. Two prototype services validate this architecture: a crash fault-tolerant Web service and a Byzantine fault-tolerant key-value store based on state machine replication.

1 Introduction

Dynamic resource provisioning is one of the most significant advantages of cloud computing. Elasticity is the property of adapting the amount and capacity of resources, which makes it possible to optimize performance and minimize costs under highly variable demands. While there is widespread support for dynamic resource provisioning in cloud management infrastructures, adequate support is generally missing for service replication, in particular for systems based on state machine replication [8,23].

Replication infrastructures typically use a static configuration of replicas. While this is adequate for replicas hosted on dedicated servers, the deployment of replicas on cloud providers creates novel opportunities. Replicated services can benefit from dynamic adaptation as replica instances can be added or removed dynamically, the size and capacity of the resources available to replicas can be changed, and replica instances may be migrated or replaced by different instances. These operations can lead to benefits such as increased performance, increased security, reduced costs, and improved legal conformity. Managing cloud resources for a replication group creates additional requirements for resource management and demands a coherent coordination of adaptations with the replication mechanism.

In this paper we present FITCH (*Fault- and Intrusion-Tolerant Cloud computing Hardpan*), a novel infrastructure to support dynamic adaptation of replicated services in cloud environments. FITCH aggregates several components found in cloud infrastructures and some new ones in a *hybrid architecture* [27] that supports *fundamental operations* required for adapting replicated services considering dependability, performance and cost requirements. A key characteristic of this architecture is that it can be easily deployed in current data centres [3] and cloud platforms.

We validate FITCH with two representative replicated services: a crash-tolerant web service providing static content and a Byzantine fault-tolerant (BFT) key-value store based on state machine replication. When deployed in our FITCH infrastructure, we were able to easily extend both services for improved dependability through proactive recovery [8,24] and rejuvenation [18] with minimal overhead. Moreover, we also show that both services can reconfigure and adapt to varying workloads, through horizontal (adding/removing replicas) and vertical (upgrading/downgrading replicas) scalability.

FITCH fills a gap between works that propose either (a) protocols for reconfiguring replicated services [20,21] or (b) techniques for deciding when and how much to adapt a service based on its observed workload and predefined SLA [6,14,17]. Our work defines a system architecture that can receive adaptation commands provided by (b) and leverages cloud computing flexibility in providing resources by orchestrating the required reconfiguration actions. FITCH provides a basic interface for adding, removing and replacing replicas, coordinating all low level actions mentioned providing end-to-end service adaptability. The main novel contributions of this paper are:

- A systematic analysis of the motivations and technical solutions for dynamic adaptation of replicated services deployed the cloud (§2);
- The FITCH architecture, which provides generic support for dynamic adaptation of replicated services running in cloud environments (§3 and §4);
- An experimental demonstration that efficient dynamic adaptation of replicated services can be easily achieved with FITCH for two representative services and the implementation of proactive recovery, and horizontal and vertical scalability (§5).

2 Adapting Cloud Services

We review several technical solutions regarding dynamic service adaptation and correlate them with motivations to adapt found in production systems, which we want to satisfy with FITCH.

Horizontal scalability is the ability of **increasing or reducing the number of computing instances** responsible for providing a service. An increase – scale-out – is an action to deal with peaks of client requests and to increase the number of faults the system can tolerate. A decrease – scale-in – can save resources and money. Vertical scalability is achieved through upgrade and downgrade procedures that respectively **increase and reduce the size or capacity of resources allocated** to service instances (e.g., Amazon EC2 offers predefined categories for VMs – small, medium, large, and extra large – that differ in CPU and memory resources [2]). Upgrades – scale-up – can improve service capacity while maintaining the number of replicas. Downgrades – scale-down – can release over-provisioned allocated resources and save money.

Moving replicas to different cloud providers can result in performance improvements due to different resource configurations, or financial gains due to different prices and policies on billing services, and is beneficial to prevent vendor lock-in [4]. **Moving service instances close to clients** can bring relevant performance benefits. More specifically, logical proximity to the clients can reduce service access latency. **Moving replicas logically away from attackers** can increase the network latency experienced by the attacker, reducing the impact of its attacks on the number of requests processed (this can be especially efficient for denial-of-service attacks).

Replacing faulty replicas (crashed, buggy or compromised) is a reactive process following fault detections, which replaces faulty instances by new, correct ones [25]. It decreases the costs for the service owner, since he still has to pay for faulty replicas, removing also a potential performance degradation caused by them, and restores the service fault tolerance. **Software replacement** is an operation where software such as operating systems and web servers are replaced in all service instances at run-time. Different implementations might differ on performance aspects, licensing costs or security mechanisms. **Software update** is the process of replacing software in all replicas by up-to-date versions. Vulnerable software, for instance, must be replaced as soon as patches are available. New software versions may also increase performance by introducing optimized algorithms. In systems running for long periods, long running effects can cause performance degradation. Software rejuvenation can be employed to **avoid such ageing problems** [18].

3 The FITCH Architecture

In this section, we present the architecture of the FITCH infrastructure for replicated services adaptation.

3.1 System and Threat Models

Our system model considers a usual cloud-based Infrastructure-as-a-Service (IaaS) with a large pool of physical machines hosting user-created virtual machines (VMs) and some trusted infrastructure for controlling the cloud resources [3]. We assume a *hybrid distributed system* model [27], in which different components of the system follow different fault and synchrony models. Each machine that hosts VMs may fail arbitrarily, but it is equipped with a trusted subsystem that can fail only by crashing (i.e., cannot be intruded or corrupted). Some machines used to control the infrastructure are trusted and can only fail by crashing. In addition, the network interconnecting the system is split into two isolated networks, here referred to as *data plane* and *control plane*.

User VMs employ the *data plane* to communicate internally and externally with the internet. All components connected to this network are untrusted, i.e., can be subject to Byzantine faults [8] (except the service gateway, see §3.3). We assume this network and the user VMs follow a *partially synchronous* system model [16]. The *control plane* connects all trusted components in the system. Moreover, we assume that this network and the trusted components follow a *synchronous* system model with bounded computations and communications. Notice that although clouds do not employ real-time software and hardware, in practice the over provision of the control network associated with the use of dedicated machines, together with over provisioned VMs running with high priorities, makes the control plane a “de facto” synchronous system (assuming sufficiently large time bounds) [22,25].

3.2 Service Model

FITCH supports replicated services running on the untrusted domain (as user VMs) that may follow different replication strategies. In this paper, we focus on two extremes of a large spectrum of replication solutions.

The first extreme is represented by *stateless services* in which the replicas rely on a shared storage component (e.g., a database) to store their state. Notice that these services do have a state, but they do not need to maintain it coherently. Service replicas can process requests without contacting other replicas. A server can fetch the state from the shared storage after recovering from a failure, and resume processing. Classical examples of stateless replicated services are web server clusters in which requests are distributed following a load balancing policy, and the content served is fetched from a shared distributed file system.

The other extreme is represented by consistent *stateful services* in which the replicas coordinate request execution following the *state machine replication* model [8,23]. In this model, an arbitrary number of client processes issue commands to a set of replica processes. These replicas implement a stateful service that changes its state after processing client commands, and sends replies to the issuing clients. All replicas have to execute the same sequence of commands, which requires the use of an agreement protocol to establish a total order before request execution. The Paxos-based coordination and storage systems used by Google [9] are examples of services that follow this model.

One can fit most popular replication models between these two extreme choices, such as the ones providing eventual consistency, used, for example, in Amazon’s Dynamo [13]. Moreover, the strategies we consider can be used together to create a dependable multi-tier architecture. Its clients connect to a stateless tier (e.g., web servers) that executes operations and, when persistent state access is required, they access a stateful tier (e.g., database or file system) to read or modify the state.

3.3 Architecture

The FITCH architecture, as shown in Fig. 1, comprises two subsystems, one for controlling the infrastructure and one for client service provisioning. All components are connected either with the control plane or the data plane. In the following we describe the main architectural components deployed on these domains.

The **trusted domain** contains the components used to control the infrastructure. The core of our architecture is the *adaptation manager*, a component responsible to perform the requested dynamic adaptations in the cloud-hosted replicated services. This component is capable of inserting, removing and replacing replicas of a service running over FITCH. It provides public interfaces to be used by the *adaptation heuristic*. Such heuristic defines when and plans how adaptations must be performed, and is determined by human administrators, reactive decision engines, security information event managers and other systems that may demand some dynamic adaptation on services.

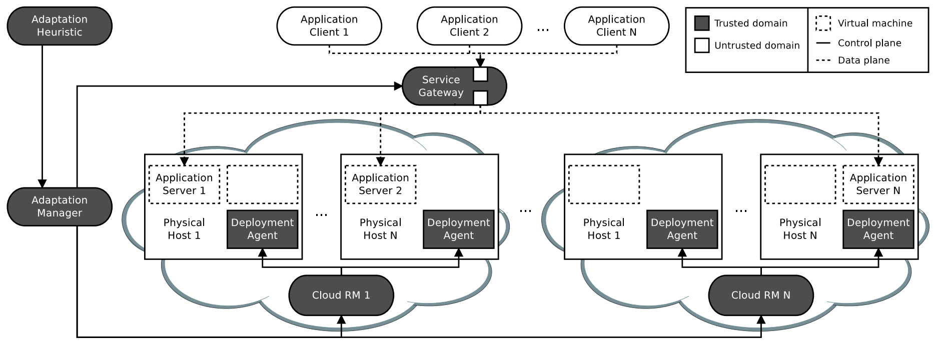


Fig. 1. The FITCH architecture

The *service gateway* maintains the group membership of each service and supports pluggable functionalities such as proxy and load balancing. It works like a lookup service, where new clients and instances request the most up-to-date group of replicas of a given service through the data plane. The adaptation manager informs the service gateway about each modification in the service membership through the control plane. The service gateway thus is a special component connected to both networks. This is a reasonable assumption as all state updates happen via the trusted control plane, whereas clients have read-only access. Moreover, having a trusted point of contact for fetching membership data is a common assumption in dynamic distributed systems [20].

Cloud resource managers (RM) provide resource allocation capabilities based on requests from the adaptation manager in order to deploy or destroy service replicas. The adaptation manager provides information about the amount of resources to be allocated for a specific service instance and the VM image that contains all required software for the new replica. The cloud RM chooses the best combination of resources for that instance based on requested properties. This component belongs to the trusted domain, and the control plane carries out all communication involving the cloud RM.

The *deployment agent* is a small trusted component, located inside cloud physical hosts, which is responsible to guarantee the deployment of service instances. It belongs to the trusted domain and receives deployment requests from cloud RM through the control plane. The existence of this component follows the paradigm of hybrid nodes [27], previously used in several other works (e.g., [15,22,24,25]).

The **untrusted domain** contains the components that provide the adaptive replicated service. *Application servers* are virtual machines used to provide the replicated services deployed on the cloud. Each of these user-created VMs contains all software needed by its service replica. Servers receive, process and answer client requests directly or through the service gateway depending on the configuration employed.

Cloud physical hosts are physical machines that support a virtualization environment for server consolidation. These components host VMs containing the replicas of services running over FITCH. They contain a hypervisor that controls the local resources and provides strong isolation between hosted VMs. A cloud RM orchestrates such environment through the control plane.

Application clients are processes that perform requests to service instances in order to interact with the replicated service. They connect to the service gateway component to discover the list of available replicas for a given service (and later access them directly) or send requests directly to the service gateway, which will forward them to the respective service instances. The application clients can be located anywhere in the internet. They belong to the untrusted domain and communicate with the application servers and gateway through the data plane.

3.4 Service Adaptation

The FITCH architecture described in the previous section supports adaptation on deployed replicated services as long as the adaptation manager, the gateway and some cloud resource managers are available. This means that during unavailability of these components (due to an unexpected crash, for instance), the replicated service can still be available, but adaptation operations are not. Since these services are deployed on a trusted and synchronous subsystem, it is relatively simple to implement fault tolerance for them, even transparently using VM-based technology [12].

The FITCH infrastructure supports three basic adaptation operations for a replicated service: *add*, *remove* and *replace* a replica. All adaptation solutions defined in §2 can be implemented by using these three operations. When the request to adapt some service arrives at the adaptation manager, it triggers the following sequence of operations (we assume a single replica is changed, for simplicity):

1. If adding or replacing:
   1.1. The adaptation manager contacts the cloud RM responsible for the physical host that matches the requested criteria for the new replica. The cloud RM informs the deployment agent on the chosen physical host asking it to create a new VM with a given image (containing the new replica software). When the VM is launched, and the replica process is started, the deployment agent informs the cloud RM, which informs the adaptation manager that a new replica is ready to be added to the service.
   1.2. The adaptation manager informs the gateway that it needs to *reconfigure* the replicated service to add the newly created replica for the service. The gateway invokes a reconfiguration command on the service to add the new replica.[1] When the reconfiguration completes, the gateway updates the current group membership information of the service.
2. If removing or replacing:
   2.1. The adaptation manager informs the gateway that it needs to *reconfigure* the replicated service to remove a service replica. The gateway invokes a reconfiguration command on the service to remove the old replica.[1] When the reconfiguration completes, the gateway updates the group membership of the service.
   2.2. The adaptation manager contacts the cloud RM responsible for the replica being removed or replaced. The cloud RM asks the deployment agent of the physical host in which the replica is running to destroy the corresponding VM. At the end of this operation, the cloud RM is informed and then it passes this information to the adaptation manager.

[1] The specific command depends on the replication technique and middleware being used by the service. We assume that the replication middleware implements a mechanism that ensures a consistent reconfiguration (e.g., [20,21]).

Notice that, for a replica replacement, the membership needs to be updated twice, first adding the new replica and then removing the old one. We intentionally use this two-step approach for all replacements, because it simplifies the architecture and is necessary to guarantee services' liveness and fault tolerance.

4 Implementation

We implemented the FITCH architecture and two representative services to validate it. The services are a crash fault-tolerant (CFT) web service and a consistent BFT key-value store.

FITCH. Cloud resource managers are the components responsible for deploying and destroying service VMs, following requests from the adaptation manager. Our prototype uses OpenNebula, an open source system for managing virtual storage, network and processing resources in cloud environments. We decided to use Xen as a virtualization environment that controls the physical resources of the service replicas. The deployment agent is implemented as a set of scripts that runs on a separated privileged VM, which has no interface with the untrusted domain.

A service gateway maintains information about the service group. In the stateless service, the service gateway is a load balancer based on Linux Virtual Server [29], which redirects client requests to application servers. In our stateful service implementation, an Apache Tomcat web server provides a basic service lookup.

Our adaptation manager is a Java application that processes adaptation requests and communicates with cloud RMs and the service gateway to address those requests. The communication between the adaptation manager and cloud resource managers is done through OpenNebula API for Java. Additionally, the communication between the adaptation manager and the service gateway is done through secure sockets (SSL).

In our implementation, each application server is a virtual machine running Linux. All software needed to run the service is present in the VM image deployed at each service instance. Different VLANs, in a Gigabit Ethernet switch, isolate data and control planes.

Stateless service. In the replicated stateless web service, each client request is processed independently, unrelated to any other requests previously sent to any replica. It is composed of some number of replicas, which have exactly the same service implementation, and are orchestrated by a load balancer in the service gateway, which forwards clients requests to be processed by one of the replicas.

Stateful service. In the key-value store based on BFT state machine replication [8], each request is processed in parallel by all service replicas and a correct answer is obtained by voting on replicas replies. To obtain such replication, we developed our key-value store over a Byzantine state machine replication library called BFT-SMaRt [5]. For the purpose of this paper, it is enough to know that BFT-SMaRt employs a leader-based total order protocol similar to PBFT [8] and that it implements a reconfiguration protocol following the ideas presented by Lamport *et al.* [20].

Adaptations. We implemented three adaptation solutions using the FITCH infrastructure. Both services employ *proactive recovery* [8,24] by periodically replacing a replica by a new and correct instance. This approach allows a fault-tolerant system to tolerate an arbitrary number of faults in the entire service life span. The window of vulnerability in which faults in more than f replicas can disrupt a service is reduced to the time it takes all hosts to finish a recovery round. We use *horizontal scalability* (adding and removing replicas) for the stateless service, and we analyse *vertical scalability* (upgrading and downgrading the replicas) of the stateful service.

5 Experimental Evaluation

We evaluate our implementation in order to quantify the benefits and the impact caused by employing dynamic adaptation in replicated services running in cloud environments. We first present the experimental environment and tools, followed by experiments to measure the impact of proactive recovery, and horizontal and vertical scalability.

5.1 Experimental Environment and Tools

Our experimental environment uses 17 physical machines that host all FITCH components (see Table 1). This cloud environment provides three types of virtual machines – *small* (1 CPU, 2GB RAM), *medium* (2 CPU, 4GB RAM) or *large* (4 CPU, 8GB RAM). Our experiments use two benchmarks. The stateless service is evaluated using the WS-Test [26] web services microbenchmark. We executed the echoList application within this benchmark, which sends and receives linked lists of twenty 1KB elements. The stateful service is evaluated using YCSB [11], a benchmark for cloud-serving data stores. We implemented a wrapper to translate YCSB calls to requests in our BFT key-value store and used three workloads: a read-heavy workload (95% of GET and 5% of PUT [11]), a pure-read (100% GET) and a pure-write workload (100% PUT). We used OpenNebula version 2.0.1, Xen 3.2-1, Apache Tomcat 6.0.35 and VM images with Linux Ubuntu Intrepid and kernel version 2.6.27-7-server for x86_64 architectures.

5.2 Proactive Recovery

Our first experiment consists in replacing the entire set of service replicas as if implementing software rejuvenation or proactive/reactive recovery [8,18,22,25]. The former is important to avoid software ageing problems, whereas the latter enforces service's fault tolerance properties (for instance, the ability to tolerate f faults) [24].

Table 1. Hardware environment

Component	Qty.	Description	Component	Qty.	Description
Adaptation Manager	1	Dell PowerEdge 850 Intel Pentium 4 CPU 2.80GHz 1 single-core, HT 2.8 GHz / 1 MB L2 2 GB RAM / DIMM 533MHz 2 x Gigabit Eth. Hard disk 80 GB / SCSI	Client (YCSB)	1	Dell PowerEdge R410 Intel Xeon E5520 2 quad-core, HT 2.27 GHz / 1 MB L2 / 8 MB L3 32 GB / DIMM 1066 MHz 2 x Gigabit Eth. Hard disk 146 GB / SCSI
Client (WS-Test)	5		Service Gateway	1	
Cloud RM	3		Physical Cloud Host	6	

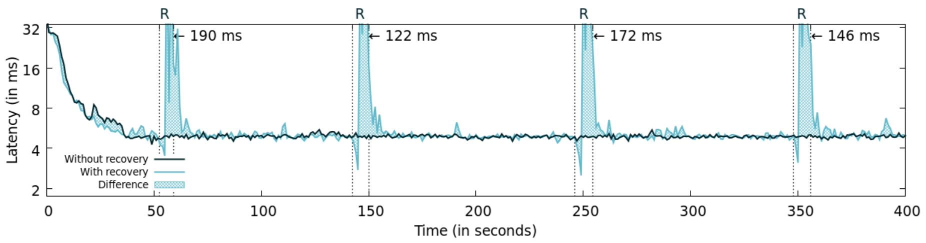


Fig. 2. Impact on a CFT stateless service. Note that the y-axis is in logarithmic scale.

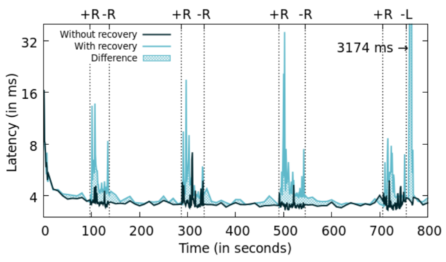
The idea of this experiment is to recover all n replicas from the service group, one-by-one, as early as possible, without disrupting the service availability (i.e., maintaining $n - f$ active replicas). Each recovery consists of creating and adding a new replica to the group and removing an old replica from it. We perform n recoveries per experiment, where n is the number of service replicas, which depends on the type and number of faults to be tolerated, and on the protocol used to replace all replicas. The time needed to completely recover a replica can also vary for each reconfiguration protocol.

Impact on a CFT stateless web service. Our first application in this experiment is a stateless web service that is initially composed of 4 large (L) replicas (which can tolerate 3 crash faults). The resulting latencies (in ms) are presented in Fig. 2, with and without the recovering operations in an experiment that took 400 s to finish. In this graph, each replica recovery is marked with a "R" symbol and has two lines representing the beginning and the end of replacements, respectively. The average of service latency without recovery was 5.60 ms, whereas with recovery was 8.96 ms. This means that the overall difference in the execution with and without recoveries is equivalent to 60% (represented in the filled area of Fig. 2). However, such difference is mostly caused during replacements, which only happens during 7.6% of the execution time.

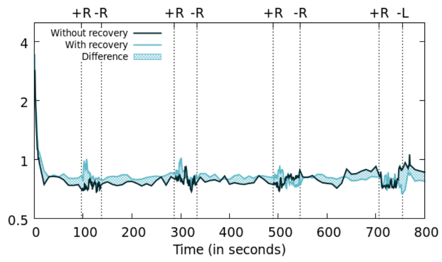
We draw attention to three aspects of the graph. First, the service has an initial warm-up phase that is independent of the recovery mechanism, and the inserted replica will also endure such phase. This warm-up phase occurs during the first 30 s of the experiment as presented in Fig. 2. Second, only a small interval is needed between insertions and removals, since the service reconfigures quickly. Third, the service latency increases 20- to 30-fold during recoveries, but throughput (operations/s) never falls to zero.

Impact on a BFT stateful key-value store. Our second test considers a BFT key-value store based on state machine replication. The service group is also composed of 4 large replicas, but it tolerates only 1 arbitrary fault, respecting the $3f + 1$ minimum required by BFT-SMaRt. Fig. 3 shows the resulting latencies with and without recovery, regarding (a) PUT and (b) GET operations. The entire experiment took 800 s to finish.

We are able to show the impact of a recovery round on service latency by keeping the rate of requests constant at 1000 operations/s. In this graph, each replica recovery is divided into the insertion of a new replica (marked with "+R") and the removal of an old replica (marked with "-R"). Removing the group leader is marked with "-L". The average latency of PUT operations without recovery was 3.69 ms, whereas with recovery it was 4.15 ms (a difference of 12.52%). Regarding GET operations,



(a) PUT operations



(b) GET operations

Fig. 3. Impact on a BFT stateful key-value store. Note that the y-axis is in logarithmic scale.

the average latency without recovery was 0.79 ms, whereas with recovery was 0.82 ms (a difference of 3.33%).

We draw attention to six aspects of these results. First, the service in question also goes through a warm-up phase during the first 45 s of the experiment. Second, the service needs a bigger interval between insertion and removal than the previous case because a state transfer occurs when inserting a new replica. Third, the service loses almost one third of its capacity on each insertion, which takes more time than in the stateless case. Fourth, the service stops processing during a few seconds (starting at 760 s in Fig. 3(a)) when the leader leaves the group. This unavailability while electing a new leader cannot be avoided, since the system is unable to order requests during leader changes. Fifth, client requests sent during this period are queued and answered as soon as the new leader is elected. Finally, GET operations do not suffer the same impact on recovering replicas as PUT operations do because GET operations are executed without being totally ordered across replicas, whereas PUT operations are ordered by BFT-SMaRt's protocol.

5.3 Scale-Out and Scale-In

Horizontal scalability is the ability of increasing or reducing the number of service instances to follow demands of clients. In this experiment, we insert and remove replicas from a stateless service group to adapt the service capacity. The resulting latencies are presented in Fig. 4. The entire experiment took 1800 s to finish, and the stateless service processed almost 4 million client requests, resulting in an average of 2220 operations/s. Each adaptation is either composed of a replica insertion ("+R") or removal ("-R").

Since all replicas are identical, we consider that each replica insertion/removal in the group can theoretically improve/reduce the service throughput by $1/n$, where n is the number of replicas running the service before the adaptation request.

The service group was initially composed of 2 small (S) replicas, which means a capacity of processing 1500 operations/s. Near the 100 s mark, the first adaptation was performed, a replica insertion, which decreased the service latency from 30 ms to 18 ms. Other replica insertions were performed near the 400 s and 700 s marks, increasing the service group to 4, and to 5 replicas, and decreasing the service latency to 13 ms, and to 10 ms, respectively. The service achieved its peak performance with 5 replicas near the

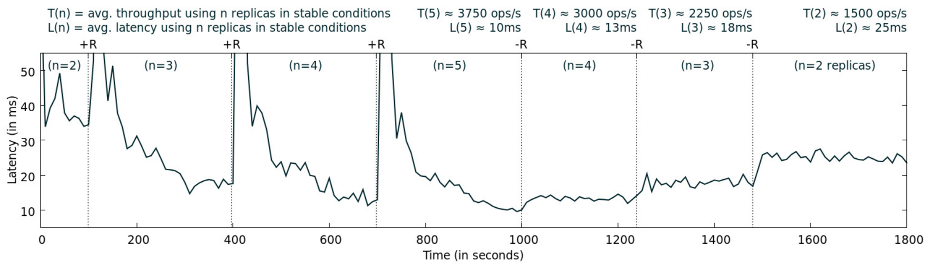


Fig. 4. Horizontal scalability test

900 s mark, and started decreasing the number of replicas with removals near the 1000 s, 1240 s and 1480 s, until when increases the service latency to 25 ms using 2 replicas.

Dynamically adapting the number of replicas can be performed reactively. A comparison between the service capacity and the current rate of client requests can determine if the service needs more or fewer replicas. In case of large differences, the system could insert or remove multiple replicas in the same adaptation request. Such rapid elasticity can adapt better to large peaks and troughs of client requests. We maintained the client request rate above the service capacity to obtain the highest number of processed operations on each second.

The entire experiment would cost \$0.200 on Amazon EC2 [2] considering a static approach (using 5 small replicas), while with the dynamic approach it would cost \$0.136. Thus, if this workload is repeated continuously, the dynamic approach could provide a monetary gain of 53%, which is equivalent to \$1120 per year.

5.4 Scale-Up and Scale-Down

Vertical scalability is achieved through upgrade and downgrade procedures to adjust the service capacity to client's demands. It avoids the disadvantages of increasing the number of replicas [1], since it maintains the number of replicas after a service adaptation.

In this experiment, we scale-up and -down the replicas of our BFT key-value store, during 8000 s. Each upgrade or downgrade operation is composed of 4 replica replacements in a chain. The service group comprises 4 initially small replicas (4S mark). Fig. 5 shows the resulting latencies during the entire experiment, where each "Upgrading" and "Downgrading" mark indicates a scale-up and scale-down, respectively. We also present on top of this figure the "(4S)", "(4M)" and "(4L)" marks, indicating the quantity and type of VMs used on the entire service group between the previous and the next marks, as well as the average latency and number of operations per second that each configuration is able to process.

The first adaptation was an upgrade from small (S) to medium (M) VMs, which reduced PUT latency from near 6 ms to 4 ms and GET latency from almost 0.9 ms to 0.82 ms. The second round was an upgrade to large (L) VMs. This reduced the PUT latency from 4 ms to 3.5 ms and the GET latency from 0.82 ms to 0.75 ms. Later, we performed downgrades to the service (from large to medium and from medium to small), which reduced the performance and increased the PUT latency to almost 5 ms and the GET latency to 0.87 ms.

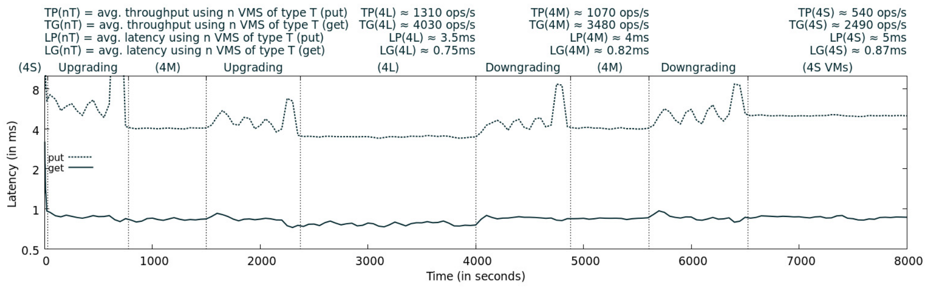


Fig. 5. Vertical scalability test. Note that y-axis is in logarithmic scale.

The entire experiment would cost \$2.84 on Amazon EC2 [2] considering the static approach (using 4 large replicas), while the dynamic approach would cost \$1.68. This can be translated into an economical gain of 32%, the equivalent to \$4559 per year.

6 Related Work

Dynamic resource provisioning is a core functionality in cloud environments. Previous work [7] has studied the basic mechanisms to provide resources given a set of SLAs that define the user's requirements. In our work, cloud managers provide resources based on requests sent by the adaptation manager, for allocating and releasing resources.

The adaptation manager is responsible for performing dynamic adaptations in arbitrary service components. These adaptations are defined by an adaptation heuristic. There are several proposals for this kind of component [6,14,17], which normally follow the "monitor-analyse-plan-execute" adaptation loop [19]. Their focus is mostly on preparing decision heuristics, not on executing the dynamic adaptations. Such heuristics are based mainly on performance aspects, whereas our work is concerned with performance and dependability aspects. One of our main goals is to maintain the service trustworthiness level during the entire mission time, even in the presence of faults. As none of the aforementioned papers was concerned about economy aspects, none of them releases or migrate over-provisioned resources to save money [28].

Regarding the results presented in these works, only Rainbow [17] demonstrates the throughput of a stateless web service running over their adaptation system. However, it does not discuss the impact caused by adaptations in the service provisioning. In our evaluation, we demonstrate the impact of replacing an entire group of service replicas, in a stateless web service, and additionally, in a stateful BFT key-value store.

Regarding the execution step of dynamic adaptation, only Dynaco [6] describes the resource allocation process and executes it using grid resource managers. FITCH is prepared to allow the execution of dynamic adaptations using multiple cloud resource managers. As adaptation heuristics is not the focus of this paper, we discussed some reactive opportunities, but implemented only time-based proactive heuristics. In the same way, proactive recovery is essential in order to maintain availability of replicated systems in the presence of faults [8,15,22,24,25]. An important difference to our work is that these systems do not consider the opportunities for dynamic management and elasticity as given in cloud environments.

Dynamic reconfiguration has been considered in previous work on group communication systems [10] and reconfigurable replicated state machines [20,21]. These approaches are orthogonal to our contribution, which is a system architecture that allows taking advantage of a dynamic cloud infrastructure for such reconfigurations.

7 Conclusions

Replicated services do not take advantage from the dynamism of cloud resource provisioning to adapt to real-world changing conditions. However, there are opportunities to improve these services in terms of performance, dependability and cost-efficiency if such cloud capabilities are used.

In this paper, we presented and evaluated FITCH, a novel infrastructure to support the dynamic adaptation of replicated services in cloud environments. Our architecture is based on well-understood architectural principles [27] and can be implemented in current data centre architectures [3] and cloud platforms with minimal changes. The three basic adaptation operations supported by FITCH – add, remove and replace a replica – were enough to perform all adaptation of interest.

We validated FITCH by implementing two representative services: a crash fault-tolerant web service and a BFT key-value store. We show that it is possible to augment the dependability of such services through proactive recovery with minimal impact on their performance. Moreover, the use of FITCH allows both services to adapt to different workloads through scale-up/down and scale-out/in techniques.

Acknowledgements. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT), through the CloudFIT project (PTDC/EIA-CCO/108299/2008) and the Multiannual Program (LaSIGE), and by EU's 7th Framework Program (FP7), through the TCLOUDS project (ICT-257243).

References

1. Abd-El-Malek, M., et al.: Fault-scalable Byzantine fault-tolerant services. In: Proc. of SOSP (2005)
2. Amazon Web Services: Amazon Elastic Compute Cloud (Amazon EC2) (2006), http://aws.amazon.com/ec2/
3. Barroso, L., Hölzle, U.: The datacenter as a computer: An introduction to the design of warehouse-scale machines. Synthesis Lectures on Computer Architecture 4(1) (2009)
4. Bessani, A., Correia, M., Quaresma, B., André, F., Sousa, P.: Depsky: dependable and secure storage in a cloud-of-clouds. In: Proc. of EuroSys (2011)
5. Bessani, A., et al.: BFT-SMaRt webpage, http://code.google.com/p/bft-smart
6. Buisson, J., Andre, F., Pazat, J.L.: Supporting adaptable applications in grid resource management systems. In: Proc. of the IEEE/ACM Int. Conf. on Grid Computing (2007)
7. Buyya, R., Garg, S., Calheiros, R.: SLA-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions. In: Proc. of Cloud and Service Computing (2011)

8. Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. 20(4) (2002)
9. Chandra, T., Griesemer, R., Redstone, J.: Paxos made live - an engineering perspective. In: Proc. of the PODC (2007)
10. Chen, W., Hiltunen, M., Schlichting, R.: Constructing adaptive software in distributed systems. In: Proc. of ICDCS (2001)
11. Cooper, B., et al.: Benchmarking cloud serving systems with YCSB. In: Proc. of SOCC (2010)
12. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus: high availability via asynchronous virtual machine replication. In: Proc. of the NSDI 2008 (2008)
13. DeCandia, G., et al.: Dynamo: Amazon's highly available key-value store. In: Proc. of SOSP (2007)
14. Dejun, J., Pierre, G., Chi, C.H.: Autonomous resource provisioning for multi-service web applications. In: Proc. of the WWW (2010)
15. Distler, T., et al.: SPARE: Replicas on Hold. In: Proc. of NDSS (2011)
16. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM 35 (1988)
17. Garlan, D., et al.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10) (2004)
18. Huang, Y., Kintala, C., Kolettis, N., Fulton, N.: Software rejuvenation: analysis, module and applications. In: Proc. of FTCS (1995)
19. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1) (2003)
20. Lamport, L., Malkhi, D., Zhou, L.: Reconfiguring a state machine. SIGACT News 41(1) (2010)
21. Lorch, J., et al.: The SMART way to migrate replicated stateful services. In: Proc. of EuroSys (2006)
22. Reiser, H., Kapitza, R.: Hypervisor-based efficient proactive recovery. In: Proc. of SRDS (2007)
23. Schneider, F.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22(4) (1990)
24. Sousa, P., Neves, N., Verissimo, P.: How resilient are distributed *f* fault/intrusion-tolerant systems? In: Proc. of DSN (2005)
25. Sousa, P., et al.: Highly available intrusion-tolerant services with proactive-reactive recovery. IEEE Trans. on Parallel and Distributed Systems (2010)
26. Sun Microsystems: Web services performance: Comparing Java™ 2 enterprise edition (J2EE™ platform) and .NET framework. Tech. rep., Sun Microsystems, Inc. (2004)
27. Veríssimo, P.: Travelling throught wormholes: Meeting the grand challenge of distributed systems. In: Proc. of FuDiCo (2002)
28. Yi, S., Andrzejak, A., Kondo, D.: Monetary cost-aware checkpointing and migration on amazon cloud spot instances. IEEE Trans. on Services Computing PP(99) (2011)
29. Zhang, W.: Linux virtual server for scalable network services. In: Proc. of Linux (2000)

Network Forensics for Cloud Computing

Tobias Gebhardt[1] and Hans P. Reiser[2]

[1] TÜV SÜD AG Embedded Systems, Barthstr. 16, 80339 Munich, Germany
tobias.gebhardt@tuev-sued.de
[2] University of Passau, Innstr. 43, 94032 Passau, Germany
hans.reiser@uni-passau.de

Abstract. Computer forensics involves the collection, analysis, and reporting of information about security incidents and computer-based criminal activity. Cloud computing causes new challenges for the forensics process. This paper addresses three challenges for network forensics in an Infrastructure-as-a-Service (IaaS) environment: First, network forensics needs a mechanism for analysing network traffic remotely in the cloud. This task is complicated by dynamic migration of virtual machines. Second, forensics needs to be targeted at the virtual resources of a specific cloud user. In a multi-tenancy environment, in which multiple cloud clients share physical resources, forensics must not infringe the privacy and security of other users. Third, forensic data should be processed directly in the cloud to avoid a costly transfer of huge amounts of data to external investigators. This paper presents a generic model for network forensics in the cloud and defines an architecture that addresses above challenges. We validate this architecture with a prototype implementation based on the OpenNebula platform and the Xplico analysis tool.

Keywords: Cloud Computing, Network Forensics, Incident Investigation.

1 Motivation

Cloud computing has become highly popular over the past decade. Many organizations nowadays use virtual resources offered by external cloud providers as part of their IT infrastructure. As a second significant trend, IT-based services are getting more and more into the focus of criminal activities [11,24]. As a result, technology for computer forensics has become increasingly important. Cloud computing imposes new challenges for this technology. This paper addresses some of these challenges that concern network forensics in an Infrastructure-as-a-Service (IaaS) model.

Computer forensics is the science of collecting evidence on computer systems regarding incidents such as malicious criminal activities. Noblett et al. define it as "the science of acquiring, preserving, retrieving, and presenting data that has been processed electronically and stored on computer media" [16]. Traditionally, forensics has been associated with the collection of evidence for supporting or refuting a hypothesis before the court. In this paper, we use the term in its broader sense, which includes collecting information for purposes such as internal incident investigation and intrusion analysis.

Network forensics is the investigation of network traffic of a live system. This means that for network forensics it is necessary to capture and analyse the network traffic of the system under investigation. If an organization runs a service on its own local IT infrastructure, the responsible administrator (or any investigator with physical access to the infrastructure) can easily apply local forensics measures. Today, a great variety of forensics frameworks are available for this purpose [9].

With cloud computing, a paradigm shift takes place. Virtualization and multitenancy create novel challenges. If a cloud user wants to investigate an incident on a virtual resource, it is usually not possible for him to use traditional forensics tools that require direct access to networks and physical machines [13]. Even if such tools can be used at the physical facilities of the cloud provider, this easily creates privacy and security issues: Investigations typically target a specific system, which might be running in a virtual machine on a physical host shared with other completely unrelated systems. These other systems should not be affected by the investigation. As an additional complication, due to dynamic elasticity and migration of virtual resources, the geographical location of systems under investigation is no longer constant [15]. These issues create a new research field called cloud forensics [3,4].

The aim of this work is to propose a solution for some of these challenges. Its focus is on network forensics for the Infrastructure-as-a-Service (IaaS) model. Specifically, this paper makes the following contributions:

– It defines a generic model for network forensics in IaaS.
– It defines an architecture for "Forensics-as-a-Service" in a cloud management infrastructure. This architecture offers an API that authorized subjects can use to remotely control the forensics process at the cloud provider. Both data acquisition and data analysis can be handled directly at the cloud provider.
– It describes and evaluates a prototype implementation of this architecture for the OpenNebula cloud management infrastructure. The prototype includes a daemon running on all cloud hosts for collecting network traffic, filtered for a specific system under observation, and the integration of an existing network forensics analysis tool as a cloud-based service.

The paper is structured as follows. The next section discusses related work on computer forensics. Section 3 describes the system model and security assumptions in our approach. Section 4 presents our generic forensics model. Section 5 focuses on the forensics architecture and describes details of the prototype for OpenNebula. Section 6 evaluates this prototype, and finally Section 7 presents our conclusions.

2 Related Work

Computer forensics is a field that has undergone thorough investigation in various directions over the past decades. Within the scope of this paper, we discuss related work in the areas of *network forensics* and *cloud computing forensics.*

2.1 Network Forensics

The term *network forensics* was coined by Ranum [18]. In his paper, the author describes a forensics system called "Network Flight Recorder". This system offers functionality similar to an Intrusion Detection System (IDS), but it makes it possible to analyse data from the past.

Some existing approaches cover only parts of the forensics process. For example, practical tools such as WireShark[1] and tcpdump[2] support only the acquisition of data, without providing mechanisms for further analysis and reporting. These tools assume that you can run them on the host under investigation and have no dedicated support for remote forensics in the cloud.

Similarly, intrusion detection systems focus on the detection and reporting of attacks as they happen, without providing specific evidence gathering functionality [21]. In the context of network forensics, intrusion detection systems can be used as a trigger point for forensic investigations, as they create events upon the detection of suspicious behaviour.

Several publications in the area of network forensics target the question of how to manage and store forensic data efficiently. For example, Pilli et al. [17] focus on reducing the file size of captured data. They consider only TCP/IP headers and additionally reduce file size with a filter. Other publications focus on the analysis step of the forensics process. For example, Haggerty et al. [12] describe a service that helps to identify malicious digital pictures with a file fingerprinting service. Such research is orthogonal to the contribution of our paper.

Some existing approaches address the full forensics process. For example, Almulhem et al. [1] describe the architecture of a network forensics system that captures, records and analyses network packets. The system combines network-based modules for identifying and marking suspect packets, host-based capturing modules installed on the hosts under observation, and a network-based logging module for archiving data. A disadvantage of this approach is that the host-based capturing module cannot be trusted after an attack has compromised the host. Shanmugasundaram et al. [23] have proposed a similar approach that suffers from the same disadvantage. Wang et al. [25] propose an approach that adds a capturing tool on a host at the moment it should be inspected. Again, this approach suffers from the same integrity weakness, as data is captured by tools running on a possibly compromised system.

All of these approaches imply the assumption that there is a single entity that has permission to perform forensic investigations over all data. While this is not

[1] `http://www.wireshark.org`
[2] `http://www.tcpdump.org`

a problem if all data is owned by a single entity, this model is not appropriate for a multi-tenancy cloud architecture.

2.2 Cloud Forensics

Cloud forensics is defined as a junction of the research areas of cloud computing and digital forensics [19]. In the recent years, cloud forensics has been identified as an area that is still faced with important open research problems [2,5].

Zafarullah et al. [26] proposed an approach to analyse log files in an IaaS environment. They use a centralized approach in which the cloud service provider is responsible for forensic investigations. In contrast, in our approach we want to offer forensics services to cloud users to investigate problems and incidents in their own virtual resources.

The works of Birk et al. [4] and Grobauer et al. [10] share same aspects of our approach, as they propose a cloud API for forensics data. However, both publications list this idea together with other high level approaches, without presenting many details. Our forensics model is a more specific approach that also discusses aspects of a real prototype implementation.

3 System Model and Security Assumptions

This paper considers an IaaS model in which a cloud provider executes virtual machines on behalf of a cloud client. The client has full control over the software running within the virtual machines. The cloud provider manages the physical machines, and the client has no direct access to them. Multiple clients can share a single physical machine.

Client virtual machines can be compromised by malicious attacks. This work proposes an architecture in which cloud clients (or authorized third parties) can autonomously perform forensic investigations targeted at cloud-based virtual machines, based on support provided by the cloud provider. For this purpose, we assume that only the cloud provider and the cloud infrastructure are trusted, whereas client virtual machines are untrusted.

Having untrusted client virtual machines means that we make no assumptions on their behaviour. An attacker that compromises a virtual machine can fully modify the virtual machine's behaviour. We therefore do not want to collect forensic data with the help of processes running within the client virtual machine, as an attacker can manipulate these. We make no assumptions on how the attacker gains access to the client virtual machine. For example, the attacker might use access credentials he obtained from the cloud user by some means, or he might be able to completely take over the virtual machine by exploiting some vulnerability of the software running within the virtual machine.

Even if the adversary has full control over an arbitrary number of client virtual machines, we assume that the virtualization infrastructure guarantees isolation of virtual machines. This means that the attacker has no access to the virtual machine monitor (VMM) itself or to other VMs running on the same physical host.

We assume that the cloud provider and the cloud infrastructure can justifiably be trusted. This means that they always behave according to their specification. Under this model, the possibility that an attacker compromises the cloud infrastructure (i.e., the VMM and the cloud management system) is not considered. This might be regarded as a strong assumption, but it is the usual assumption that is made by practically all cloud users.

In addition, several research projects have shown that the trust into the cloud infrastructure can be further justified by technical mechanisms. For example, Garfinkel et al. [8] introduce a trusted virtual machine monitor (TVMM). Santos et al. [20] address the problem of creating a trustworthy cloud infrastructure by giving the cloud user an opportunity to verify the confidentiality and integrity of his own data and VMs. Doelitzscher et al. [7] designed a security audit system for IaaS. This means that trust into a cloud infrastructure can be enforced not only by contracts (SLAs), but also by technical mechanisms. The exact details of such mechanisms are beyond the scope of this paper.

4 Network Forensics Architecture for the Cloud

In this chapter we define our model for network forensics in IaaS environments and describe a generic network forensics architecture.

4.1 Forensics Process Model

The model for our forensics process is shown in Fig. 1. Five horizontal layers interact with a management component, which is needed as central point of control. The layers of the model are adopted from the process flow of the NIST forensics model [14]. The layers represent independent tasks regarding the investigation of an incident. The tasks are executed in a distributed multi-tenant environment, as described before in Section 3.

The first layer is the *Data Collection* layer. All network data can be captured at this point. The management component interactions with the data collection layer for starting and stopping the capture of network traffic. The data collection also needs to be coordinated with migration of virtual machines in the IaaS environment. For this purpose, the management component coordinates a continuous capture of network traffic for a migrating virtual machine.

On top of the data collection layer resides the *Separation* layer. The task of this layer is to separate data by cloud users. At the output of the separation layer, each data set must contain data of only a single cloud client. Optionally, the separation layer can additionally provide client-specific compression and filtering of network traffic to reduce the size of the collected forensics data.

The third layer is called *Aggregation* layer. It combines data from multiple sources belonging to the same cloud client. Data is collected at multiple physical locations if a cloud client uses multiple virtual machines (e.g., for replication or load balancing), or if a virtual machine migrates between multiple locations. All network data is combined into a single data set at this layer.

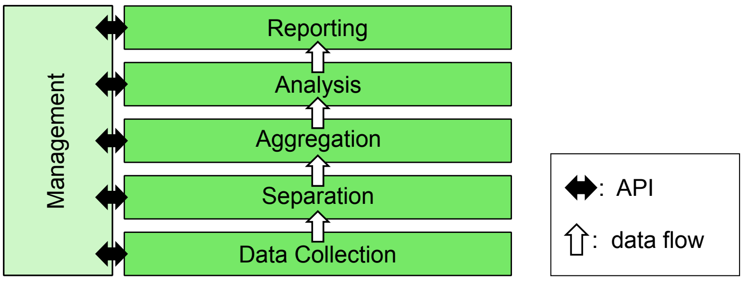


Fig. 1. The forensics process for the cloud is modelled by five basic layers, controlled by a central management block

The next layer is the *Analysis* layer. This layer receives pre-processed data sets from the aggregation layer and starts the analysis starts of the investigation. The management layer configures the transmission of collected data from aggregation to analysis. Typically, the analysis is run as a service within the same cloud.

The top layer is the *Reporting* layer in which the analysis results and consequences are presented.

4.2 Network Forensics Architecture

Our network forensics architecture directly translates the conceptional blocks of the model into separate services.

- The *management* layer is realized as part of the cloud management infrastructure. This infrastructure typically offers a central point of access for cloud clients. The infrastructure is augmented with an interface for configuring and controlling the forensics process. This component also handles authorization of forensics requests. A cloud client can control forensic mechanisms targeted at his own virtual machines, and this control privilege can also be delegated to a third party.
- The *data collection* layer is realized in the architecture by a process that executes on the local virtual machine monitor of each physical host. In a typical virtualization infrastructure, all network traffic is accessible at the VMM level. For example, if using the Xen hypervisor, all network traffic is typically handled by the Dom0 system, and thus all traffic can be captured at this place. The management layer needs to determine (and dynamically update upon reconfigurations) the physical hosts on which virtual machines of a specific client are running, and start/stop the data collection on the corresponding hosts.
- The *separation* layer is responsible for filtering data per cloud user. It is possible to investigate multiple clients on the same physical hosts, but all investigations must be kept independent. This layer separates network traffic into data sets each belonging to a single cloud client, and drops all traffic not pertaining to a client under investigation. To monitor the network data of

a specific cloud client, a means of identification of the corresponding traffic is required. In our architecture, we use a lookup table from virtual machine ID to MAC address and use this address for filtering.

- The *aggregation* layer is realized by a simple component that can collect and combine data from the separation layer at multiple locations.
- For *analysis* layer, our architecture potentially supports multiple possibilities. A simple approach is to transfer the output of the aggregation layer to the investigator, who then can locally use any existing analysis tools. In practice, the disadvantage of such approach is the high transfer cost (in terms of time and money). A better option is to run the analysis within the cloud. For this purpose, the cloud user can deploy the analysis software as a service within the cloud.
- The task of the *reporting* layer is to produce reports on the analysis results that are suitable for further distribution. This step is identical to other forensics frameworks.

5 Prototype Implementation

We have implemented a prototype according to the architecture described in the previous section. As a basis for this implementation, we have selected and extended existing projects in order to create a prototype system that is usable for network forensics in IaaS clouds. The basic idea of the prototype is the following: A cloud user has an account at a cloud provider to manage VMs. This user should be able to start and to stop the forensics process for his VMs and access analysis results using a web-based system inside the cloud.

The user's central point of interaction is the management software for IaaS. The user is able to work with the VMs he owns, to monitor or to change their status and parameters. We extend the management software by adding API commands for controlling network forensics actions. Two established systems for cloud management are Eucalyptus[3] and OpenNebula[4] [22]. Both are open source systems and have an active community. We have chosen OpenNebula for our prototype, as it supports more different VMMs and we want our prototype to be usable independent of specific VMM products.

Our goal is not to develop new analysis tools, but instead make existing tools and approaches applicable to cloud-based systems. Because of this, we did not want to implement a custom integration of the analysis part into OpenNebula. Instead, the idea is to create an interface for existing analysis software and run it as a service in the cloud. With this approach it is easy to replace it by another software if requirements changed for the analysis steps. PyFlag[5] and Xplico[6] are both network forensics analysis frameworks that could be used for analysis within our work. Cohen et al. have presented PyFlag in 2008 [6], but apparently

[3] `http://www.eucalyptus.com`
[4] `http://www.opennebula.org`
[5] `http://code.google.com/p/pyflag/`
[6] `http://www.xplico.org`

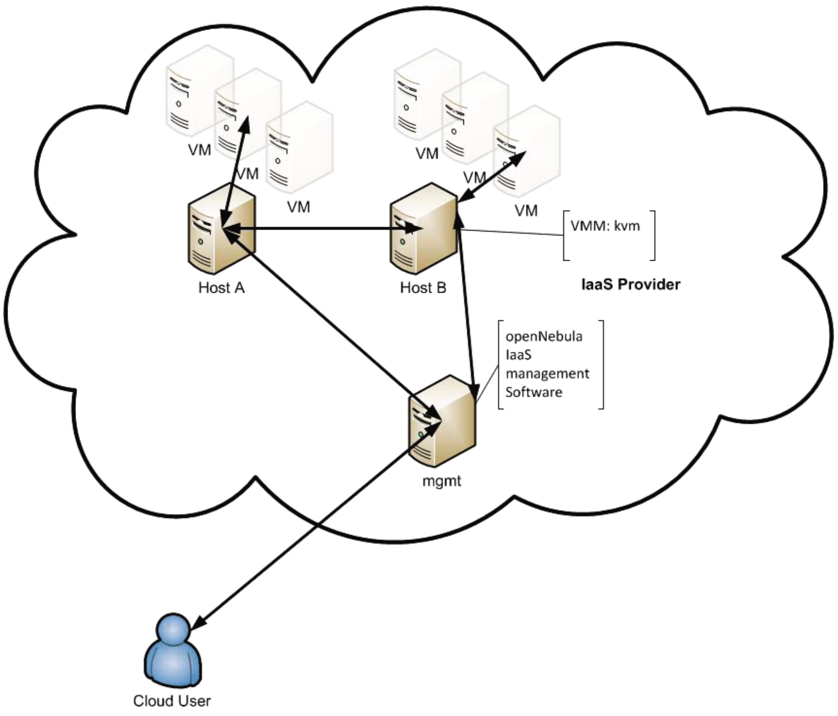


Fig. 2. In an OpenNebula-based IaaS environment, the user interacts with a central OpenNebula management instance, which in turn controls the execution of user VMs on various physical hosts

it is not being maintained any more. Xplico has a more active community and thanks to the helpful author Gianluca Costa, a developer version of Xplico is available.

In the prototype, we have implemented the *Management* layer of our architecture as an extension to the OpenNebula API. The *Aggregation* and *Analysis* layers have been implemented by a cloud-based service running Xplico, and currently the *Reporting* functionality is limited to the visualization of the Xplico output. The *Data Collection* and *Separation* layers have been realized by a custom implementation called *nfdaemon*.

The original cloud environment as provided by OpenNebula is shown in Fig. 2. Our modified environment with additional forensics components is shown in Fig. 3. The cloud user is interacting with OpenNebula, e.g. through the OpenNebula CLI. Actions to start, stop or restart virtual machines are examples for actions that are already implemented by the OpenNebula project.

Beside these actions we added the actions *startnf* and *stopnf*. The first one, *startnf*, triggers the process of starting a network forensics session. The VM-ID from OpenNebula is used as a parameter for the action to identify a particular VM. *stopnf* is doing the opposite and stops the process.

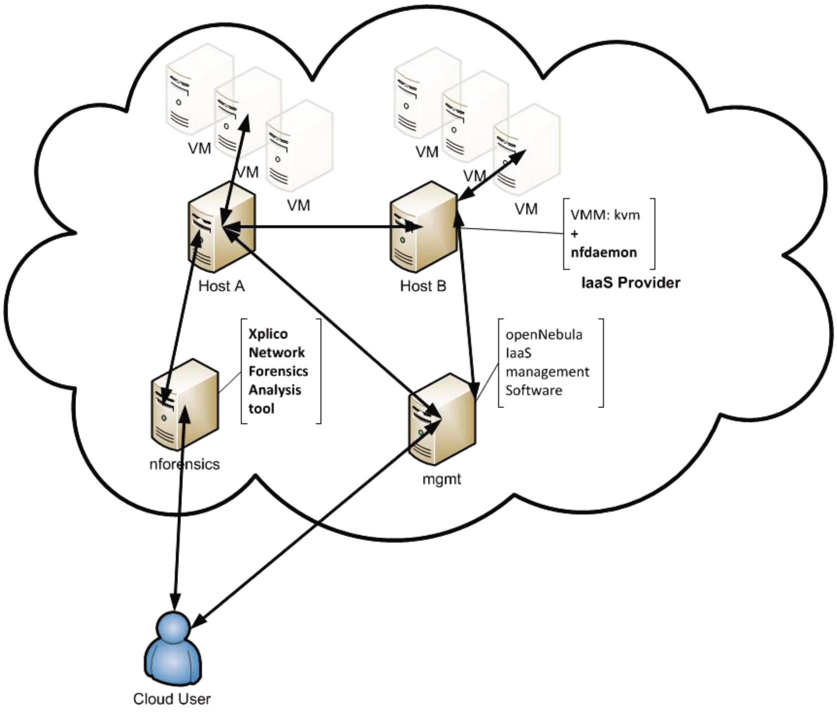


Fig. 3. The user controls forensics processes via our extended OpenNebula management component. The analysis software (Xplico) runs as a service within the cloud and offers the user a direct interface for accessing the analysis results.

A *nfdaemon* (network forensics daemon) needs to be running an each VMM (see Fig. 3). The main tasks of *nfdaemon* are collecting data, separating data per VM, and transferring it to the aggregation and analysis system. The communication interfaces are shown in Fig. 4.

The control FIFO channel is the interface for interaction between *nfdaemon* and the OpenNebula management system. The *nfdaemon* waits for input on this channel. Each command sent by the management system triggers actions of *nfdaemon*.

For filtering network data pertaining to a specific virtual machine, a mechanism for translating the VM-ID (which is used for identifying VMs by OpenNebula) to MAC network address is needed. The information about the corresponding MAC address is obtained from OpenNebula.

The *Data Collection* internally uses tcpdump to capture data. The *Separation* layer is realized on the tcpdump output on the basis of filtering by MAC addresses. The monitored data is periodically written into PCAP files ("Packet CAPture", a widely used format for network traffic dumps).

The *nfdaemon* periodically transfers PCAP files to the aggregation and analysis system. In our prototype, Xplico handles all analysis. This tool already

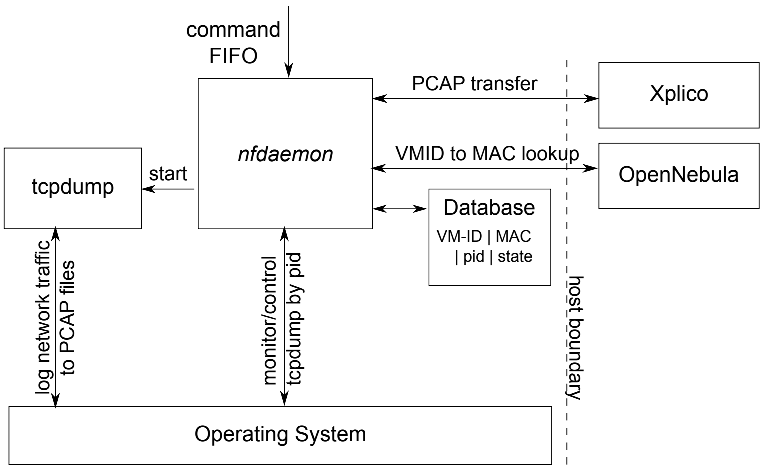


Fig. 4. The *nfdaemon* receives command via the command FIFO. It stores its state in a local database, manages tcpdump processes, and interacts with remote OpenNebula and Xplico components.

contains an interface for receiving PCAP files over the network (PCAP-over-IP). This standard interface, however, is insufficient for our prototype, as it accepts connections from any source. We wanted to make sure that the only data used for analysis is traffic collected by *nfdaemon*. Therefore, we implemented a date source authentication mechanism based on TLS, using a private key stored within *nfdaemon*. PCAP files from multiple virtual machines of the same cloud users can be aggregated within the same Xplico analysis.

6 Evaluation

As a first step for evaluating our approach, we have performed basic functionality tests using a vulnerable web application running within an OpenNebula-based cloud environment. The web application was vulnerable for remote code execution due to inappropriate input validation of a CGI script. Using our architecture, we could analyse attacks targeted at this web application using Xplico in the same way as we were able to analyse the same attack on a local system.

These basic tests convinced us that we can use our approach for forensic investigations in the cloud. For a real-world deployment of our approach, however, we wanted to analyse two additional questions: What is the performance impact of such analysis, and what are over-all benefits and implications of our approach?

6.1 Performance Evaluation

The purpose of the following performance evaluation is to verify whether our modified cloud system suffers from significant performance degradations if the

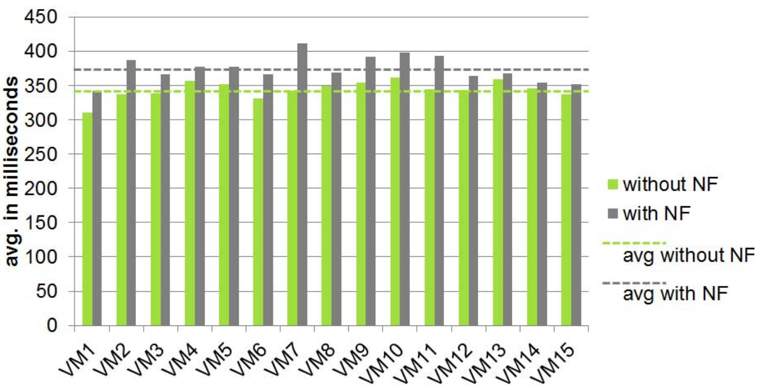


Fig. 5. Measurements show that ongoing data acquisition (NF) has an only minor impact on the running system

network forensics functionality is activated. The setup for measuring the performance impact uses a running *nfdaemon* on each VMM in a cloud environment. This process consumes computation and communication resources when capturing, processing, and transferring network traffic. The following experiment intends to quantify this performance impact by comparing two configurations with and without *nfdaemon*.

The measurements have been done on hosts with 2.8 GHz Opteron 4280 CPUs and 32 GB RAM, connected via switched Gigabit Ethernet. 15 VMs are used in parallel on one VMM for each scenario. Each VM hosts a web service that executes a computationally intensive task. The web service chooses two random values, calculates the greatest common divisor of the two values is calculated, and shows the result on its output. The calculation is repeated 1000 times for each client request. Clients iteratively call functions of this web service. Client and Xplico analysis is running on a separate host.

Clients and the service calls are realized with the tool *ab*[7]. For each VM, the web service is called 2,500 times with 25 concurrent requests. The time for each request and a summary of every 2500 calls are stored in text files. This procedure is repeated 60 times every two minutes. 150,000 measurement values are collected for each VM.

Fig. 5 shows the results of these measurements, with network forensics turned on and off. The results show a measurable but insignificant impact on the performance of the VMs. The average performance reduction between the two runs is between 2% to 17%. On average, the difference between active and inactive forensic data collection is 9%. The measurement shows that it is possible to transfer the concept of the network forensics service to a real life scenario.

[7] `http://httpd.apache.org/docs/2.0/programs/ab.html`

6.2 Discussion of Results

The measurements show the overhead for monitoring a single physical host. For a real cloud deployment, the question of scalability to a large number of hosts arises. Furthermore, the business model for the cloud provider and benefits for the cloud client need to be discussed.

Regarding the *scalability* of our approach we note that the observed overhead does not depend on the number of physical machines of the cloud provider. If a cloud client uses a single virtual machine in a large cloud, the previously shown overhead exists only on the physical machine hosting the client VM. Our approach can, however, be used to observe multiple client VMs simultaneously. In this case, a single analysis VM can become a bottleneck. A distributed approach in which multiple VMs are used to analyse forensic data could be used to leverage this problem, but this is beyond the scope of this paper.

For cloud providers, offering *Forensics-as-a-Service* is a *business model.* On the one hand, the client uses additional virtual resources for the analysis of forensics data. This utilization can easily be measured by existing accounting mechanisms. On the other hand, the overhead caused by the *nfdaemon* cannot be accounted by existing means. Therefore, we propose to estimate this overhead by measuring the amount of forensic data transferred to the aggregation and analysis component.

For the cloud user, the benefits of our approach are two-fold: Most importantly, the user re-gains the possibility to perform network-based forensic investigations on his services, even if they are running on a remote cloud. This way, the lack of control caused by cloud computing is reduced. Second, the client needs resources for performing the analysis of forensic data. With our approach, he can dynamically allocate cloud resources for the duration of the analysis.

7 Conclusion

The growing use of cloud-based infrastructure creates new challenges for computer forensics. Efficient means for remote forensics in the cloud are needed. The location of a virtual resource is often hard to determine and may even change over time. Furthermore, forensics needs to be limited to the specific system under observation in multi-tenant environments.

In this paper, we have analysed these problems with a special focus on network forensics for the Infrastructure-as-a-Service (IaaS) model in the cloud. We have defined a generic model for network forensics in the cloud. Based on this model, we have developed a system architecture for network forensics and developed a prototype implementation of this architecture. The implementation integrates forensics support into the OpenNebula cloud management infrastructure. Our approach solves three basic issues:

– It provides a remote network forensics mechanism to cloud clients. Network data acquisition and processing can be controlled remotely, independent of the physical location of virtual machines. If virtual machines migrate, the data acquisition transparently follows the location of the migrating VM.

– It ensures separation of users in a multi-tenant environment. The mechanism of acquiring data is limited to network traffic of the user's virtual machines, without infringing the privacy and security of others.
– It avoids the cost of transferring captured network data to external investigation tools by implementing the analysis step by a cloud-internal service under to control of the investigator.

An evaluation shows that the additionally needed computing power for running our data collection and processing service in parallel to an existing service is at an acceptable level. All in all, with the results of this work an organization will be able to use network forensics for a service hosted on an IaaS cloud infrastructure. This contribution eliminates a disadvantage that cloud-based services have compared to traditional services running locally on the organization's internal IT-infrastructure.

References

1. Almulhem, A., Traore, I.: Experience with engineering a network forensics system. In: Proc. of the 2005 Int. Conf. on Information Networking, Jeju (2005)
2. Beebe, N.: Digital forensic research: The good, the bad and the unaddressed. In: Peterson, G., Shenoi, S. (eds.) Advances in Digital Forensics V. IFIP AICT, vol. 306, pp. 17–36. Springer, Boston (2009)
3. Biggs, S.: Cloud computing: The impact on digital forensic investigations. In: Proc. of the 4th Int. Conf. for Internet Technology and Secured Transactions, ICITST (2009)
4. Birk, D.: Technical Challenges of Forensic Investigations in Cloud Computing Environments. In: Workshop on Cryptography and Security in Clouds, pp. 1–6 (2011)
5. Catteddu, D., Hogben, G.: Cloud Computing – Benefits, risks and recommendations for information security. ENISA Technical Report (2009)
6. Cohen, M.I.: PyFlag – an advanced network forensic framework. Digit. Investig. 5, 112–120 (2008)
7. Doelitzscher, F., Reich, C., Knahl, M., Clarke, N.: Incident Detection for Cloud Environments. In: EMERGING 2011, The Third International Conference on Emerging Network Intelligence, pp. 100–105 (2011)
8. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: a virtual machine-based platform for trusted computing. SIGOPS Oper. Syst. Rev. 37(5), 193–206 (2003)
9. Glavach, S., Zimmerman, D.: Cyber Forensics in the Cloud. IAnewsletter 14(1), 1–36 (2011)
10. Grobauer, B., Schreck, T.: Towards incident handling in the cloud. In: Proceedings of the 2010 ACM Workshop on Cloud Computing Security Workshop, CCSW 2010, pp. 77–85. ACM Press, New York (2010)
11. Grobauer, B., Walloschek, T., Stocker, E.: Understanding Cloud Computing Vulnerabilities. IEEE Security & Privacy Magazine 9(2), 50–57 (2011)
12. Haggerty, J., Llewellyn-Jones, D., Taylor, M.: FORWEB: file fingerprinting for automated network forensics investigations. In: Proceedings of the First International Conference on Forensic Applications and Techniques in Telecommunications Information and Multimedia eForensics (2008)

13. Hoopes, J., Bawcom, A., Kenealy, P., Noonan, W., Schiller, C., Shore, F., Willems, C., Williams, D.: Virtualization for Security. Syngress Publishing, Burlington (2009)
14. Kent, K., Chevalier, S., Grance, T., Dang, H.: SP800-86: Guide to Integrating Forensic Techniques into Incident Response. National Institute of Standards and Technology, Gaithersburg (2006)
15. Mather, T., Kumaraswamy, S., Latif, S.: Cloud Security and Privacy – An Enterprise Perspecive on Risks and Compliance. O'Reilly Media, Sebastopol (2009)
16. Noblett, M.G., Pollitt, M.M., Presley, L.A.: Recovering and examining computer forensic evidence. Forensic Science Communications 2(4) (2000)
17. Pilli, E.S., Joshi, R.C., Niyogi, R.: Data reduction by identification and correlation of TCP/IP attack attributes for network forensics. In: Proceedings of the International Conference & Workshop on Emerging Trends in Technology, ICWET 2011, pp. 276–283. ACM Press, New York (2011)
18. Ranum, M.J.: Network forensics and traffic monitoring. Computer Security Journal, 35–39 (1997)
19. Ruan, K., Carthy, J., Kechadi, T., Crosbie, M.: Cloud Forensics. In: Advances in Digital Forensics VII, vol. 361, pp. 35–46 (2011)
20. Santos, N., Gummadi, K.P., Rodrigues, R.: Towards trusted cloud computing. In: Proceedings of the 2009 Conference on Hot Topics in Cloud Computing, HotCloud 2009, USENIX Association, Berkeley (2009)
21. Scarfone, K., Mell, P.: Guide to intrusion detection and prevention systems. NIST Special Publication 800-94 (2007)
22. Sempolinski, P., Thain, D.: A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, 417–426. IEEE (November 2010)
23. Shanmugasundaram, K., Memon, N., Savant, A.: ForNet: A distributed forensics network. In: Second International Workshop on Mathematical Methods. Models and Architectures for Computer Networks Security (2003)
24. Somorovsky, J., Heiderich, M., Jensen, M.: All your clouds are belong to us: security analysis of cloud management interfaces. In: Proceedings of the ACM Cloud Computing Security Workshop, CCSW (2011)
25. Wang, H.-M., Yang, C.-H.: Design and implementation of a network forensics system for Linux. In: 2010 International Computer Symposium (ICS 2010), pp. 390–395. IEEE (December 2010)
26. Zafarullah, A.F., Anwar, Z.: Digital forensics for Eucalyptus. In: Proceedings of the, Frontiers of Information Technology, FIT 2011, pp. 110–116. IEEE Computer Society, Washington, DC (2011)

Dynamic Deployment of Sensing Experiments in the Wild Using Smartphones

Nicolas Haderer, Romain Rouvoy, and Lionel Seinturier

Inria Lille – Nord Europe,
LIFL - CNRS UMR 8022,
University Lille 1, France
firstname.lastname@inria.fr

Abstract. While scientific communities extensively exploit simulations to validate their theories, the relevance of their results strongly depends on the realism of the dataset they use as an input. This statement is particularly true when considering human activity traces, which tend to be highly unpredictable. In this paper, we therefore introduce *APISENSE*, a distributed crowdsensing platform for collecting realistic activity traces. In particular, *APISENSE* provides to scientists a participative platform to help them to easily deploy their sensing experiments in the wild. Beyond the scientific contributions of this platform, the technical originality of *APISENSE* lies in its Cloud orientation and the dynamic deployment of scripts within the mobile devices of the participants.We validate this platform by reporting on various crowdsensing experiments we deployed using Android smartphones and comparing our solution to existing crowdsensing platforms.

1 Introduction

For years, the analysis of activity traces has contributed to better understand crowd behaviors and habits [13]. For example, the *Urban Mobs* initiative[1] visualizes SMS or call activities in a city upon the occurrence of major public events. These activity traces are typically generated from GSM traces collected by the cellphone providers [21]. However, access to these GSM traces is often subject to constraining agreements with the mobile network operators, which restrict their publication, and have a scope limited to telecom data.

In addition to that, activity traces are also used as a critical input to assess the quality of scientific models and algorithms. As an example, the *Reality Mining* activity traces[2] collected by the MIT Media Lab or the *Stanford University Mobile Activity TRAces* (SUMATRA)[3] have become a reference testbed to validate mobile algorithms in ad hoc settings [18]. The *Community Resource for Archiving Wireless Data At Dartmouth* (CRAWDAD)[4] is another initiative from the Dartmouth College aiming at building a repository of wireless network traces. Nonetheless, the diversity of the activity traces

[1] http://www.urbanmobs.fr
[2] http://reality.media.mit.edu
[3] http://infolab.stanford.edu/pleiades/SUMATRA.html
[4] http://crawdad.cs.dartmouth.edu

available in these repositories remains limited and thus often constrains scientists to tune inadequate traces by mapping some of the parameters to their requirements. More recently, some approaches have mined the data exposed by location-based social network like Gowalla or Foursquare, but the content of these activity traces remains limited to coarse-grained locations collected from users check-ins.

In this context, we believe that cellphones represent a great opportunity to collect a wide range of crowd activity traces. Largely adopted by populations, with more than 472 millions sold in 2011 (against 297 millions in 2010) according to Gartner institute[5], smartphones have become a key companion in people's dailylife. Not only focusing on computing or communication capabilities, modern mobile devices are now equipped of a wide range of sensors enabling scientist to build a new class of datasets. Furthermore, the generalization of *app stores* or *markets* on many mobile phone platforms leverages the enrollment of participants to a larger scale than it was possible previously.

Using cellphones to collect user activity traces is reported in the literature either as *participatory sensing* [4], which requires explicit user actions to share sensors data, or as *opportunistic sensing* where the mobile sensing application collect and share data without user involvement. These approaches have been largely used in the multiples research studies including traffic and road monitoring [2], social networking [15] or environmental monitoring [16]. However, developing a sensing application to collect a specific dataset over a given population in not trivial. Indeed, a participatory and opportunistic sensing application needs to cope with a set of key challenges [6,12], including energy limitation, privacy concern and needs to provide incentive mechanisms in order to attract participants.

These constraints are making difficult, for scientists non expert in this field, to easily collect realistic datasets for their studies. But more importantly, the developed ad hoc applications may neglect privacy and security concerns, resulting in the disclosure of sensible user information. With regards to the state-of-the-art in this field, we therefore observe that current solutions lack of reusable approaches for collecting and exploiting crowd activity traces, which are usually difficult to setup and tied to specific data representations and device configurations. We therefore believe that crowdsensing platforms require to evolve in order to become more open and widely accessible to scientific communities. In this context, we introduce *APISENSE*, an open platform targeting multiple research communities, and providing a lightweight way to build and deploy opportunistic sensing applications in order to collect dedicated datasets.

This paper does not focus on user incentive challenges, which we already addressed in [10] to provide an overview of appropriate levers to encourage scientists and participants to contribute to such sensing experiments.

The remainder of this paper is organized as follows. We provide an overview of the *APISENSE* platform by detailling the server-side infrastructure as well as the client-side application (cf. Section 2). Then, we report on the case studies we deployed in the wild using *APISENSE* (cf. Section 3) before comparing our solution to the state-of-the-art approaches (cf. Section 4). Finally, we discuss the related work in this domain (cf. Section 5) before concluding (cf. Section 6).

[5] `http://www.gartner.com/it/page.jsp?id=1924314`

2 Distributed Crowdsensing Platform

The *APISENSE* platform distinguishes between two roles. The former, called *scientist*, can be a researcher who wants to define and deploy an experiment over a large population of mobile users. The platform therefore provides a set of services allowing her to describe experimental requirements in a scripting language, deploying experiment scripts over a subset of participants and connect other services to the platform in order to extract and reuse dataset collected in other contexts (*e.g.*, *visualization*, *analysis*, *replay*). Technically, the server-side infrastructure of *APISENSE* is built on the principles of Cloud computing in order to offer a modular service-oriented architecture, which can be customized upon scientist requirements. The latter is the mobile phone user, identified as a *participant*. The *APISENSE* platform provides a mobile application allowing to download experiments, execute them in a dedicated sandbox and automatically upload the collected datasets on the *APISENSE* server.

2.1 Server-Side Infrastructure

The main objective of *APISENSE* is to provide to scientist a platform, which is open, easily extensible and configurable in order to be reused in various contexts. To achieve this goal, we designed the server-side infrastructure of *APISENSE* as an SCA distributed system (cf. Figure 1). The *Service Component Architecture* (SCA)[6] standard is a set of specifications for building distributed application based on *Service-Oriented Architectures* (SOA) and *Component-Based Software Engineering* (CBSE) principles.

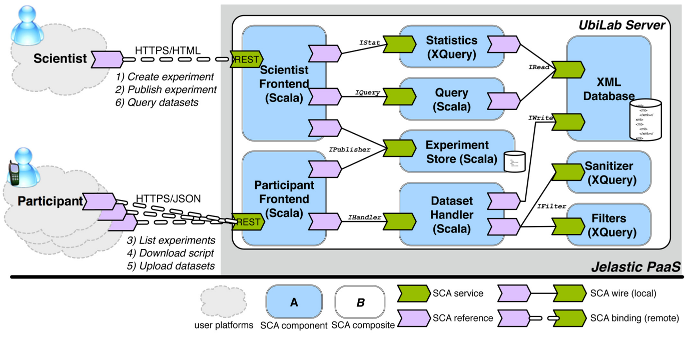


Fig. 1. Architecture of the *APISENSE* Web Infrastructure

All the components building the server-side infrastructure of *APISENSE* are hosted by a Cloud computing infrastructure [17]. The *Scientist Frontend* and *Participant Frontend* components are the endpoints for the two categories of users involved in the platform. Both components define all the services that can be remotely invoked by the scientists or the participants. For example, once authenticated, the scientist can create new experiments, follow their progression, and exploit the collected dataset directly from this web interface.

[6] `http://www.osoa.org`

Crowdsensing Library. To reduce the *learning curve*, we decided to adopt standard scripting languages in order to ease the description of experiments by the scientists. We therefore propose the *APISENSE* crowdsensing library as an extension of the JavaScript, CoffeeScript, and Python languages, which provides an efficient mean to describe an experiment without any specific knowledge of mobile device programming technologies (*e.g.*, Android SDK). The choice of these host languages was mainly motivated by their native support for JSON (*JavaScript Object Notation*), which is a lightweight data-interchange format reducing the communication overhead.

The *APISENSE* crowdsensing library adopts a reactive programming model based on the enlistment of handlers, which are triggered upon the occurence of specific events (cf. Section 3). In addition to that, the API of *APISENSE* defines a set of sensing functions, which can be used to retrieve specific contextual data form sensors. The library supports a wide range of features to build dataset from built-in sensors proposed by smartphones technologies, such as GPS, compass, accelerometers, bluetooth, phone call, application status (installed, running) in the context of *opportunistic* crowdsensing, but also to specify participatory sensing experiments (*e.g.*, surveys).

Privacy Filters. In addition to this script, the scientist can configure some privacy filters to limit the volume of collected data and enforce the privacy of the participants. In particular, *APISENSE* currently supports two types of filters. **Area filter** allows the scientist to specify a geographic area where the data requires to be collected. For example, this area can be the place where the scientist is interested in collecting a GSM signal (*e.g.*, campus area). This filter guarantees the participants that no data is collected and sent to the *APISENSE* server outside of this area. **Period filter** allows the scientist to define a time period during which the experiment should be active and collect data. For example, this period can be specified as the working hours in order to automatically discard data collected during the night, while the participant is expected to be at home.

By combining these filters, the scientist preserves the privacy of participants, reduces the volume of collected data, and improves the energy efficiency of the mobile application (cf. Section 4).

Deployment Model. Once an experiment is defined with the crowdsensing library, the scientist can publish it into the *Experiment Store* component in order to make it available to participants. Once published, two deployment strategies can be considered for deploying experiments. The former, called pull-based approach, is a proactive deployment strategy where participants download the list of experiments from the remote server. The latter, known as push-based approach, propagates the experiments list updates to the mobiles devices of participants. In the case of *APISENSE*, the push-based strategy would induce a communication and energy overhead and, in order to leave the choice to participants to select the experiments they are interested in, we adopted the pull-based approach as a deployment strategy. Therefore, when the mobile device of a participant connects to the *Experiment Store*, it sends its characteristics (including hardware, current location, sensor available and sensors that participants want to share) and receives the list of experiments that are compatible with the profile of the participant. The scientists can therefore configure the *Experiment Store* to limit the visibility of their experiments according the characteristics of participants. In order to reduce the privacy risk, the device characteristics sent by the participants are not stored by the infrastructure and scientist cannot access to this information.

Additionally, the *Experiment Store* component is also used to update the behavior of the experiment once deployed in the wild. When an opportunistic connection is established between the mobile device and the *APISENSE* server, the version of the experiment deployed in the mobile device is compared to the latest version published in the server. The installed crowdsensing experiment is automatically updated with the latest version of the experiment without imposing participants to re-download manually the experiment. In order to avoid any versioning problem, each dataset uploaded automatically includes the version of the experiment used to build the dataset. Thus, scientists can configure the *Experiment Store* component in order to keep or discard datasets collected by older versions of the experiment.

2.2 Client-Side Library

Although our solution could be extended to other *Operating Systems*, the *APISENSE* mobile application is currently based on the Android operating system for the following reasons. First, the Android operating system is popular and largely adopted by the population, unit sales for Android OS smartphones were ranked first among all smartphone OS handsets sold worldwide during 2011 with a market share of 50.9% according to Gartner. Secondly, Android is an open platform supporting all the requirements for continuous sensing applications (*e.g.*, multitasking, background processing and ability to develop an application with continuous access to all the sensors), while for example, iOS 6 does no permit a continuous accelerometer sampling.

A participant willing to be involved in one or more crowdsensing experiments proposed by a scientist can download the *APISENSE* mobile application by flashing the QR code published on *APISENSE* website, install it, and quickly create an account on the remote server infrastructure. Once registered, the HTTP communications between the mobile device of the participant and the remote server infrastructure are authenticated and encrypted in order to reduce potential privacy leaks when transferring the collected datasets to the *APISENSE* server. From there, the participant can connect to the *Experiment Store*, download and execute one or several crowdsensing experiments proposed by scientists.

Figure 2 depicts the *APISENSE* software architecture. Building on the top of Android SDK, this architecture is mainly composed of four main parts allowing *i*) to interpret experiment scripts (*Facades*, *Scripting engine*) *ii*) to establish connection with the remote server infrastructure (*Network Manager*), *iii*) to control the privacy parameters of the user (*Privacy Manager*), and *iv*) to control power saving strategies (*Battery Manager*).

Scripting Engine. *Sensor Facades* bridge the Android SDK with the *Scripting Engine*, which integrates scripting engines based on the JSR 223 specification. We build a middleware layer *Bee.sense Scripting*, which exposes the sensors that can be accessed from the experiment scripts. This layer covers three roles: a *security role* to prevent malicious calls of critical code for the mobile device, a *efficiency role* by including a cache mechanism to limit system calls and preserve the battery, and an *accessibility role* to leverage the development of crowdsensing experiments, as illustrated in Section 3.

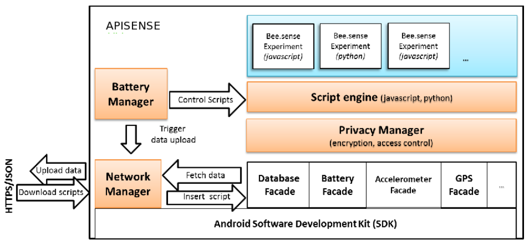

APISENSE
Bee.sense Experiment (javascript)
Bee.sense Experiment (python)
Bee.sense Experiment (javascript)
...
Battery Manager
Control Scripts
Script engine (javascript, python)
Trigger data upload
Privacy Manager (encryption, access control)
HTTPS/JSON
Upload data
Download scripts
Network Manager
Fetch data
Insert script
Database Facade
Battery Facade
Accelerometer Facade
GPS Facade
...
Android Software Development Kit (SDK)

Fig. 2. Architecture of the *APISENSE* Mobile Application

Battery Manager. Although the latest generations of smartphones provides very powerful computing capabilities, the major obstacle to enable continuous sensing application refers to their energy restrictions. Therefore, in order to reduce the communication overhead with the remote server, which tends to be energy consuming, datasets are uploaded only when the mobile phone is charging. In particular, the battery manager component monitors the battery state and triggers the network manager component when the battery starts charging in order to send all the collected datasets to the remote server. Additionally, this component monitors the current battery level and suspends the scripting engine component when the battery level goes below a specific threshold (20% by default) in order to stop all running experiments. This threshold can be configured by the participant to decide the critical level of battery she wants to preserve to keep using her smartphone.

Privacy Manager. In order to cope with the ethical issues related to crowdsensing activities, the *APISENSE* mobile application allows *participants* to adjust their privacy preferences in order to constrain the conditions under which the experiments can collect data. As depicted in Figure 3, three categories of privacy rules are currently defined. Rules related to *location* and *time* specify geographical zone and time intervals conditions under which experiments are authorized to collect data, respectively. All the privacy rules defined by the participant are interpreted by the *Privacy Manager* component, which suspends the scripting engine component when one these rules is triggered. The last category of privacy rules refers to *authorization rules*, which prevent sensors activation or access to raw sensor data if the participant does not want to share this information. Additionally, a built-in component uses cryptography hashing to prevent experiments from collecting sensitive raw data, such as phone numbers, SMS text, or address book.

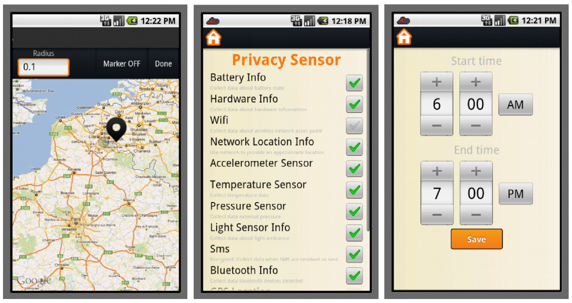
3 Crowdsensing Experiments

This section reports on four experiments that have been deployed in the wild using our platform. These examples demonstrate the variety of crowdsensing experimentations that are covered by the *APISENSE* infrastructre.

3.1 Revealing Users' Identity from Mobility Traces

This first experiment aimed at identifying the potential privacy leaks related to the sporadic disclosure of user's locations. To support this experiment, we developed a

APISENSE script, which reports every hour the location of a participant, known as Alice. Listing 1.1 describes the Python script we used to realize this experiment. This script subscribes to the periodic scheduler provided by the `time` facade in order to trigger the associated lambda function every hour. This function dumps a timestamped longitude/latitude position of Alice, which is automatically forwarded to the server.



Radius 0.1

Privacy Sensor

Battery Info Collect data about battery state	✓
Hardware Info Collect data about hardware information	✓
Wifi Collect data about wireless network acces point	
Network Location Info Use network to provide an approximate location	✓
Accelerometer Sensor	✓
Temperature Sensor Collect temperature data	✓
Pressure Sensor Collect data external pressure	✓
Light Sensor Info Collect data about light ambiance	✓
Sms Encrypted: Collect data when SMS are received or sent.	✓
Bluetooth Info Collect data bluetooth devices detected	✓

Start time 6 00 AM

End time 7 00 PM

Save

Fig. 3. Participant Privacy Preferences

```
time.schedule({'period': '1h'},
  lambda t: trace.add({ 'time': t.timestamp,
                         'lon': gps.longitude(), 'lat': gps.latitude() }))
```

Listing 1.1. Identifying GeoPrivacy Leaks (Python)

While this periodic report can be considered as anonymous since no identifier is included in the dataset, this study has shown that the identity of Alice can be semi-automatically be inferred from her mobility traces. To do so, we built a mobility model from the dataset we collected in order to identify clusters of Alice's locations as her *points of interest* (POI). By analyzing the size of the clusters and their relative timestamps, we can guess that the largest POI in the night relates to the house of Alice. Invoking a geocoding service with the center of this POI provides us a postal address, which can be used as an input to the yellow pages in order to retrieve a list of candidate names. In parallel, we can identify the places associated to the other POIs by using the Foursquare API, which provides a list of potential places where Alice is used to go. From there, we evaluate the results of search queries made on Google by combining candidate names and places and we rank the names based on the number of pertinent results obtained for each name. This heuristic has demonstrated that the identity of a large population of participants can be easily revealed by sporadically monitoring her location [11].

3.2 Building WiFi/GSM Signal Open Data Maps

This second experiment illustrates the benefits of using *APISENSE* to automatically build two open data maps from datasets collected in the wild. Listing 1.2 is a JavaScript script, which is triggered whenever the location of a participant changes by a distance of 10 meters in a period of 5 minutes. When these conditions are met, the script builds a trace which contains the location of the participant and attaches WiFi and GSM networks characteristics.

```
trace.setHeader('gsm_operator', gsm.operator());
location.onLocationChanged({ period: '5min',
  distance: '10m' }, function(loc) {
    return trace.add({
      time: loc.timestamp,
      lat: loc.latitude, lon: loc.longitude,
      wifi: { network_id: wifi.bssid(),
              signal_strength: wifi.rssi() },
      gsm: { cell_id: gsm.cellId(),
             signal_strength: gsm.dbm() } });
});
```

Listing 1.2. Building an Open Data Map (JavaScript)

From the dataset, collected by three participants over one week, we build an QuadTree geospatial index to identify the minimum bounding rectangles that contain at least a given number of signal measures. These rectangles are then automatically colored based on the median signal value observed in this rectangle (cf. Figure 4). This map has been assessed by comparing it with a ground truth map locating the GSM antennas and WiFi routers[7].

3.3 Detecting Exceptions Raised by User Applications

The third experiment highlights that *APISENSE* does not impose to collect geolocated dataset and can also be used to build realistic dataset focusing on the exceptions that are raised by the participants' applications. To build such a dataset, Listing 1.3 describes a CoffeeScript script that uses the Android logging system (`logCat`) and subscribes to error logs (`'*:E'`). Whenever, the reported log refers to an exception, the script builds a new trace that contains the details of the log and retrieves the name of the application reporting this exception.

```
logcat.onLog {filter: '*:E'},
  (log) -> if log.message contains 'Exception'
    trace.save
      message: log.message,
      time: log.timestamp,
      application: apps.process(log.pid).applicationName,
      topTask: apps.topTask().applicationName
```

Listing 1.3. Catching Mobile Applications' Exceptions (CoffeeScript)

Once deployed in the wild, the exceptions reported by the participants can be used to build a taxonomy of exceptions raised by mobile applications. The Figure 5, depicts the results of this experiment based on a dataset collected from three participants over one month. In particular, one can observe that a large majority of errors reported by the participant's applications are related to permission or database accesses, which can usually be fixed by checking that the application is granted an access prior to any invocation of a sensor or the database. This experiment is a preliminary step in order to better identify bugs raised by applications once they are deployed in the wild as we believe that the diversity of mobile devices and operating conditions makes difficult the application of traditional *in vitro* testing techniques.

[7] http://www.cartoradio.fr

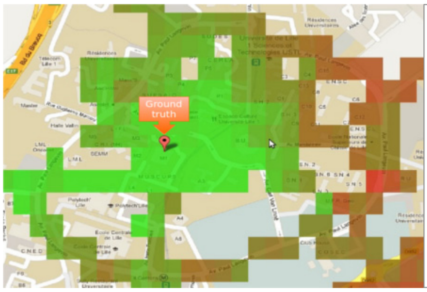


Fig. 4. GSM Open Data Map

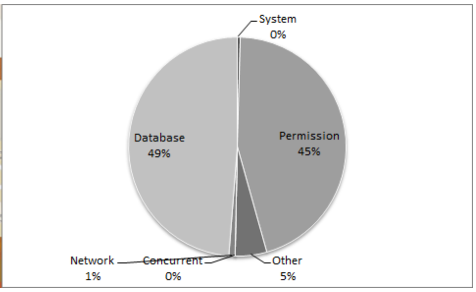


Fig. 5. Exceptions Taxonomy

3.4 Experimenting Machine Learning Models

The fourth experiment does not only collect user-contributed datasets, but also deals with the empirical validation of models on a population of participants. In this scenario, the scientist wanted to assess the machine learning model she defined for detecting the activity of the users: *walking*, *sitting*, *standing*, *running*, or *climbing and down stairs*. To assess this model, she deployed a script that integrates two phases: an exploration phase and an exploitation one. To set up this experiment, we extended the scripting library by integrating a popular machine learning [14] and adding a new facade to use its features from script. The script (cf. Listing 1.4) therefore starts with an exploration phase in order to learn a specific user model. During this phase, *APISENSE* generates some dialogs to interact with the participant and ask her to repeat some specific movements. The script automatically switches to the next movement when the model has recorded enough raw data from the accelerometer to provide an accurate estimation. Once the model is considered as complete, the script dynamically replace the timer handler to switch into the exploration phase. The dataset collected by the server-side infrastructure of *APISENSE* contain the model statisitcs observed for each participant contributing to the experiment.

Figure 6 reports on the collected statistics of this experiment and shows that the prediction model developed by the scientist matches quite accurately the targeted classes.

	Predicted class						**Acc** (%)
	Walk	Jog	Stand	Sit	Up	Down	
Walk	**66**	0	4	0	0	0	94,3
Jog	0	**21**	0	0	0	0	100
Stand	4	0	**40**	0	0	0	90,9
Sit	0	0	2	**83**	0	0	97,6
Up stair	0	0	0	0	**22**	0	100
Down stair	0	0	0	0	0	**11**	100

Fig. 6. Representative Confusion Matrix

```
var classes = ["walk","jog","stand", "sit", "up", "down"];
var current = 0; var buffer = new Array();
var model = weka.newModel(["avrX","avrY",...], classes);
var filter = "|(dx>"+delta+")(dy>"+delta+")(dz>"+delta+")";

var input = accelerometer.onChange(filter,
   function(acc) { buffer.push(acc) });

var learn = time.schedule({ period: '5s' }, function(t) {
  if (model.learn(classes[current]) >= threshold) {
    current++;
  }
  if (current < classes.length) { // Learning phase
    input.suspend();
    var output = dialog.display({ message: "Select movement", spinner: classes });
    model.record(attributes(buffer), output);
    sleep('2s');
    buffer = new Array();
    input.resume();
  } else { // Exploitation phase
    dialog.display({message: "Learning phase completed"});
    learn.cancel();
    model.setClassifier(weka.NAIVE_BAYES);
    time.schedule({ period: '5s' }, function(t) {
      trace.add({
        position: model.evaluate(attributes(buffer)),
          stats: model.statistics() });
      buffer = new Array();
} } });
```

Listing 1.4. Assessing Machine Learning Models (JavaScript)

4 Empirical Validations

Evaluating the Programming Models. In this section, we compare the *APISENSE* crowdsensing library to two state-of-the-art approaches: ANONYSENSE [20] and POGO [3]. We use the *RogueFinder* case study, which has been introduced by AnonySense and recently evaluated by POGO. Listings 1.5 and 1.6 therefore reports on the implementation of this case study in ANONYSENSE and POGO, as decribed in the literature, while Listing 1.7 describes the implementation of this case study in *APISENSE*.

```
(Task 25043) (Expires 1196728453)
(Accept (= @carrier 'professor'))
(Report (location SSIDs) (Every 1 Minute)
(In location
  (Polygon (Point 1 1) (Point 2 2)
  (Point 3 0))))
```

Listing 1.5. Implementing *RogueFinder* in ANONYSENSE

One can observe that *APISENSE* provides a more concise notation to describe crowdsensing experiments than the state-of-the-art approaches. This concisition is partly due to the fact that *APISENSE* encourages the separation of concerns by externalizing the management of time and space filters in the configuration of the experiment. A direct impact of this property is that the execution of *APISENSE* scripts better preserves the battery of the mobile device compared to POGO, as it does not keep triggering the script when the user leaves the assigned polygon. Nonetheless, this statement is only based on an observation of POGO as the library is not made freely available to confirm this empirically.

```
function start() {
  var polygon = [{x:1, y:1}, {x:2, y:2}, {x:3, y:0}];
  var subscription = subscribe('wifi-scan', function(msg){
    publish(msg, 'filtered-scans');
  }, { interval: 60 * 1000 });
  subscription.release();
  subscribe('location', function(msg) {
    if (locationInPolygon(msg, polygon))
      subscription.renew();
    else
      subscription.release();
  });
}
```

Listing 1.6. Implementing *RogueFinder* in POGO (JavaScript)

```
time.schedule { period: '1min' },
  (t) -> trace.add { location: wifi.bssid() }
```

Listing 1.7. Implementing *RogueFinder* in *APISENSE* (CoffeeScript)

Evaluating the Energy Consumption. In this section, we compare the energy consumption of *APISENSE* to a native Android application and another state-of-the-art crowdsensing solution: FUNF [1]. FUNF provides an Android toolkit to build custom crowdsensing applications à la carte. For each technology, we developed a sensing application, which collects the battery level every 10 minutes. We observed the energy consumption of these applications and we report their consumption in Figure 7.

Compared to the baseline, which corresponds to the native Android application, one can observe that the overhead induced by our solution is lower than the one imposed by the FUNF toolkit. This efficiency can be explained by the various optimizations included in our crowdsensing library. Although more energyvorous than a native application, our solution does not require advanced skills of the Android development framework and covers the deployment and reporting phases on behalf of the developer.

As the energy consumption strongly depends on *i)* the nature of the experiment, *ii)* the types of sensors accessed, and *iii)* the volume of produced data, we conducted a second experiment in order to quantify the impact of sensors (cf. Figure 8). For this experiment, we developed three scripts, which we deployed separately. The first script, labelled Bee.sense + Bluetooth, triggers a Bluetooth scan every minute and collects both the battery level as well as the resulting Bluetooth scan. The second script, Bee.sense + GPS, records every minute the current location collected from the GPS sensor, while the third script, Bee.sense + WiFi, collects a WiFi scan every minute. These experiments demonstrate that, even when stressing sensors, it is still possible to collect data during a working day without charging the mobile phone (40% of battery left after 10 hours of pulling the GPS sensor).

5 Related Work

A limited number of data collection tools are freely available on the market. SYSTEMSENS [8], a system based on Android, focuses on collecting usage context (*e.g.*, CPU, memory, network info, battery) of smartphones in order to better understand the battery consumption of installed applications. Similarly, LIVELABS [19] is a tool to measure

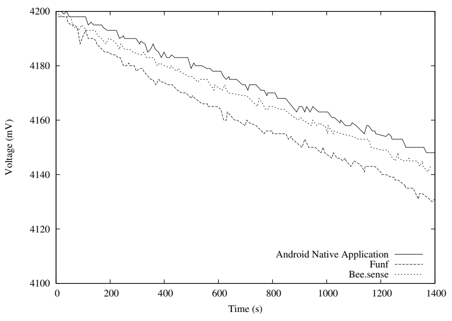


Fig. 7. Energy Consumptions of Android, *APISENSE*, and FUNF

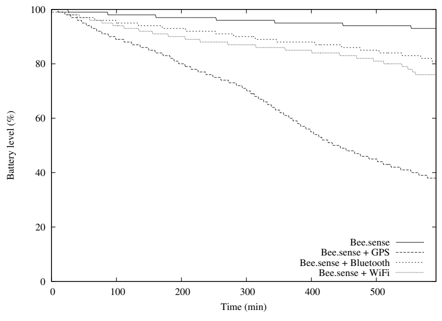


Fig. 8. Impact of *APISENSE* on the Battery Lifespan

wireless networks in the field with the principal objective to generate a complete network coverage map in order to help client to select network interface or network operators to identify blind spots in the network. However, all these tools are closed solutions, designed for collecting specific datasets and cannot be reused in unforeseen contexts in contrast to *APISENSE*. Furthermore, these projects are typical experiments deployed on mobile devices, without providing any privacy guarantee.

FUNF [1] is an Android toolkit focusing on the development of sensing applications. FUNF in a box is a service provided by FUNF to easily build a dedicated sensing application from a web interface, while data is periodically published via the Dropbox service. As demonstrated in Section 4, the current version of FUNF does not provide any support for saving energy nor preserving user privacy. Furthermore, the current solution does not support the dynamic re-deployment of experiments once deployed in the wild.

More interestingly, MYEXPERIENCE [9] is a system proposed for Windows mobile smartphones, tackling the *learning curve* issue by providing a lightweight configuration language based on XML in order to control the features of the application without writing C# code. MYEXPERIENCE collects data using a *participatory* approach—*i.e.*, by interacting with users when a specific event occurs (*e.g.*, asking to report on the quality of the conversation after a phone call ends). However, MYEXPERIENCE does not consider severals critical issues, such as maintaining the privacy of participants or the strategic deployment of experiments. Even if an experiment can be modified in the wild, each experiment still requires a physical access to the mobile device in order to be installed, thus making it difficult to be applied on a large population of participants.

In the literature, severals deployment of crowdsensing applications strategies have been studied. For example, ANONYSENSE [20] uses—as *APISENSE*—a pull-based approach where mobile nodes periodically download all sensing experiments available on the server. A crowdsensing experiment is written in a domain-specific language and defines when a mobile node should sense and under which conditions the report should be submitted to the server. However, ANONYSENSE does not provide any mechanism to filter the mobile nodes able to download sensing experiments, thus introducing a communication overhead if the node does not match the experiment requirements.

On the contrary, PRISM [7] and POGO [3] adopts a push-based approach to deploy sensing experiments over mobile nodes. PRISM is a mobile platform, running on

Microsoft Windows Mobile 5.0, and supporting the execution of generic *binary code* in a secure way to develop real-time participatory sensing applications. To support real-time sensing, PRISM server needs to keep track of each mobile node and the report they periodically send (*e.g.*, current location, battery left) before selecting the appropriate mobile phones to push application binaries. POGO proposes a middleware for building crowdsensing applications and using the XMPP protocol to disseminate the datasets. Nonetheless, POGO does not implement any client-side optimizations to save the mobile device battery (*e.g.*, area and period filters) as it systematically forwards the collected data to the server.

SensorSafe [5] is another participatory platform, which allows users to share data with privacy guaranties. As our platform, *SensorSafe* provides fine-grained temporal and location access control mechanisms to keep the control of data collected by sensors on mobile phone. However, participants have to define their privacy rules from a web interface while in *APISENSE* these rules are defined directly from the mobile phone.

6 Conclusion

While it has been generally acknowledged as a keystone for the mobile computing community, the development of crowdsensing platforms remains a sensitive and critical task, which requires to take into account a variety of requirements covering both technical and ethical issues.

To address these challenges, we report in this paper on the design and the implementation of the *APISENSE* distributed platform. This platform distinguishes between two roles: *scientists* requiring a sustainable environment to deploy sensing experiments and *participants* using their own mobile device to contribute to scientific experiments. On the server-side, *APISENSE* is built on the principles of Cloud computing and offers to scientists a modular service-oriented architecture, which can be customized upon their requirements. On the client-side, the *APISENSE* platform provides a mobile application allowing to download experiments, executing them in a dedicated sandbox and uploading datasets to the *APISENSE* server. Based on the principle of *only collect what you need*, the *APISENSE* platform delivers an efficient yet flexible solution to ease the retrieval of realistic datasets.

References

1. Aharony, N., Pan, W., Ip, C., Khayal, I., Pentland, A.: Social fmri: Investigating and shaping social mechanisms in the real world. In: Pervasive and Mobile Computing (2011)
2. Biagioni, J., Gerlich, T., Merrifield, T., Eriksson, J.: EasyTracker: automatic transit tracking, mapping, and arrival time prediction using smartphones. In: 9th Int. Conf. on EmbeddedNetworked Sensor Systems. ACM (November 2011)
3. Brouwers, N., Langendoen, K.: Pogo, a Middleware for Mobile Phone Sensing. In: Narasimhan, P., Triantafillou, P. (eds.) Middleware 2012. LNCS, vol. 7662, pp. 21–40. Springer, Heidelberg (2012)
4. Burke, J.A., Estrin, D., Hansen, M., Parker, A., Ramanathan, N., Reddy, S., Srivastava, M.B.: Participatory Sensing (2006)

5. Choi, H., Chakraborty, S., Greenblatt, M., Charbiwala, Z.M., Srivastava, M.B.: Sensorsafe: Managing health-related sensory information with fine-grained privacy controls. Technical report (TR-UCLA-NESL-201009-01) (September 2010)
6. Cuff, D., Hansen, M., Kang, J.: Urban Sensing: Out of the Woods. Communications of the ACM 51(3) (2008)
7. Das, T., Mohan, P., Padmanabhan, V.N., Ramjee, R., Sharma, A.: Prism: Platform for Remote Sensing Using Smartphones. In: 8th Int. Conf. on Mobile Systems, Applications, and Services. ACM (2010)
8. Falaki, H., Mahajan, R., Estrin, D.: SystemSens: a tool for monitoring usage in smartphone research deployments. In: 6th ACM Int. Work on Mobility in the Evolving Internet Architecture (2011)
9. Froehlich, J., Chen, M.Y., Consolvo, S., Harrison, B., Landay, J.A.: Myexperience: a system for in situ tracing and capturing of user feedback on mobile phones. In: 5th Int. Conf. on Mobile Systems, Applications, and Services. ACM (2007)
10. Haderer, N., Rouvoy, R., Seinturier, L.: A preliminary investigation of user incentives to leverage crowdsensing activities. In: 2nd International IEEE PerCom Workshop on Hot Topics in Pervasive Computing (PerHot). IEEE (2013)
11. Killijian, M.-O., Roy, M., Trédan, G.: Beyond San Fancisco Cabs: Building a *-lity Mining Dataset. In: Work. on the Analysis of Mobile Phone Networks (2010)
12. Lane, N.D., Miluzzo, E., Lu, H., Peebles, D., Choudhury, T., Campbell, A.T.: A Survey of Mobile Phone Sensing. IEEE Communications Magazine 48(9) (2010)
13. Liu, L., Andris, C., Biderman, A., Ratti, C.: Uncovering Taxi Driver's Mobility Intelligence through His Trace. In: IEEE Pervasive Computing (2009)
14. Liu, P., Chen, Y., Tang, W., Yue, Q.: Mobile weka as data mining tool on android. In: Advances in Electrical Engineering and Automation, pp. 75–80 (2012)
15. Miluzzo, E., Lane, N.D., Lu, H., Campbell, A.T.: Research in the App Store Era: Experiences from the CenceMe App Deployment on the iPhone. In: 1st Int. Work. Research in the Large: Using App Stores, Markets, and Other Wide Distribution Channels in UbiComp Research (2010)
16. Mun, M., Reddy, S., Shilton, K., Yau, N., Burke, J., Estrin, D., Hansen, M., Howard, E., West, R., Boda, P.: PEIR, The Personal Environmental Impact Report, as a Platform for Participatory Sensing Systems Research. In: 7th Int. Conf. on Mobile Systems, Applications, and Services. ACM (2009)
17. Paraiso, F., Haderer, N., Merle, P., Rouvoy, R., Seinturier, L.: A Federated Multi-Cloud PaaS Infrastructure. In: 5th IEEE Int. Conf. on Cloud Computing (2012)
18. Roy, M., Killijian, M.-O.: Brief Announcement: A Platform for Experimenting with Mobile Algorithms in a Laboratory. In: 28th Annual ACM Symp. on Principles of Distributed Computing, ACM (2009)
19. Shepard, C., Rahmati, A., Tossell, C., Zhong, L., Kortum, P.: LiveLab: measuring wireless networks and smartphone users in the field. ACM SIGMETRICS Performance Evaluation Review 38(3) (2011)
20. Shin, M., Cornelius, C., Peebles, D., Kapadia, A., Kotz, D., Triandopoulos, N.: AnonySense: A System for Anonymous Opportunistic Sensing. In: Pervasive and Mobile Computing (2010)
21. Sohn, T., et al.: Mobility Detection Using Everyday GSM Traces. In: Dourish, P., Friday, A. (eds.) UbiComp 2006. LNCS, vol. 4206, pp. 212–224. Springer, Heidelberg (2006)

AJITTS: Adaptive Just-In-Time Transaction Scheduling

Ana Nunes, Rui Oliveira, and José Pereira

HASLab / INESC TEC and U. Minho
{ananunes,rco,jop}@di.uminho.pt

Abstract. Distributed transaction processing has benefited greatly from optimistic concurrency control protocols thus avoiding costly fine-grained synchronization. However, the performance of these protocols degrades significantly when the workload increases, namely, by leading to a substantial amount of aborted transactions due to concurrency conflicts.

Our approach stems from the observation that when the abort rate increases with the load as already executed transactions queue for longer periods of time waiting for their turn to be certified and committed. We thus propose an adaptive algorithm for judiciously scheduling transactions to minimize the time during which these are vulnerable to being aborted by concurrent transactions, thereby reducing the overall abort rate. We do so by throttling transaction execution using an adaptive mechanism based on the locally known state of globally executing transactions, that includes out-of-order execution.

Our evaluation using traces from the industry standard TPC-E workload shows that the amount of aborted transactions can be kept bounded as system load increases, while at the same time fully utilizing system resources and thus scaling transaction processing throughput.

Keywords: Optimistic concurrency control, adaptive scheduling.

1 Introduction

Optimistic concurrency control in distributed data processing systems is increasingly popular. In replicated database systems [1–3], it allows concurrent transactions to execute at different sites regardless of possible conflicts. Conflict detection and resolution are performed at commit time, before the changes are applied to the database. In large scale, high throughput transactional systems such as Google Percolator [4] and OMID [5], implementations of optimistic concurrency control with different isolation levels and locking policies are key to achieving radical scalability.

While optimistic concurrency control allows more concurrency and thus better use of resources than its counterpart, transactions that are later found to conflict are aborted and must be re-executed. Notice that the more transactions are allowed to execute concurrently, the more likely it is for conflicts to arise. Also, any transaction is vulnerable to being aborted by other transactions from the moment it starts to execute until is is certified: the longer it takes to execute and certify a given transaction, the more vulnerable it is. This is the caveat of most optimistic concurrency control strategies: when loaded, latency increases and fairness is compromised, particularly for long-running transactions [1].

In contrast, conservative concurrency control is implemented by subjecting transactions to a priori conflict detection. In some of these protocols, conflict detection is done by associating queues to data partitions [3, 6–8]. The queues are used to serialize transactions that access the same data partitions. The performance penalty imposed by the conservative strategy depends on the grain considered for concurrency control: if the grain is too fine, conflict detection will result in a delay before transaction execution; on the other hand, if the grain is too coarse, transactions that would not otherwise conflict are unnecessarily prevented from executing concurrently [1].

It has been proposed that both approaches be combined by conservatively re-executing previously aborted transactions, which mitigates this issue. However, if appropriate conflict classes cannot be used to ensure no conflicts occur during re-execution, the system will quickly be prevented from exploting optimism [1]. Moreover, although most optimistic concurrency control protocols execute transactions as soon as these are submitted [2, 3], it has been pointed out that the worst scenarios for optimistic concurrency control can be mitigated by limiting the number of transactions executing concurrently [1]. Transaction scheduling on non-distributed settings using queue-theoretic models for automatically adjusting the maximum parallelism level has been studied [9]. However, selecting the correct level of parallelism is not straightforward and can result in a severe limitation to maximum throughput.

In this paper we solve this problem with AJITTS, an adaptive just-in-time transaction scheduler. This mechanism minimize aborts while maximizing transaction throughput by computing the appropriate start time for each transaction. The intuition behind this proposal is simple: If a transaction must wait to be certified in the correct order, to ensure consistency in a distributed system, it is better that it waits prior to execution, as it is not susceptible to being aborted by conflicts with concurrent transactions. The implementation of this simple intuition does however imply that the system is continually monitored and that an appropriate execution start time is computed for each transaction.

The rest of this paper is structured as follows. In Section 2 we show that there is an ideal configuration for each workload, that improves performance regarding the basic optimistic protocol, introducing then the mechanism used to dynamically compute such configuration. In Section 3 we use traces obtained from the TPC-E workload running on a MySQL database server to simulate different workload scenarios and evaluate our proposal. Finally, Section 4 concludes the paper.

2 Approach

The main insight leading to our proposal is as follows: Since transactions are vulnerable to being aborted from the time execution starts until certification, in order to minimize the number of aborts, execution should start as late as possible. On the other hand, if certification goes idle because transactions at the head of the queue have not been completely executed, the throughput decreases. Our approach is thus based on reaching and maintaining the optimal level of queuing in the system: As low as possible to minimize aborts but as high as needed to ensure that certification doesn't go idle, to maximize throughput.

2.1 System Model

To test this hypothesis, we assume an abstract model that captures key aspects of a distributed transaction processing system. First, we assume that transactions submitted to the system are totally ordered and placed at the tail of a queue in the *not_executed* state. This models either a centralized queue at the transaction manager server [5] or local replicas of a queue built by a group communication system [2]. Because transactions must be certified in a conflict-equivalent order to the total order on which replicas agreed, the system can be modelled as a single queue in which transactions go through several states.

We then introduce a line in the queue that determines which transactions should start execution: all transactions before the line are not eligible to start executing, while all transactions between the line and the head of the queue that are in the *not_executed* state are to be executed. Simply put, transactions are evaluated for execution whenever a transaction arrives to (i.e. is submitted) or leaves the queue (i.e. committed or aborted). Transactions can only be certified upon reaching the head of the queue and having completed execution, entering the *certification* state.

Conflict detection ensues: if the transaction was in the *executed* state, conflicting transactions in either *executing* or *executed* states are aborted and the transaction is immediately certified; if it was in the *aborted* state, the outcome is an abort. For certification, we assume *snapshot isolation*, which differs from serializability by considering only write/write conflicts [10]. This is used in the overwhelming majority of current RDBMSs and has also been favored in distributed transaction processing systems.

2.2 Impact of Scheduling

Ideally, the line would be placed at such a position that each transaction completes execution just as it arrives at the head of the queue, minimizing the time spent in the *executed* state before reaching *certification*, thus minimizing its vulnerability to being aborted by others. However if the transaction reaches the head of the queue in either *not_executed* or *executing* states, it cannot be certified until it finishes. Certification must occur in a conflict-equivalent order to the previously established total order, which is key to guaranteeing determinism in a distributed setting. As such, transactions running late cannot be overtaken by others, leaving certification idle.

Transactions can have widely varying execution times (i.e. duration), which should be considered when scheduling them. Let d_t be the estimate of the duration of transaction t, which can be easily obtained from a database planner.

Assuming that the head of the queue corresponds to position 1, let

$$pos_t = input \cdot d_t \tag{1}$$

be the position in the queue after which transaction t will be executed. This means that transaction t will be executed when there are $pos_t - 1$ or less transactions ahead of it in the queue, which is the same as placing a line in the queue. Notice that transaction t is not scheduled for a particular instant in absolute time: it is relative to the current inter-arrival rate of transactions at the head of the queue. This enables out-of-order execution: a small transaction will begin execution near the head of the queue, while a very large transaction will begin execution as soon as it is submitted.

The *input* parameter provides a simple way to adjust how early transactions should be executed: for the same estimated duration, a higher value of *input* means that the transaction will be executed earlier than with a lower value.

We then built an event-driven simulation of the abstract system model with the adjustable executuon start line. This simulation allows the position of the line that triggers execution to be adjusted, as well as to use different workloads, further described in Section 3. From this simulation we collect a number of statistics, namely: usable throughput, considering transactions that can be committed; share of transactions aborted due to concurrency conflicts; and latency at each state, from which we derive also end-to-end latency.

Figure 1 shows the latency breakdown for a particular workload (400 clients) while varying the input parameter. On the right hand, transactions are scheduled early, thus reducing the amount of time in the *not_executed* state, shown in blue. In fact, an extreme setting of the parameter is equivalent to the baseline optimistic protocol, meaning that transactions are immediatly scheduled for exection and the entire impact of synchronization happens in the *executed* state. On the left, transactions are scheduled later, thus waiting an increased amount of time before execution, but waiting very little as *executed* (in brown). As expected, varying this parameter does not have an impact in execution duration (in red). As expected, we observe that overly delaying transaction execution has an impact in total latency.

Figure 2 shows a complete set of statistics for three different workloads. These workloads differ only in the number of concurrent clients submitting transactions. First, we observe that besides impacting end-to-end latency, the input parameter that determines when execution is started also impacts throughput and the abort rate leading to the following trade off:

– On the left, with a larger delay before execution, transactions arrive at the head of the queue but are not yet fully executed, thus stalling certification

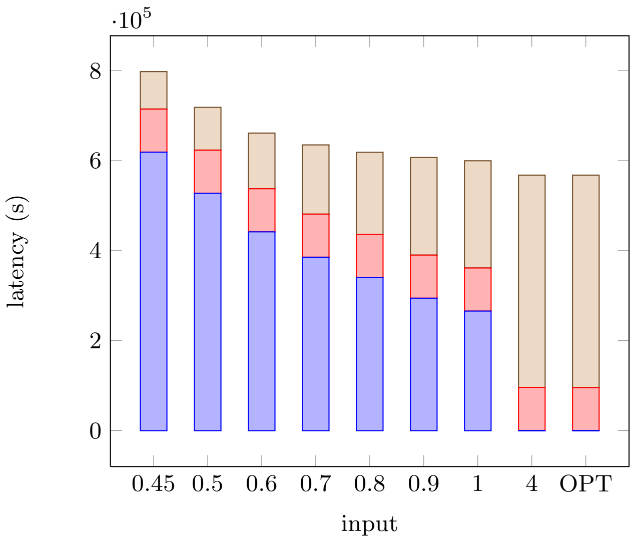


Fig. 1. Average latency breakdown with a varying scheduler parameter: Pre-execution delay (blue), execution latency (red), and queueing for certification (brown)

and leading to reduced throughput. However, the small delay to certification leads to a reduced number of concurrency aborts.

– On the right, transactions are executed fairly ahead of time, thus avoiding stalling the queue. On the other hand, by having started early they become concurrent with a larger number of transactions and thus lead to an increased amount of rollbacks due to conflicts.

Notice that, for example, if *input* is between $0.4 \cdot 10^{-3}$ and $0.9 \cdot 10^{-3}$ for 800 clients, throughput is sub-optimal because transactions are being executed too late. This can be confirmed by analyzing the transaction latency in the same interval. Also, for example for 200 clients, the abort rate steadily rises as *input* increases, but when *input* becomes larger than 1, the abort rate stabilizes at around 5%. This happens because after this point roughly all transactions are being executed as soon as they are submitted, equivalent to using DBSM [2]. This effect occurs for any number of clients, it is just a question of using a large enough *input*.

Figure 3 shows similar results when, instead of varying the workload, we vary the resources available for execution. This leads to the time spent in the *executing* state growing. However, the same trade off holds. In short, we observe that there is an intermediate configuration that provides the best usable throughput with moderate latency. This is true regardless of the number of concurrent clients or the resources available to execute transactions. This optimal configuration is however different for different settings, which makes its configuration by the system developer unfeasible. As it varies with the workload, it is also impractical as a configuration parameter.

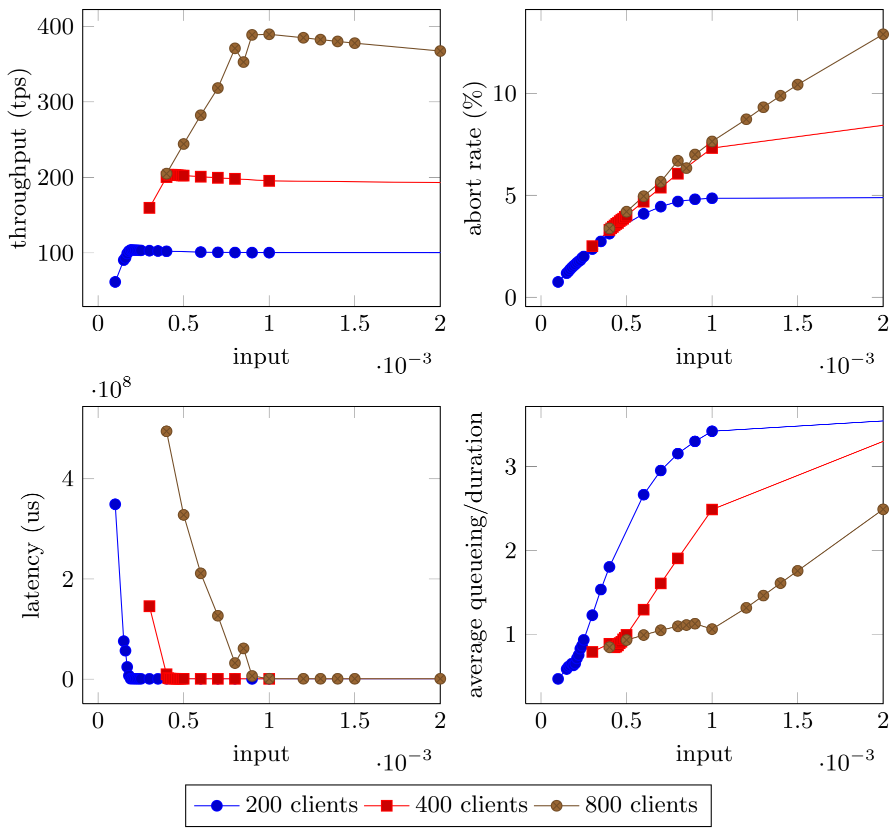


Fig. 2. Effect of the input value on throughput, the abort rate, transaction latency and on the ratio between average transaction queueing and average duration for different numbers of clients

2.3 Adapting the Workload

The question becomes how to determine an appropriate *input* value that provides optimal throughput without resorting to trial and error.

A simple adaptation mechanism would be to simply start execution one position sooner whenever a transaction reaches the head of the queue in the *not_executed* or *executing* states or one position later whenever it has to wait in the *executed* state or has been *aborted*. This approach was tested, but such an adaptation mechanism, while simple, causes oscillation, since such changes are too abrupt [11].

Let t_{state} be the instant in which transaction t reaches state *state*. For any transaction t,

$$s_t = t_{not\_executed} - t_{executing}$$

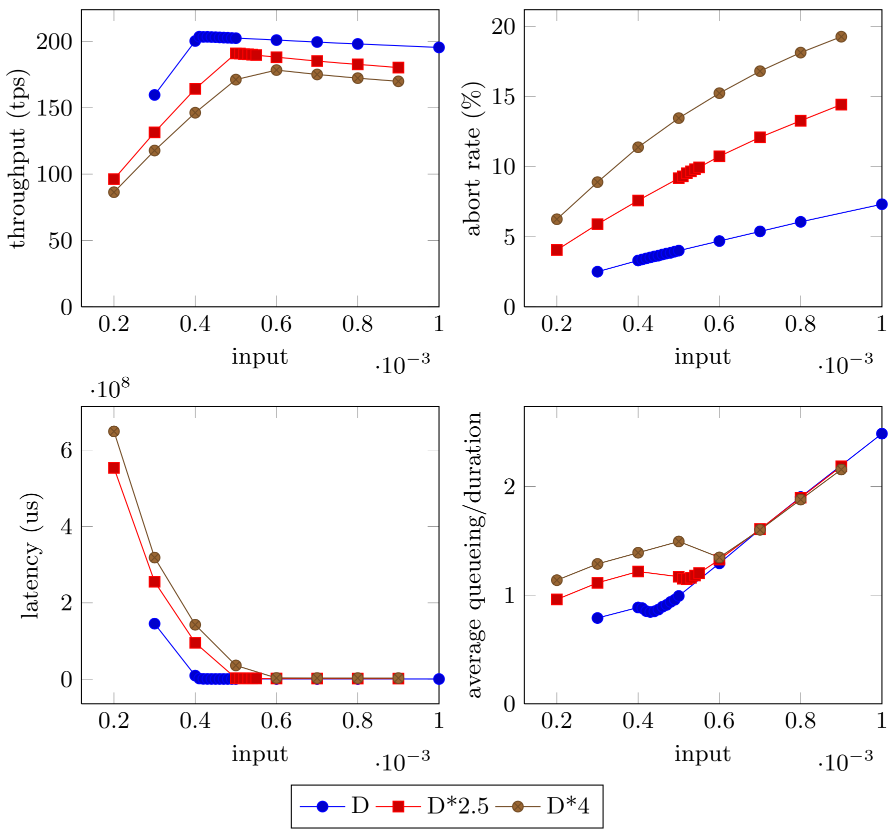


Fig. 3. Effect of the input value on throughput, the abort rate, transaction latency and on the ratio between average transaction queueing and average duration for different distributions of transaction duration

measures how long transaction t had to wait to begin to execute since it was submitted, and

$$q_t = t_{certification} - t_{executed}$$

measures how long transaction t had to wait after its execution was complete before reaching the head of the queue to be certified, henceforth simply referred to as pre-execution delay and queuing, respectively. Queuing is directly affected by whether transactions are executed sooner or later: on average, the former increases queuing while the latter decreases it. Let Q be a weighted cumulative rolling average of q and Q_{opt} the optimal level of queuing for a system. An adaptive mechanism that reacts to the state of the queue can be defined using a proportional-integral-derivative controller[11] with Q as the sensor, Q_{opt} as the set point and *input* as the system input [11] such that:

$$\begin{aligned} error &= setpoint - sensor \\ P_{value} &= Kp * error \\ input{+} &= P_{value} \end{aligned}$$

Simply put, the error of the measured value (*sensor*) relatively to the desired value (*setpoint*) is used to update an input to the system (*input*) which will in turn impact the measured value, constituting the control feedback loop.

Kp is referred to as proportional gain, a tuning parameter that adjusts how the sensitivity of the controller, *i.e.*, the magnitude of the adaptation relatively to the magnitude of the error. Several methods exist for selecting an appropriate value for Kp, from manual tuning to methods based on heuristics[12].

Because computing the position of the line relies on an estimated value for transaction duration, for which the expected value is the mean, depending on the variance of the population, selecting a set point of 0 would mean that several transactions would not finish its execution in time, leaving certification idle.

The key to finding Q_{opt} is in comparing the bottom-right chart of Figure 2 which shows the ratio between the average queuing and the average duration of all transactions with the top-left chart showing throughput. Notice that the *input* values that achieve optimal throughput in the top-left chart match those for which the ratio in the bottom-right chart is roughly 1. Intuitively, selecting the set point to target average duration would mean that certification does not go idle and, consequently, that the rate at which transactions are certified is the same as the rate at which transactions arrive at the head of the queue which corresponds to optimal throughput but minimizing the size of the queue meaning a minimal abort rate. If deemed necessary, due to high variance in transaction duration, from the cumulative distribution function of transaction duration one can choose a value for the set point corresponding to a desired percentile: the higher the percentile of the chosen value, the higher the number of transactions that will have completed execution as expected.

3 Evaluation

3.1 Workload

TPC-E [13] is a benchmark that simulates the activities of a brokerage firm which handles customer account management, trade order execution on behalf of customers and the interaction with financial markets. This analysis was based on *tpce-mysql*,[1] an open-source implementation of the TPC-E benchmark. This benchmark defines 33 tables across four domains: customer, broker, market and dimension and 10 main transaction types that operate across the domains.

[1] `https://code.launchpad.net/perconadev/perconatools/tpcemysql`

TPC-E's read/write transactions are: Market Feed (MF), Trade Order (TO), Trade Result (TR), Trade Update (TU) and Data Maintenance (DM).[2]

AJITTS was evaluated using a simple event-driven simulator that enables a profound analysis of each aspect of scheduling and concurrency control of replication protocols. The simulator processes execution traces obtained by running TPC-E like benchmark on a centralized MySQL[3] database and then parsing the resulting binlog to generate the workload to test replication protocols. In essence, the simulator uses the following information from the binlog: the timestamps at which each transaction started, how long it took to execute each transaction and transaction write sets.

Another relevant feature of the simulator is that it allows the parallelization of the load generated by serial runs of TPC-E over the same database. In short, it does so by creating unique identifiers for each transaction and by manipulating timestamps making these relative to a reference instant. As a result, the load applied to the protocol under test can be easily scaled. Also, the applied load is not limited by resource constraints on the original MySQL database: there is no limit on the number of load units that can be applied in parallel.

The values of transaction duration extracted from the binlog reflect the penalty introduced by synchronization and locking in the MySQL engine when the benchmark is executed. A correction factor (β) can be calibrated by running the traces through the simulator with optimistic scheduling, without admission restriction and without re-execution, chosen such that the abort rate is close to 1%. The reason for this is that the sequence of transactions in the binlog has implicitly been proved to be conflict-free with the original values for transaction duration.

Let dur'_t be the duration extracted from the binlog for transaction t. The respective value to be used in the simulation is $dur_t = \beta * dur'_t$.

The value of the correction factor depends on the benchmark load induced on MySQL. Therefore, the β used in the simulation is independent of the number of parallel traces used to fuel the simulator, as long as the load induced by each benchmark run was about the same. If using another set of traces, the beta must be recalculated. β is 0.2 for the traces used to evaluate AJITTS.

As discussed in Section 2, the set point should be chosen taking into consideration the distribution of transaction duration. In the results presented here, the set point used in AJITTS is the same as the average transaction duration of the given workload.

In order to simplify the implementation of AJITTS, instead of estimating the duration of each transaction, a line is placed on the queue for each type of transaction. The position of the line for a transaction of type MF, for example, is calculated as

$$line_{MF} = input * d_{MF}$$

[2] The Data Maintenance transaction type operates exclusively on a separate group of tables. As such, it is not relevant for this analysis and is essentially omitted from the discussion that follows.

[3] `http://www.mysql.com`

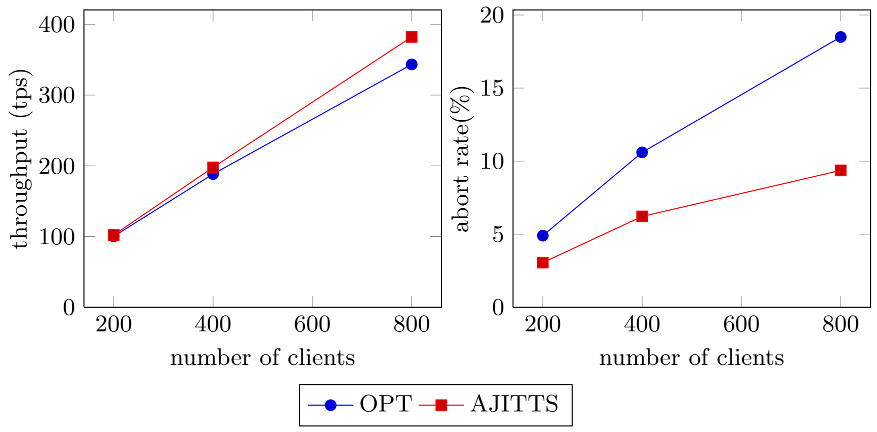


Fig. 4. Throughput and abort rates using OPT and AJITTS for different numbers of clients

where d_{MF} is an online estimate of the duration of MF transactions using a cumulative rolling average.

While TPC-E already provides strictly defined transaction types, one could classify transactions in generic workloads by either considering the similarity in execution plans or, for example, the conflict classes accessed by transactions. Still, AJITTS can be implemented without this simplification, computing a line for each individual transaction.

3.2 Results

We compare AJITTS with OPT, a protocol with a standard optimistic scheduler. Simply put, OPT schedules each execution as soon as it is submitted. Using the TPC-E based workload described in Section 3.1, this is simulated by admitting at most one transaction per client, since clients are single-threaded. Notice that without this restriction, the number of concurrent transactions would be higher than allowed in the original benchmark.

Figure 4 compares OPT and AJITTS in terms of throughput and aborts for three workloads that differ only on the number of concurrent clients submitting transactions. Notice that even though AJITTS introduces delays on transaction executions, throughput is not only not adversely affected, but actually improved. Also, AJITTS clearly succeeds in significantly reducing the abort rate. In fact, a clear trend of further improvement can be observed in both charts as the load increases.

Figure 5 shows how the line positions per transaction type evolve during a run with a particular workload. Line positions are updated whenever the estimates for execution duration change or whenever the adaptation input parameter changes. The position of the line for each transaction type converges quickly: the

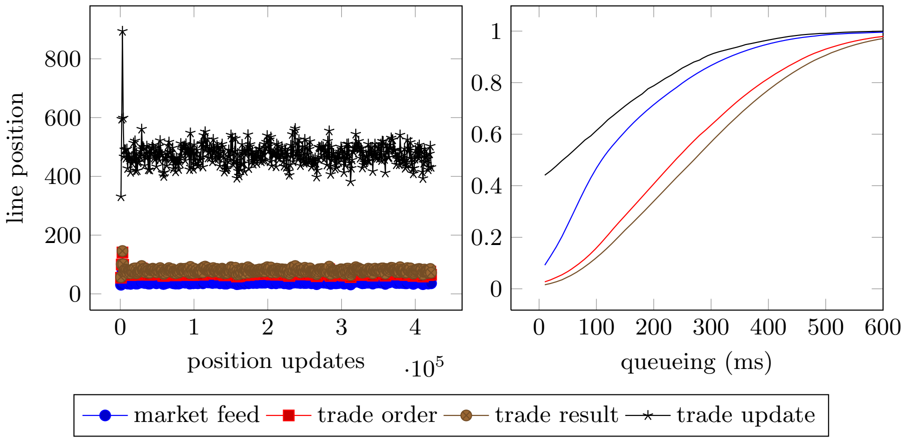


Fig. 5. Evolution of the position of the lines during a particular run

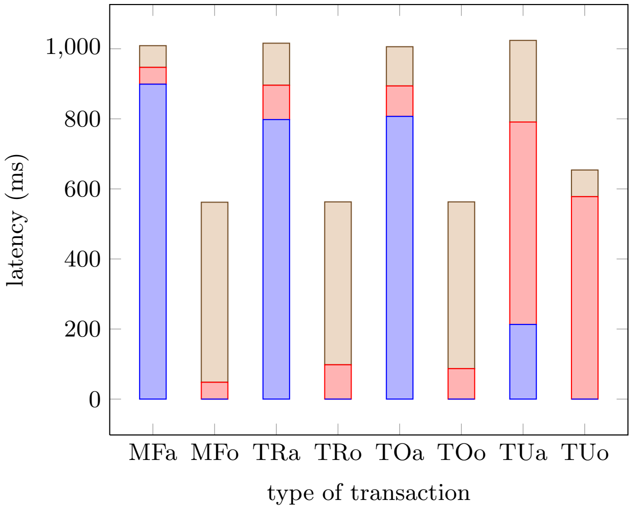


Fig. 6. Average latency breakdown using AJITTS and OPT: Pre-execution delay (blue), execution latency (red), and queueing for certification (brown). Columns MFa, TRa, TOa and TUa refer to an execution of the AJITTS protocol, while the others refer to an execution of the OPT protocol.

amplitude of the variation stabilizes after considerably few updates. In particular, TU transactions actually consist of three different types of subtransactions: the variability of the duration of trade update transactions is mirrored in the variation of the position of the line for this type of transaction. Notice that TU transactions are scheduled much earlier than other types of transactions. Figure 5 also shows the cumulative distribution function of the measured queueing (q) aggregated by transaction type, which is a result of the position of the lines.

Figure 6 shows how the different average durations (in red) influence the pre-execution delay (in blue) when using AJITTS: again, TU transactions (TUa) are scheduled much earlier than others, while MF transactions (MFa), for instance, are only executed nearer the head of the queue. When compairing the results regarding, for example, MF transactions, the average time during which these are vulnerable to being aborted much smaller using AJITTS (110 ms) than using OPT (562 ms). This is also the case for TR and TO transactions. However, for TU transactions queueing actually increases using AJITTS. This is a consequence of ensuring that certification does not go idle. As expected, the net effect is still a reduced abort rate.

Considering different values of β shapes the workload: higher βs simulate less available resources and vice-versa. Figure 7 shows how AJITTS leverages available resources significantly better than OPT. In particular, the less available resources, the more OPT's throughput decreases relatively to AJITTS.

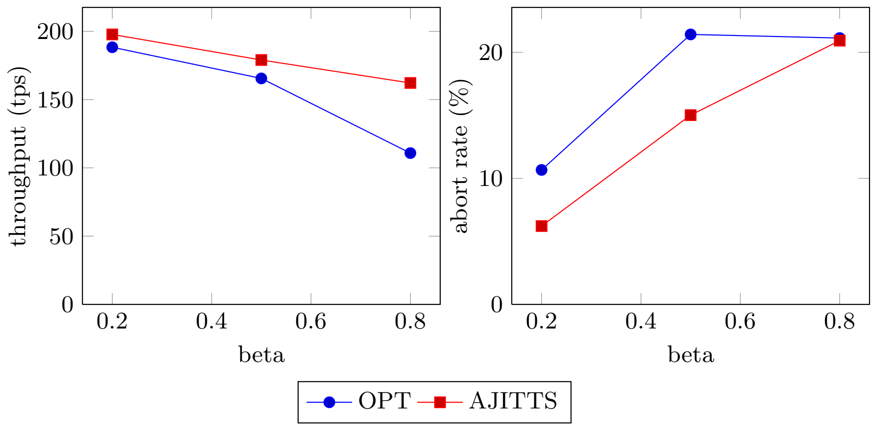


Fig. 7. Throughput and abort rates for OPT and AJITTS for different βs

4 Conclusion

Although increasingly popular and often used, optimistic concurrency control may lead, with more demanding workloads, to a large number of conflicts and aborted transactions. This endangers fairness and reduces usable throughput. Previous attempts at tackling this problem required workload-specific configuration and would still impact peak throughput [1].

With AJITTS, the adaptive just-in-time transaction scheduler, we provide a solution that does not require workload specific configuration and adapts in runtime to current workload and resource availability conditions. This is achieved by delaying transaction execution, for each transaction individually based on the estimated time to complete and current queueing within the system.

AJITTS was then evaluated using a simulation model driven by traces from TPC-E running on MySQL, demonstrating that it clearly outperforms the baseline protocol. In fact, in addition to reduced aborts, it actually improves peak throughput even if it throttles transaction execution. This is the consequence of using available resources better.

Acknowledgements. This work is part-funded by: ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project Stratus/FCOMP-01-0124-FEDER-015020.

References

1. Correia Jr., A., Pereira, J., Oliveira, R.: AKARA: A flexible clustering protocol for demanding transactional workloads. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part I. LNCS, vol. 5331, pp. 691–708. Springer, Heidelberg (2008)
2. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Distributed and Parallel Databases 14, 71–98 (2003), doi:10.1023/A:1022887812188
3. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In: Proceedings of the 26th International Conference on Very Large Data Bases, VLDB 2000, pp. 134–143. Morgan Kaufmann Publishers Inc., San Francisco (2000)
4. Peng, D., Dabek, F., Inc, G.: Large-scale incremental processing using distributed transactions and notifications. In: 9th USENIX Symposium on Operating Systems Design and Implementation, pp. 4–6 (2010)
5. Yabandeh, M., Gómez Ferro, D.: A critique of snapshot isolation. In: Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys 2012, pp. 155–168. ACM, New York (2012)
6. 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. 147–160. Springer, Heidelberg (2000)
7. Jiménez-Peris, R., Patiño-Martínez, M., Kemme, B., Alonso, G.: Improving the scalability of fault-tolerant database clusters. In: Proceedings of IEEE 22nd International Conference on Distributed Computing Systems, pp. 477–484 (2002)
8. Kemme, B., Pedone, F., Alonso, G., Schiper, A.: Processing transactions over optimistic atomic broadcast protocols. In: Proceedings of 19th IEEE International Conference on Distributed Computing Systems, pp. 424–431 (1999)
9. Schroeder, B., Harchol-Balter, M., Iyengar, A., Nahum, E., Wierman, A.: How to determine a good multi-programming level for external scheduling. In: Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, p. 60 (April 2006)

10. Lin, Y., Kemme, B., Patiño-Martínez, M., Jiménez-Peris, R.: Middleware based data replication providing snapshot isolation. In: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 419–430. ACM (2005)
11. Aström, K.J., Murray, R.M.: Feedback systems: An introduction for scientists and engineers. Technical report, Princeton University Press (2007)
12. Aström, K., Hägglund, T.: Automatic tuning of simple regulators with specifications on phase and amplitude margins. Automatica 20(5), 645–651 (1984)
13. Transaction Processing Performance Council (TPC): TPC Benchmark E - Standard Specification. Revision 1.12.0 edn. (June 2010)

Strategies for Generating and Evaluating Large-Scale Powerlaw-Distributed P2P Overlays

Ana-Maria Oprescu[1], Spyros Voulgaris[2], and Haralambie Leahu[2]

[1] Universiteit van Amsterdam
a.m.oprescu@uva.nl
[2] VU University
spyros@cs.vu.nl, h.leahu@vu.nl

Abstract. A very wide variety of physical, demographic, biological and man-made phenomena have been observed to exhibit powerlaw behavior, including the population of cities and villages, sizes of lakes, etc. The Internet is no exception to this. The connectivity of routers, the popularity of web sites, and the degrees of World Wide Web pages are only a few examples of measurements governed by powerlaw. The study of powerlaw networks has strong implications on the design and function of the Internet.

Nevertheless, it is still uncertain how to explicitly *generate* such topologies at a very large scale. In this paper, we investigate the generation of P2P overlays following a powerlaw degree distribution. We revisit and identify weaknesses of existing strategies. We propose a new methodology for generating powerlaw topologies with predictable characteristics, in a completely decentralized, emerging way. We provide analytical support of our methodology and we validate it by large-scale (simulated) experiments.

1 Introduction

Many real-world large-scale networks demonstrate a power-law degree distribution, that is, a very small fraction of the nodes has a high *degree* (i.e., number of neighbors), while the vast majority of nodes has a small degree. In nature, such networks typically emerge over time, rather than being instantiated on the spot based on a blueprint. Providing researchers from different disciplines with a framework that allows them to control the self-emerging process of power-law networks, could substantially help them in studying and better understanding such networks, as well as deploying them at will to serve new applications (e.g., bio-inspired algorithms for peer-to-peer systems).

There are several algorithms to generate power-law networks, however little has been done for a self-emerging method for building such networks [3,5,6,4]. In this work, we first investigate existing research with an emphasis on the decentralization properties of proposed algorithms. Next, we select one approach that looks promising for straightforward decentralization. We identify several limitations within the existing approach and we present a novel algorithm that has been tailored specifically to the needs of a large P2P network. Starting from a given, static distribution of random values among the P2P network nodes, we control the emerging power-law overlay .

We summarize related research conducted on power-law generation in Section 2, where we assess the degree to which such approaches may be decentralized. In Section 3.1 we identify several limitations (both theoretical and empirical) with an existing sequential approach and proceed to present a novel algorithm to alleviate the respective issues. In Section 4 we show how the decentralized algorithm may be implemented in a P2P network and present our evaluation results. We summarize our findings in Section 5.

2 Related Work

There is a vast literature on properties and characteristics of scale-free and small-world networks. The research behind such literature is focused on the observation of aforementioned topologies and their behavior (like finding the λ value) rather than construction methodologies. However, there are several important generative mechanisms which produce specific models of power-law networks. It started with the *Erdös and Rényi* random-graph theory and continued with the *Watts and Strogatz* model, which was the first to generate a small-world topology from a regular graph, by random rewiring of edges. Drawbacks of this initial model are its degree homogeneity and static number of nodes. These limitations can be addressed by scale-free networks, but the clustering coefficient becomes an issue. In turn, the clustering coefficient can be controlled through the employed generative mechanism. However, generating a random scale-free network having a specific λ value is not trivial. Moreover, most existing algorithms to generate scale-free networks are centralized and their decentralization, again, far from trivial. We present several types of generative models.

Preferential Attachment. This model, also known as the “rich-get-richer” model, combines preferential attachment and growth. It assumes a small initial set of m_0 nodes, with $m_0 > 2$, forming a connected network. The remaining nodes are added one at a time. Each new node is attached to m existing nodes, chosen with probabilities proportional to their degrees. This model is referred to as the Barabási-Albert (BA) model [2], though it was proposed by Derek J. de Solla Price [7] in 1965 and Yule in 1925 [12]. The degree distribution is proven to be $P(k) \sim k^{-3}$. Dorogovtsev and Mendes [11] have extended the model to a linear preference function, i.e., instead of a preference function $f_{BA}(i) = k_i$ they use $f_{DM}(i) = k_i + D, D \geq 0$. Dangalchev [6] introduced the two-level network model, by considering the neighbor connectivity as a second “attractiveness” discriminator, $f_{Da}(i) = k_i + c \times \sum_j k_j$, where $c \in [0, 1]$. The global view required at each node attachment renders this algorithm difficult to decentralize.

Preferential Attachment with Accelerated Growth. This model [10] extends the previous model with a separate mechanism to add new links between existing nodes, hence accelerating the growth of the number of links in the network (much like the Internet). This algorithm inherits the difficulties of the basic preferential attachment with respect to decentralization.

Non-linear Preferential Attachment. Krapivsky, Redner, and Leyvraz propose a model [14] that produces scale-free networks as long as $f_{KRL}(i) \sim k_i; k \to \infty$, where $f_{KRL}(i) = k_i^{\gamma}$. This algorithm inherits the difficulties of the basic preferential attachment with respect to decentralization.

Deterministic Static Models. Dangalchev proposed two such networks, the k-control and the k-pyramid, where the latter can be extended to a growth model. Ravasz and Barabási [1] explored hierarchical (fractal-like) networks in an effort to meet both the power-law degree distribution of scale-free networks and the high clustering coefficient of many real networks. Their model starts with a complete q-node graph which is copied $q-1$ times ($q > 2$); the root of the initial graph (selected arbitrarily from the q nodes) is connected with all the leaves at the lowest level; these copy-connect steps can be repeated indefinitely. Such networks have degree distribution $P(k) \sim k^{\frac{\ln q}{\ln(q-1)}}$. Cohen and Havlin [5] use a very simple model which delivers an ultra-small world for $\lambda > 2$; it assumes an origin node (the highest degree site) and connects it to next highest degree sites until the expected number of links is reached. Since loops occur only in the last layer, the clustering coefficient is intuitively high for a large number of nodes. According to [9], some deterministic scale-free networks have a clustering coefficient distribution $C(q) \sim q^{-1}$, where q is the degree. This implies well-connected neighborhoods of small degree nodes. This algorithm seems promising with respect to decentralization, except for the initial phase of complete q-node connectedness.

Fitness-Driven Model. This was introduced by Caldarelli [4] and proves how scale-free networks can be constructed using a power-law fitness function and an attaching rule which is a probability function depending on the fitness of both vertices. Moreover, it shows that even non-scale-free fitness distributions can generate scale-free networks. Recently, the same type of model with infinite mean fitness-distribution was treated in [13]. This power-law network generative algorithm seems the most promising with respect to decentralization.

3 Decentralizable Algorithms for Building Scale-Free Networks

We are interested in analyzing approaches that are feasible to decentralize. We first look at an existing model, presented by Caldarelli in [4], for which we introduce an analytical and empirical verification. We then present an improved model to build scale-free networks, which we also analyze and verify empirically. Our model maintains the property of easy decentralization.

3.1 Caldarelli's Fitness-Driven Model

In this model, power-law networks are generated using a "recipe" that consists of two main ingredients: a fitness density, $\rho(x)$, and a vicinity function, $f(x_i, x_j)$. The fitness density is used to assign each node a fitness value, while the vicinity function is used to decide, based on the fitness values, whether a link should be placed between two nodes.

One instance of this model assumes each node to have a fitness value x_i drawn from a Pareto distribution with density $\rho(x) \sim x^{-\gamma}$. For each node i, a link to another node j is drawn with probability $f(x_i, x_j) = \frac{x_i x_j}{x_M^2}$, where x_M is the maximum fitness value currently in the network.

This model looks very appealing for a self-emerging approach to power-law network generation, since it requires very little information to be globally available. Using epidemic dissemination techniques [8], the maximum fitness value currently existing in the network, x_M, may be easily propagated throughout the network.

According to [4], this approach leads to a network with a power-law degree distribution, that should have the same exponent as the non-truncated Pareto distribution of the fitness values. However, our initial set of experiments show that Caldarelli's approach rarely converges for very large networks. Figure 1 presents the data collected from four different experiments. Each experiment corresponds to a different degree distribution exponent and was repeated for two network sizes: 10,000 nodes and 100,000 nodes. For each experiment we constructed 100 different graphs, each with the same fitness distribution and different random seeds for the neighbor selection. We remark that for a desired power-law degree distribution with exponent $\gamma \neq 3$ and larger values of N ($100K$), the obtained degree distribution exponent does not converge to its desired value. Also, for $\gamma = 4$ and a network of $10K$ nodes, the general approximation function used to determine the degree distribution exponent could not be applied here. We investigated the issue further, by constructing the histogram corresponding to the degree distribution. Figure 5 shows the histograms obtained for each experiment. We remark that the histograms do not coincide with a power-law distribution.

A second set of experiments evaluated how well the algorithm controlled the emerging degree distribution exponent, γ. We increased the control γ in steps of 0.1 and ran the algorithm ten times for each value on a network of 100,000 nodes. We collected the estimated value of the emerging degree distribution exponent and the percentage of isolated nodes (i.e., nodes of degree zero). Both types of results are plotted in Figures 3a and 3b. We note that in the Caldarelli model a large number of nodes remain isolated.

We verified the empirical results by revisiting the assumptions made in [4]. We localized a possible problem with the way the vicinity function is integrated. Intuitively, the problem is that while the X's (fitnesses) are independent r.v. (by assumption), their maximum (x_M) is dependent on all of them, hence can not be pulled out of the integral. To explain this formally, using the Law of Large Numbers we obtain the estimation

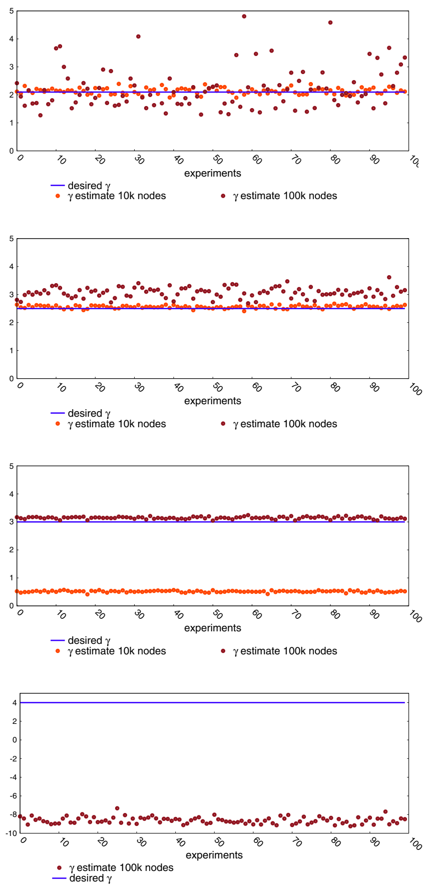


Fig. 1. Caldarelli's model

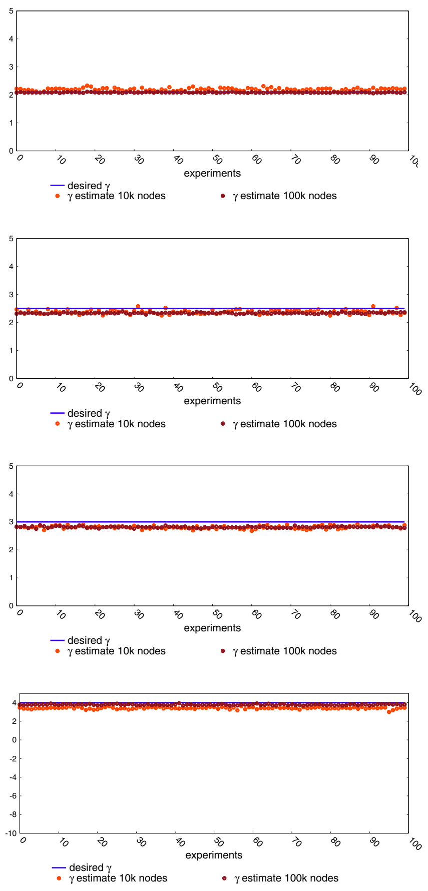
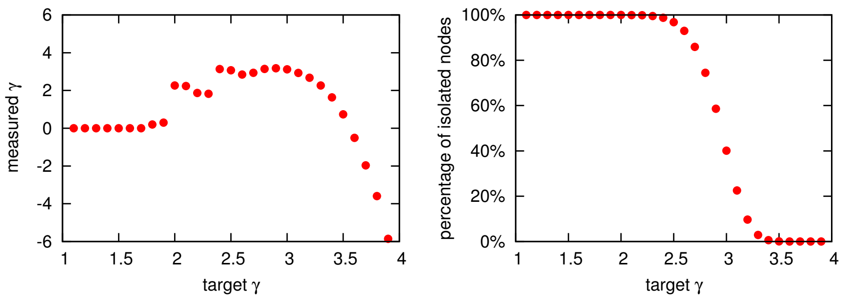


Fig. 2. Improved model

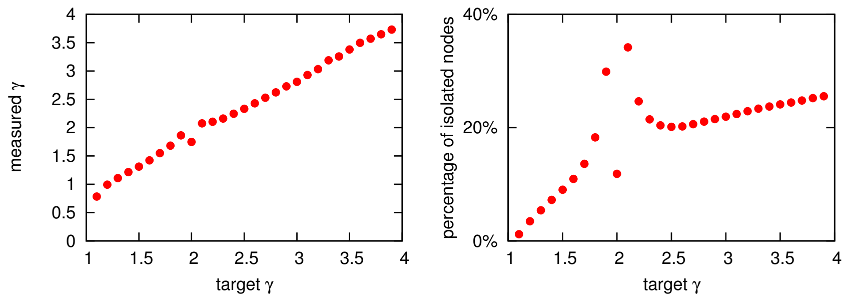
$$V_k = \frac{\text{Number of nodes of degree } k}{n} \approx \frac{1}{n} \frac{\rho\left(P^{-1}(k/n)\right)}{P'\left(P^{-1}(k/n)\right)},$$

where P^{-1} denotes the inverse of P, which is the probability that a node u with fitness x will be linked with any other node v ($P(x) := \mathbb{E}[p(X_u, X_v)|X_u = x]$). This approximation, in conjunction with the assumption $\rho(x) \sim x^{-\gamma}$ would provide the power-law behavior $V_k \sim k^{-\gamma}$, as claimed in [4]. However, this is not the case since x_M is a random variable dependent on all fitnesses (thus, also on X_u and X_v). Hence $P(x)$ is not linear, but is a rather intricate expression of x and an analytical expression for the inverse of P is infeasible. Even worse, if $\rho(x) \sim x^{-\gamma}$, the squared-maximum x_M^2 will grow to infinity at rate $n^{2/(\gamma-1)}$ (by Fisher-Tippet-Gnedenko Theorem), so that $D = (n-1)\mathbb{E}[p(X_1, X_2)]$, will tend to 0 when $\gamma < 3$ (the resulting graph will have a very large fraction of isolated



(a) γ estimates (b) percentage of isolated nodes

Fig. 3. Caldarelli's fitness-driven model



(a) γ estimates (b) percentage of isolated nodes

Fig. 4. Our model

nodes) and will grow to infinity for $n \to \infty$, when $\gamma > 3$; however, as explained before, this last fact is impossible in a power-law graph in which all nodes are linked with the same probability; see equation (3).

3.2 Improved Model

Here we present a novel model for a power-law graph with n nodes. It addresses the limitations found with the Caldarelli model by avoiding certain mathematical pitfalls. Our assumptions differ from the Caldarelli model in that we consider a *truncated* Pareto distribution, with density function $\rho(x) \sim x^{-2}$, for $x \in (l, b_n)$. We emphasize that, unlike Caldarelli, we start with a fixed distribution exponent.

Another considerable difference is the truncation and its upper bound $b_n \to \infty$. The upper bound will depend on the density $\rho(x)$ and on the desired outcome graph degree distribution exponent, denoted by γ in Caldarelli's model. The global variable x_M from Caldarelli's model will be replaced in our model by b_n.

We summarize the mathematical model below:

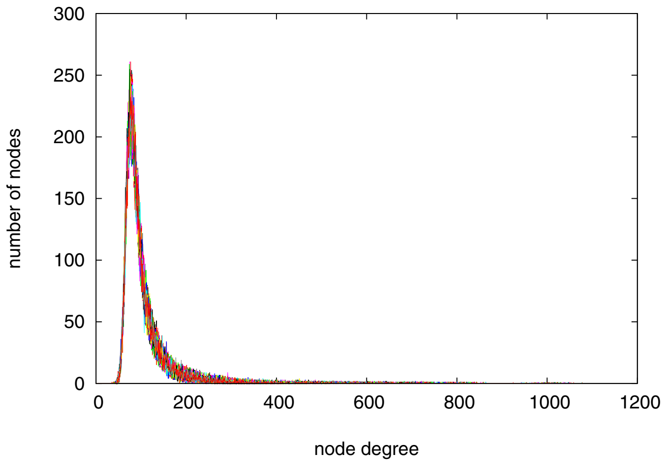


Fig. 5. Caldarelli degree histograms for $\gamma = 4$

(I) The fitnesses $X_1, \dots, X_n$ are drawn from a truncated Pareto distribution, with lower bound $l = 1$, upper bound $b_n \to \infty$ (it will depend on the desired outcome) and density $\rho(x) \sim x^{-2}$, for $x \in (l, b_n)$.

(II) Every pair of nodes (u, v) will be linked with a probability given by $p(X_u, X_v)$, where we define

$$p(x_1, x_2) := \left(\frac{x_1 x_2}{b_n^2} \right)^{\eta}, \tag{1}$$

with $\eta > 0$ depending (again) on the desired outcome.

For appropriate choices of the upper-bound b_n (see details below), performing steps (I) and (II) will result in a power-law graph with index $\gamma := 1 + (1/\eta)$, satisfying (for large $k \leq n$)

$$V_k := \frac{\text{Number of nodes of degree } k}{n} \approx \frac{\gamma - 1}{k^{\gamma}};$$

in other words, if a power-law degree-distribution with exponent $\gamma > 1$ is desired, then one must choose $\eta = (\gamma - 1)^{-1}$ in step (II), while the upper-bound b_n must be chosen according to the following rules:

(i) For $\gamma \in (1, 2)$ we choose $b_n := \left[\left(\frac{\gamma-1}{2-\gamma} \right) n \right]^{\frac{\gamma-1}{\gamma}}$, which gives an expected degree

$$D \approx \left(\frac{\gamma - 1}{2 - \gamma} \right)^{\frac{2}{\gamma}} n^{\frac{2-\gamma}{\gamma}}.$$

(ii) For $\gamma = 2$ we choose $b_n := \sqrt{(n/2)\log(n)}$ and obtain for the expected degree

$$D \approx \frac{\log(n)}{2}.$$

(iii) For $\gamma > 2$ we choose $b_n := \left[\left(\frac{\gamma-1}{\gamma-2}\right) n\right]^{\frac{\gamma-1}{2}}$ which yields

$$D \approx \frac{\gamma-1}{\gamma-2}.$$

In the model described by steps (I) and (II), the probability that a node u, having fitness x, will be linked with any other node v is given by

$$P(x) := \mathbb{E}[p(X_u, X_v)|X_u = x] = \frac{x^\eta}{b_n^{2\eta}} \int_0^\infty z^\eta \rho(z)\, dz. \tag{2}$$

The expected degree of a node of fitness x is $(n-1)P(x)$. The (unconditional) probability of having the edge (u,v) is $\pi_n := \mathbb{E}[p(X_u, X_v)] = \mathbb{E}[P(X_u)]$ and the expected degree of a node is $D := (n-1)\pi_n$. For the choices (i)–(iii), the expected degree of a node of fitness x will be approximately x^η, for large enough n.

At this point, it should be noted that power-law graphs with index $\gamma > 1$ (regardless of how they are generated) in which every two nodes are linked with the same probability π_n, enjoy the following property: If $\gamma > 2$ then the expected degree D must remain bounded as the number of nodes n grows arbitrarily large. When $\gamma = 2$ the expected degree D may grow to infinity with the number of nodes, but no faster than $\log(n)$. Finally, when $\gamma \in (1,2)$ the expected degree D may again grow to infinity with the number of nodes, but no faster than $n^{2-\gamma}$. To justify the above claims, one may express the total expected number N of edges in the graph in two ways: first, since any two nodes are linked with the same probability π_n, the expected number of edges $\mathbb{E}[N]$ is given by $n(n-1)\pi_n/2$. On the other hand, N is half of the sum of all degrees in the graph, hence

$$D = (n-1)\pi_n = \frac{2\mathbb{E}[N]}{n} = \sum_{k=1}^{n-1} k\mathbb{E}[V_k] \le c\sum_{k=1}^{n-1} \frac{1}{k^{\gamma-1}}, \tag{3}$$

where V_k denotes the number of nodes of degree k and $c > 0$ is some finite constant. Since the r.h.s. in (3) is bounded for $\gamma > 2$ and using the estimates

$$\sum_{k=1}^{n-1} \frac{1}{k^{\gamma-1}} \sim \begin{cases} n^{2-\gamma}, & \gamma \in (1,2), \\ \log(n), & \gamma = 2, \end{cases}$$

hence our claims are justified. The conclusion is that the power-law structure of a graph, in which every two nodes interact with the same probability, induces an upper-bound on the magnitude of the expected degree of the nodes. Comparing the expected degree estimates in (i)–(iii) with the maximal rates imposed by (3) reveals that our method maximizes the expected degree when $\gamma \ge 2$.

We also remark that the graph resulted at step (II) will have a certain fraction of isolated nodes which increases with γ. More precisely, for γ close to 1 this fraction will be very small (close to 0), while for very large γ it will approach $1/e \simeq 37\%$; when $\gamma \in (2,3)$ this fraction will stay between $14-22\%$.

The existence of these isolated nodes in our model is a consequence of the upper-bound established by (3) since, in general, by Jensen's Inequality it holds that

$$\mathbb{E}[\deg(v) = 0] \approx \mathbb{E}[\exp(-(n-1)P(X))] \geq \exp[-(n-1)\pi_n] = \exp(-D),$$

so whenever the expected degree D is bounded (recall that this is necessarily the case when $\gamma > 2$) the expected fraction of isolated nodes will be strictly positive.

Similarly to the experiments conducted on the Caldarelli model, we also performed a set of extensive tests on our novel model. Results from the set of 100 experiments are collected in Figure 2. We remark that our model performs in a more stable fashion with respect to the emerging degree distribution exponent. We also notice that our model provides a better convergence with respect to the size of the network.

Next, we analyzed how well our model controlled the emerging degree distribution exponent, γ, by performing the same set of averaging experiments as in Caldarelli's case. All results are collected in Figures 4a and 4b. Our model outperforms Caldarelli's model both in terms of control over the emerging degree distribution exponent, γ, and in terms of the number of isolated nodes. Finally, we notice that the theoretically proven discontinuity at $\gamma = 2$ is illustrated by the experimental results.

In this section, we have presented and experimentally evaluated a novel method for generating connected power-law graphs with any index $\gamma > 1$. In our proposed model, we correct the issues in [4], by considering $t$runcated (bounded) fitness-values and use a deterministic bound b_n instead of a random one. While the lower bound ($l = 1$) is included in the model for technical purposes only, the upper bound b_n is crucial and plays the role of a tuning parameter which allows one to obtain the desired power-law index γ as well as the correct behavior for the expected degree. In fact, the upper-bound b_n is strongly related to the number of edges in the graph by means of the vicinity function defined in (1); namely, the larger the b_n the smaller the number of edges in the graph. In general, increasing the magnitude of b_n will damage the power-law behavior, while for $\gamma > 2$ decreasing b_n will result in an asymptotically empty (still power-law) graph. Therefore, the model is extremely sensitive to the choice of the upper-bound b_n when $\gamma > 2$.

4 Building Power-Law Overlays

4.1 Algorithm

Building power-law overlays in the real world is a nontrivial task. Following the standard methodology, that is, applying the vicinity function on all possible pairs of nodes to decide which edges to place is impractical: It assumes either centralized membership management, or complete membership knowledge by each node. Neither of these scales well with the size of the overlay.

Instead, we explicitly designed a solution in which nodes are not required to traverse the whole network to determine their links. They form links by considering a small partial view of the network. The key point, however, in this

```
Active Thread (on node p)

while true do
    // wait T time units
    S ← r random peers from Cyclon
    foreach q in S do
        v = VicinityFunc(fitness(p), fitness(q))
        with prob v
            Send (q, "INVITE")

function TerminationCondition()
    if degree ≥ expected degree then
        return true
    else
        return false
```

```
Passive Thread (on node q)

while true do
    Receive msg from p
    if msg == "INVITE" then
        if not TerminationCondition()
        then
            Send(p, "ACCEPT")
            AddLink(p)
    else if msg == "ACCEPT" then
        AddLink(p)
    if TerminationCondition() then
        Cease(Active Thread)
        Cease(Cyclon)
```

Fig. 6. The generic gossiping skeleton for building power-law overlays

approach is the termination condition, that is, a criterion that lets a node decide when to stop looking for additional links.

Our method exploits the analytic findings of the previous section. In a nutshell, each node periodically picks a few random other nodes, and feeds the two fitness values into the vicinity function to determine whether to set up a link or not. A node performs this repeatedly until it has satisfied its termination condition, that is, it has established a number of links equal to its expected degree, as computed by the respective formula.

In more detail, our protocol works as follows. Nodes run an instance of Cyclon [15], a peer sampling service that provides each node with a regularly refreshed list of pointers to random other peers, in a fully decentralized manner and at negligible bandwidth cost. Upon being handed a number of random other peers, a node applies the vicinity function and decides if it wants to set up a link with one or more of them. It sends an Invite message to the respective peers, and awaits their responses. Upon receiving an Invite, a node checks if its degree has already reached its expected degree value. If not, it sends back an Accept message as a response to the invitation, and the two nodes establish a link with each other on behalf of the power-law overlay.

When a node's termination condition is met, that is, the number of established links of that node has reached its expected degree, it refrains from further gossiping. That is, it stops considering new neighbors to send Invite messages to, and it responds to other nodes' invitations by a Reject message. Notably, a node also refrains from all Cyclon communication. This is particularly important for letting the network converge fast. By ceasing its Cyclon communication, a node is prompty and conveniently "forgotten" by the Cyclon overlay, letting the latter be populated exclusively by nodes that are still in search of additional links. Thus, Cyclon constitutes a good source of random other peers, as it picks random nodes out of a pool of peers that are willing to form additional links. Even in a network of hundreds of thousands of nodes, when a small number of nodes are left searching for additional links,they quickly discover each other and decide what links to establish.

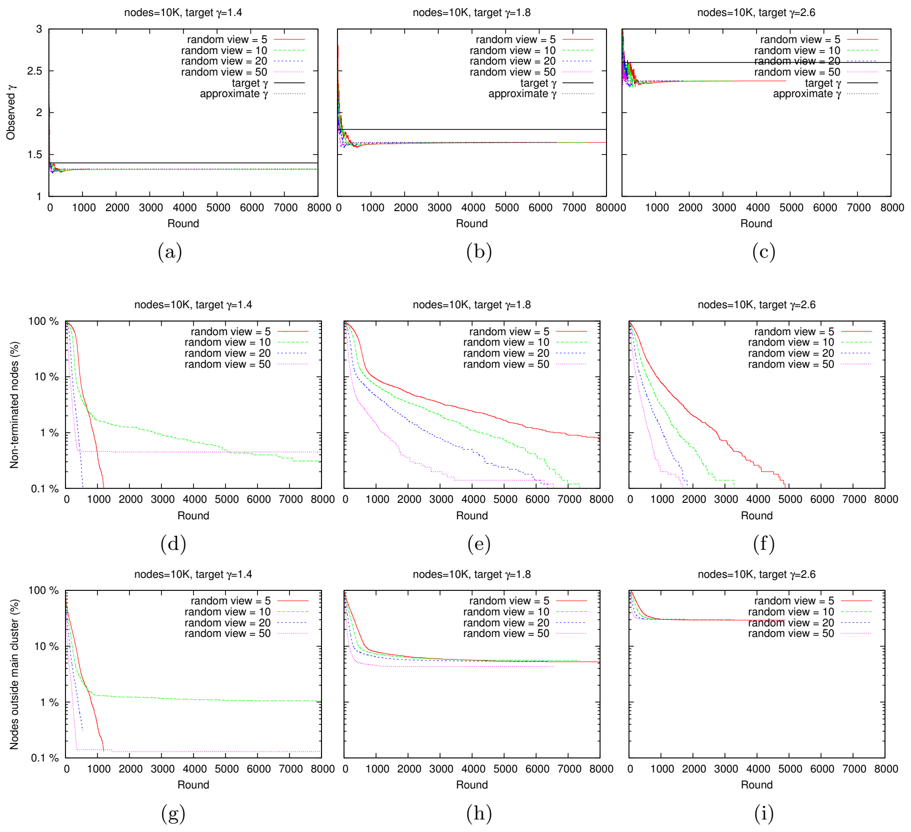


Fig. 7. Statistics collected for different γ values and different random views

Figure 6 shows the programming model of our protocol, without including Cyclon. As most gossiping protocols, it is modelled by two threads. An *active thread*, taking care of all periodic behavior and sending invitations, and a *passive thread* receiving messages (invitations or responses to invitations) and reacting accordingly.

4.2 Evaluation

We implemented our algorithm in PeerNet, a branch of the popular PeerSim simulator written in Java.

We consider a network consisting of a fixed set of N nodes. We assume that communication is reliable. Links are persistent and bidirectional (i.e., when x establishes a link to y, y gets a message to establish a link to x). A node's active thread operates in a periodic manner, and all nodes' periods are the same, yet they are triggered completely asynchronously with respect to each other.

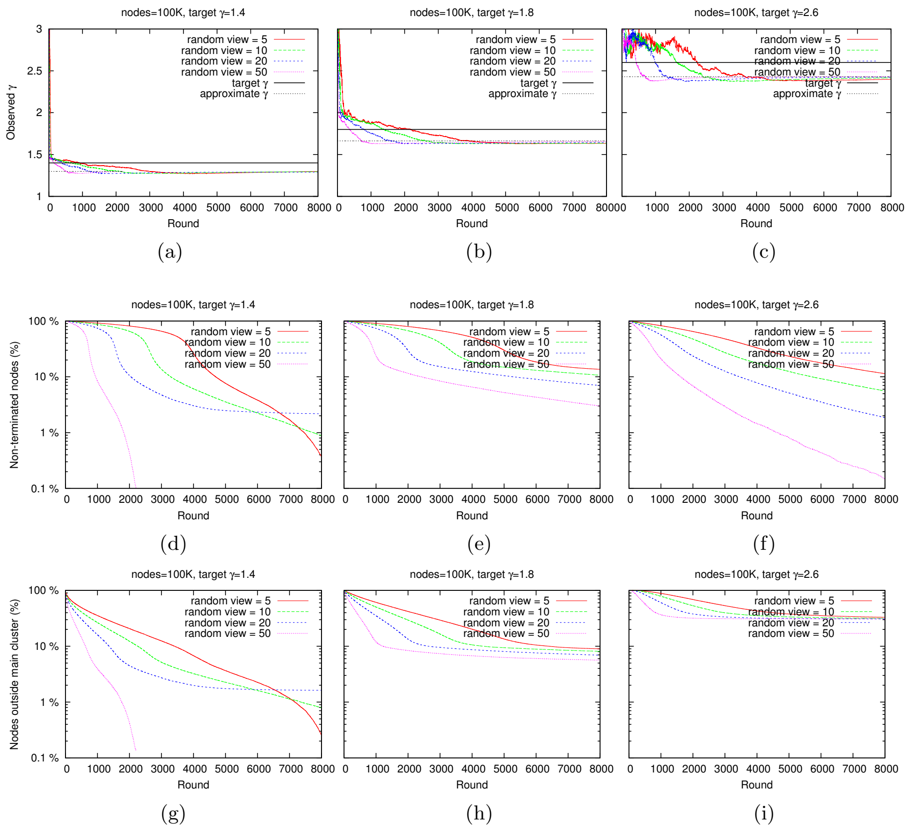


Fig. 8. Statistics collected for different γ values and different random views

The behavior of the protocol depends on three main parameters. First, the *target* γ; second, the *number of nodes* in the network; and third, the *random view size*, that is, the number of random links a node is handed by Cyclon in each round.

Figures 7 and 8 show the results of our experiments for 10,000 and 100,000 nodes, respectively. The first row in each figure (i.e., Figure 7(a-c) and Figure 8(a-c)) shows the observed γ of the emerged overlay, as a function of the number of rounds elapsed since the beginning of the experiment, for three sample values of γ, namely, 1.4, 1.8, and 2.6. The four different lines in each plot correspond to four different random view sizes. In the case of 10K nodes, all four lines converge equally fast to the (approximate) target γ. For the larger network of 100K nodes, checking out more random nodes per round provides some advantage with respect to convergence time.

Note that each graph shows a different target value of γ and the corresponding approximate value. Our formula for a node's expected degree is derived from the mathematical model presented in Section 3.2. However, it is based on the assumption of a large enough number of nodes and therefore we evaluate the error introduced by this approximation. We construct the histogram of all expected degrees (i.e., the expected degree distribution) and we use it to compute an *a*pproximate γ. In each Figure 7a–7c and 8a–8c we compare the target γ, the approximate γ and the γ values of the self-emerging overlays.

The second row of the figures (i.e, Figure 7(d-f) and Figure 8(d-f)) shows the percentage of nodes that have not yet established as many links as their expected degree mandates, and are, therefore, still gossiping in search of new connections. We see that, particularly for the 10K network, the most of the nodes meet their termination criterion within the first few hundred rounds, which means they do not spend any network resources thereafter.

Our formula for a node's expected degree is derived from the mathematical model presented in Section 3.2. However, it is based on the assumption of a large enough number of nodes and therefore we evaluate the error introduced by this approximation. We construct the histogram of all expected degrees, which corresponds to the expected degree distribution and use it to compute an *a*pproximate γ. In each Figure 7a–7c and 8a–8c we compare the target γ, the approximate γ and the γ values of the self-emerging overlays.

Most importantly, though, the graphs of the second row show that the vast majority of the nodes reach their exact expected degree, contributing to the excellent γ approximation observed in the first row graphs.

Finally, the third row graphs (i.e, Figure 7(g-i) and Figure 8(g-i)) show the number of nodes *not* contained in the largest cluster. For low values of γ the largest cluster is massive, containing virtually the whole set of nodes. This is expected, as nodes tend to have high degrees. For higher values of γ, though, which experience long tails of nodes with very low degrees, we see that the resulting overlay is split in many disconnected components. This does not mean that nodes are isolated at an individual level (as confirmed by the graphs of the second rows), but that nodes are connected according to their expected degrees in smaller components. Making sure of connecting all these components in a single connected overlay is the subject of future work.

5 Conclusions

Self-emerging power-law networks are an important area of research. However, algorithms that generate such topologies in a controlled manner are still scarce.

In this work, we investigated existing approaches to sequential power-law graphs generation and selected a model that allowed for straightforward decentralization. We then experimentally identified limitations with the selected model which have been supported by our theoretical findings. We presented a novel model, built on a thorough mathematical support, that addressed the limitations found with previous models. Under the same experimental settings, our results show that our proposed model significantly outperforms the initial one in different convergence aspects.

Next, we implemented a prototype self-emerging power-law network based on our model and gossiping protocols. We show that the theoretical and sequential implementations of the novel model are closely followed in performance by the decentralized prototype. Furthermore, the theoretical bounds are observed throughout an extensive set of experiments. Such a result encourages us to consider the theoretical model already robust with respect to implementation approximations and to continue our research efforts having this model as a foundation. One interesting future research question, identified by our decentralized prototype evaluation, is how to alleviate the the problem of (many) disconnected components.

References

1. Albert, R., Barabási, A.-L.: Emergence of scaling in random networks. Science (1999)
2. Barabasi, A.L., Albert, R.: Emergence of scaling in random networks. Science 286, 509–512 (1999)
3. Batagelj, V., Brandes, U.: Efficient generation of large random networks. Phys. Rev. E 71(3), 036113 (2005)
4. Caldarelli, G., Capocci, A., De Los Rios, P., Muñoz, M.A.: Scale-free networks from varying vertex intrinsic fitness. Phys. Rev. Lett. 89(25), 258702 (2002)
5. Cohen, R., Havlin, S.: Scale-free networks are ultrasmall. Phys. Rev. Lett. 90(5), 058701 (2003)
6. Dangalchev, C.: Generation models for scale-free networks. Physica A: Statistical Mechanics and its Applications 338(3-4), 659–671 (2004)
7. de Solla Price, D.J.: A general theory of bibliometric and other cumulative advantage processes. Journal of the American Society for Information Science 27(5), 292–306 (1976)
8. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic Algorithms for Replicated Database Maintenance, pp. 1–12. ACM Press, New York (1987)
9. Dorogovtsev, S.N., Goltsev, A.V., Mendes, J.F.F.: Pseudofractal scale-free web. Phys. Rev. E 65, 66122 (2002)
10. Dorogovtsev, S.N., Mendes, J.F.F.: Effect of the accelerating growth of communications networks on their structure. Phys. Rev. E 63(2), 025101 (2001)
11. Dorogovtsev, S.N., Mendes, J.F.F., Samukhin, A.N.: Structure of growing networks with preferential linking. Phys. Rev. Lett. 85(21), 4633–4636 (2000)
12. Yule, G.U.: A Mathematical Theory of Evolution Based on the Conclusions of Dr. J. C. Willis, F.R.S. Journal of the Royal Statistical Society 88(3), 433–436 (1925)
13. Flegel, F., Sokolov, I.M.: Canonical fitness model for simple scale-free graphs. Phys. Rev. E 87, 022806 (2013)
14. Krapivsky, P.L., Redner, S., Leyvraz, F.: Connectivity of growing random networks. Phys. Rev. Lett. 85(21), 4629–4632 (2000)
15. Voulgaris, S., Gavidia, D., van Steen, M.: Cyclon: Inexpensive membership management for unstructured p2p overlays. 13(2), 197–217 (June 2005)

Ambient Clouds: Reactive Asynchronous Collections for Mobile Ad Hoc Network Applications

Kevin Pinte, Andoni Lombide Carreton,
Elisa Gonzalez Boix, and Wolfgang De Meuter

Software Languages Lab., Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussel, Belgium
{kpinte,alombide,egonzale,wdmeuter}@vub.ac.be
http://soft.vub.ac.be

Abstract. In MANET applications, a common pattern is to maintain and query time-varying collections of remote objects. Traditional approaches require programmers to manually track the connectivity state of these remote objects and adding or removing them from local collections on a per-object basis. Queries over these collections have to be manually recomputed whenever the collection or its elements change.

The code for maintaining these ad-hoc collections is scattered across the application code and leads to bugs hindering the application development process. In this paper, we propose an object-oriented abstraction called *ambient clouds*: a collection of objects whose contents are implicitly updated when changes occur. Ambient clouds can be queried and composed using *reactive* standard query operators. We show how ambient clouds ease the development of a collaborative peer-to-peer drawing application.

Keywords: collection, mobile ad hoc network, peer-to-peer application, language abstraction.

1 Introduction

The steep increase in popularity of mobile devices has yielded a market for applications running on mobile ad hoc networks (MANETs). MANET applications assume no fixed infrastructure and spontaneously engage in interaction when the devices they run on are in communication range. These applications communicate over wireless networks, for example Wi-Fi Direct or Bluetooth. Using mainstream programming languages such applications are usually conceived as distributed object-oriented applications, coordinating their actions by exchanging objects.

In MANETs, the number of devices participating in an interaction is not known a priori, but it varies as devices join and leave the network as they move about. Typically, applications are interested in communicating only with a specific group of those *remote* objects which are *discovered* at runtime. For example, in a chat application, users can join or leave a chat room at any moment in time. In order to reflect the communication state of the users in the chat room and to allow communication with them, the programmer has to manually maintain a collection of remote objects. In MANETs, the nature of the connection to the devices hosting these objects is volatile. Such a collection of remote objects is continuously fluctuating because of the volatile nature of the connections to the devices hosting these objects.

At the software level, we can identify two ad hoc ways commonly used to implement such collections. A first approach when using a distributed object-oriented programming language is to discover and store the remote objects in a local collection. Once discovered, the programmer must manually iterate over the collection's content and communicate with the stored remote objects by means of remote method invocations or asynchronous message passing. It is the programmer's responsibility to make sure that these objects are still connected by making use of try-catch blocks or other failure handling mechanisms.

A second strategy is to employ an event-based distributed model such as a publish/subscribe architecture. In this case, the collections become groups of objects classified under a topic and potentially filtered on their content using predicates [1]. However, publish/subscribe middlewares abstract the network connectivity between the publisher and subscriber. This obliges programmers to bypass the middleware periodically to detect whether the publishers are still connected and verify that their published objects are still "alive" (i.e., in case of a crash). Additionally, the events signalled by the publish/subscribe middleware must still be manually converted to additions and removals on local collections. This hinders straightforward and efficient querying and composition of such collections.

None of both solutions provides adequate means to create, maintain and query collections of remote objects. This leads the programmer to write boilerplate code that is scattered throughout the actual application code. In this paper, we propose *ambient clouds*: a reactive asynchronous collection abstraction to maintain and query collections of remote objects in MANET applications. Additionally, ambient clouds provide *reactive* standard query operators. Querying or composing ambient clouds using these operators constructs a chain of dependent result collections. The operators observe the collections they were applied to. Changes in either the composition of the constituent elements are implicitly and incrementally reflected throughout the chain of dependent result collections.

In the remainder of this paper, we show the problems that led us to explore ambient clouds (section 2), how ambient clouds tackle these problems at a high level, and how programmers can use ambient clouds to quickly develop mobile applications (section 3). We explain how ambient clouds are implemented (section 4) and how they are used in the collaborative drawing application, called *weScribble* (section 5). Subsequently, we discuss related work (section 6) and, finally, conclude this paper and suggest how we intend to improve ambient clouds in the future (section 7).

2 Problem Statement

In what follows, we detail the issues that can be identified when developing applications that deal with collections of remote objects. We illustrate the problems using a chat application as a running example.

The chat application presents the user with the option to create a chatroom or join an existing one. When the user enters a chatroom, he or she is presented with a list of users that joined this room and are currently in communication range. The application also shows the messages that belong to the chatroom and users can choose to ignore messages from certain other users.

P1. Volatile Collections. MANET applications discover other applications and services running in the environment to interact with them. The applications typically maintain a collection of "currently available" objects in which the application is interested. Since devices hosting MANET applications can appear and disappear at any moment in time, we regard these collections of remote objects as highly *volatile*. Therefore, specifying the contents of the collection *extensionally* (i.e., on a per-element basis) is problematic as the contents of the collections can change at any point in time. With current techniques, the programmer is left to manually synchronise the contents of these volatile collections of remote objects.

To interact with these collections, the programmer typically uses constructs such as indices and iterators. These constructs do not map well to volatile collections of remote objects because the collection changes underneath them. For example, the chat application maintains a list of currently co-located users. As users move about, the composition of this list changes dynamically.

P2. Querying and Composing Collections. A natural operation on collections of objects is to apply operators to compose them with other collections or query them. For example, the chat application applies a filter operation on the list of users to display only those that reside in the chosen chatroom.

Querying and composing volatile collections is not an *atomic action*: collections can grow or shrink several times while a composition or query is being computed. Furthermore, when employing asynchronous method invocation to communicate with remote objects, the results of a query over the elements of a collection may not be available instantaneously.

As the composition of a collection evolves, the initial result of applying an operator diverges from the current state. This means that programmers have to write additional code to ensure results from applying operators remain synchronised with the collection they were applied to. It also implies that compositions of collections resulting from queries over such volatile collections are themselves volatile. This requirement is a serious deviation from the traditional notion of querying collections, where the result of a query does not bear any relationship to the target collection.

P3. Propagating State Changes. When employing a distributed object-oriented model objects are either exchanged "by copy" or "by reference". This results in a collection of respectively local copies of the objects published by a remote device, or remote references to these objects. In the former case, whenever the owner changes an object, the changes should be propagated to all the copies spread across the network. In the latter case, whenever an object is changed, collections containing a reference to this object should be notified of this change. For example, when a user changes his nickname in the chat application, this should be reflected in the buddy list on the applications of other users.

This propagation of state changes can be accomplished through, for example, a publish/subscribe framework. However, changes made to an object can result in it being removed from or added to the result of a certain operation on the collection. In the chat application, a user that decides to move to another chat room changes the "current chat room" property. This change causes some collections to remove this user object from the old chat room, while other collections add the user to the new chat room. Thus, collections containing remote objects should have a means to subscribe to changes on properties of their constituent objects.

3 Ambient Clouds

In this section, we introduce a novel abstraction representing collections of remote objects named *ambient clouds*. Ambient clouds are an object-oriented abstraction that tackles the issues outlined above by combining event-driven interaction, based on a publish/subscribe model, and reactive programming.

We solve problem ***P1*** by allowing developers to specify an ambient cloud of a certain type of objects they want to interact with. The type of the objects acts as an initial filter in order to collect objects of interest. An *event-driven API* signals events whenever an object is added to or removed from the ambient cloud. To address problem ***P2*** we provide *reactive* standard query operators to query and compose ambient clouds. Any computation performed using such an operator is re-executed as the collection changes. The operators automatically handle asynchronous operations. This does not require breaking the abstraction of the collections by looking at their contents at a certain moment in time. To tackle ***P3*** we model object pass-by-copy and pass-by-reference semantics using *reactive objects* and *reactive isolates*. These are object-oriented reactive values of which the state changes over time. These changes are observed and the event-driven API *signals state modification events* to the collections in which they are contained.

3.1 AmbientClouds at Work

We have prototyped ambient clouds in the distributed programming language AmbientTalk [2]. AmbientTalk is an experimental programming language tailored towards developing peer-to-peer applications that operate in MANETs.

We now describe the language constructs provided to create and interact with ambient clouds in the context of the chat application.

Ambient clouds coarsely collect objects related to the application using a *type tag*. Type tags are a lightweight classification mechanism to categorise remote objects explicitly by means of a nominal type. They can best be compared to a topic in publish/subscribe terminology or marker interfaces in Java. Below we create the ambient cloud of all co-located users using the **cloudOf**: construct and passing it the ChatUser type tag as argument.

```
deftype ChatUser;
def users := cloudOf: ChatUser
```

We define an object representing a user of the chat application by means of the **object**: construct. A user has an identifier, a nickname and an attribute containing the name of the selected chatroom. To publish this object in the network as a ChatUser we use the **export**:**as**: construct of AmbientTalk.

```
def me := object: {
  def id := 123;
  def nickname := "Kevin";
  def currentChatroom := "purple" };
export: me as: ChatUser
```

Ambient clouds continuously synchronise their composition as devices move in and out of range. Users that leave communication range are automatically removed from the collection, users that appear in range are added automatically.

Note that users are represented as regular objects, they are published in the network by reference. The ambient cloud of users thus contains remote objects references that can be contacted by sending it asynchronous messages. Later we will show an example of an ambient cloud of remote objects passed by copy.

We further refine the ambient cloud of users using the reactive standard query operators we provide for ambient clouds:

```
def usersInRoom := users.where: { |user|
  equals(user←currentChatroom(), me.currentChatroom) };
def nicknames := usersInRoom.select: { |user| user←nickname() }
```

In this example we first filter the ambient cloud of users, selecting only users that reside in the same chatroom. To this end we use the **where**: method that takes a predicate as argument. After the filter operation we select the nicknames of the users using the **select**: operator, for example to display them in the GUI.

In both query operations, we send an asynchronous message to the remote user object (expressed by the ← operator in AmbientTalk). An asynchronous message send immediately returns a *future*, which is a placeholder for the actual return value of the message. In the above example the equals function waits for the result of the message[1] to be available before computing the equality and in turn immediately returns a future. The **where:** and **select:** operators will automatically wait for futures to resolve with their values and process the results of asynchronous operations as they become available. If an element was removed before an asynchronous operation on that element was completed, the operation is cancelled and the result ignored.

The collections that result from applying operators to an ambient cloud transparently maintain a dependency relation to that ambient cloud. Any time the composition of the ambient cloud is changed or any time one of its constituents is changed, the operation *incrementally* updates the result collection. The result collection is never totally recomputed, rather dependent elements are either added to the result collection, removed from it or updated. Figure 1 shows the dependency chain that corresponds to the above code snippet.

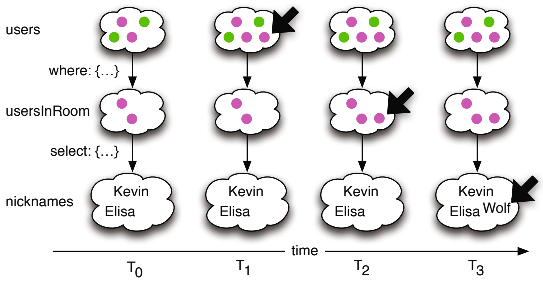


Fig. 1. Dependency tree of the ambient clouds in the chat application

Figure 1 shows the progression of events when the `users` ambient cloud changes. First, in step T_1, a new user is discovered and added to the `users` ambient cloud. In step T_2, the filter operation that generated the `usersInRoom` ambient cloud is then applied to this user object. Since the user has also joined the "purple" chatroom, the user is added to the resulting collection. In step T_3, the dependent `nicknames` ambient cloud is extended with the nickname of the user by applying the select operator to the added user object.

Aside from addition, two more events cause the operators to update their results. Figure 2 depicts the progression of events when a user is removed from the `users` ambient cloud in step T_4 to T_6. The user and nickname are subsequently removed from the dependent result collections. Starting from step T_7 a user switches from the "purple" to the "orange" chatroom. The filter operation is reapplied and causes the user and nickname to be automatically removed from the resulting collections.

[1] By annotating the message send with `Due(t)`, a timeout `t` (in milliseconds) can be specified for the resulting future.

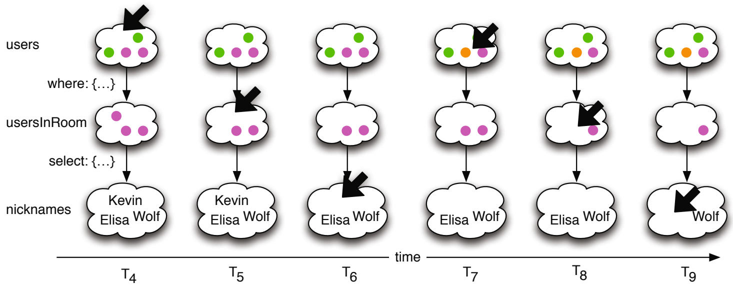


Fig. 2. Removing a user from the ambient cloud and changing the chatroom attribute of a user

3.2 Reactive Standard Query Operators

Our ambient clouds support over 20 operators, including the operators defined by the Standard Query Operation (SQO) [3] API of the Language Integrated Query (LINQ) framework for .NET [4].

The reactive standard query operators are defined as methods on ambient clouds. We also provide a language extension that provides syntactic sugar for writing queries and turns them into first-class language constructs. In the example below we illustrate how to group chat messages with their authors.

```
1  deftype ChatMessage;
2  def messages := cloudOf: ChatMessage
3            groupBy: { |msg| msg.userId }
4            join: usersInRoom
5              on: { |msg, user|
6              equals(msg.userId, user←id()) }
```

In the example above we define the ambient cloud of chat messages starting from line 2. In line 3 we group the ambient cloud based on a `userId` attribute using the **`groupBy:`** operator. This results in an ambient cloud of groups of messages identified by the `userId`. These groups of messages are again ambient clouds which in turn can be queried. Starting from line 4 we associate users with a group of messages using the **`join:on:`** operator. The join is performed based on matching the user's identifier with the identifier attribute in the message. This can be used to ignore the messages of a certain user in the chatroom.

4 Implementation

As mentioned before, ambient clouds are implemented in AmbientTalk and available to the programmer as a library. Ambient clouds are built on top of the Java Collections Framework[2] and combine AmbientTalk's service discovery mechanism

[2] AmbientTalk is entirely implemented in Java and runs on top of the JVM. Java classes and objects can be accessed from within AmbientTalk and vice versa [5].

(based on IP multicasting) with *reactive sets* containing *reactive values*. Note that AmbientTalk is an actor-based language. Execution (e.g., updating ambient clouds) within an actor is sequential and actors communicate by sending asynchronous messages that are processed in sequence by the receiving actor.

4.1 Reactive Asynchronous Collections

The implementation of ambient clouds is based on an reactive asynchronous collection framework that employs local Java collections. Reactive asynchronous collections are conceived as the combination of observable collections with reactive asynchronous operators. Our model consists out of the following collection types:

- **set**: no ordering, no duplicates (similar to Java HashSet)
- **list**: ordering, duplicates allowed (similar to Java ArrayList)
- **sorted set**: sorting, no duplicates (similar to Java TreeSet)

The programmer uses an event-driven API to install event handlers in order to observe a collection. These event handlers are executed when elements are either added or removed from the collection.

```
1 def s := ObservableSet.new();
2 whenever: s extended: { |el| system.println("added " + el) };
3 whenever: s reduced: { |el| system.println("removed " + el) }
```

This example creates a new reactive set and registers two event handlers that write a message to the standard output every time an element is added (line 2) or removed (line 3) from the collection.

Additionally, the collections can be observed for changes to their constituents. When a reactive value is added to a collection, the collection installs an event handler to observe the state of the value. When the state of the reactive value changes, the collection in turn notifies its own observers. The example below shows an event handler that writes a message to the standard output whenever the state of a constituent reactive value changes.

```
whenever: s changed: { |el| system.println("changed " + el) }
```

Ambient clouds are created by connecting the AmbientTalk discovery protocol to reactive sets. Below we show the skeleton code to manually construct the ambient cloud of users in the chat application.

```
def users := ReactiveSet.new();
whenever: ChatUser discovered: { |user|
  users.add(user);
  whenever: user disconnected: { users.remove(user) };
  whenever: user changed: { users.notifyChangeObservers(user) }}
```

In line 1 we first create an empty reactive set. In line 2 we install an event handler to discover objects of type `ChatUser`, using the built-in **`whenever:discovered:`** construct of AmbientTalk. When an object of this type is discovered, it is added to the set in line 3. Two more event handlers are installed in lines 4 and 5. The first event handler is installed using AmbientTalk's built-in **`when:disconnected:`** construct and removes the element from the set when a disconnection occurs. The last event handler is executed when the state of the `user` object is altered and notifies the set of this change.

4.2 Reactive Objects and Isolates

In the implementation of ambient clouds we modelled objects as *reactive values.* A reactive value (also *behavior*) is a value that changes over time [6]. We regard objects as composite values of which the state can be altered over time using field assignment.

Programmers can publish objects in the network either by reference or by copy, as *reactive objects* or as *reactive isolates.* Reactive objects are transferred by reference, remote peers obtain an *observable remote object reference.* An observable remote object reference consists of a proxy object and a reference to the remote object. Reactive collections can install observers on the proxy object which are notified when the state of the object is modified locally. This causes operations applied on the object to be recomputed. Reactive isolates are special objects that have no surrounding lexical scope (i.e., similar to *structs*, but they can have methods defined on them). This way, they can be easily *copied* over the network and *cached* in the ambient clouds of remote peers. The underlying implementation locally observes the state of an isolate and implicitly synchronises state across the copies on different devices. Note that race conditions are prevented by the actor system of AmbientTalk, as explained in section 4. Of course, the programmer should bear in mind that the remote peer may be disconnected and that the copy is temporary out of sync. Any time the state is synchronised, the observers registered by reactive collections are notified.

In subsection 3.1 we showed an example of a user represented by a regular object. The example below shows the creation of a chat message represented by an isolate.

```
def aMessage := isolate: {
  def text := "Hello!"
  def userId := 123 }
```

Reactive objects and isolates are key in the design of ambient clouds since changes to their state triggers re-computation of dependent results.

4.3 Reactive Standard Query Operators

The reactive standard query operators rely heavily on the observable features of the collections to incrementally update their results. They update their results based on three kinds of events: the insertion and removal of elements in a collection and state changes in the elements. When an operator is applied to a collection, the necessary observers are installed and the operator is applied on all elements already contained in the collection. For example, consider the implementation of the **where**: operator below.

```
1  def where: predicate {
2    def result := self.new(); // self refers to the current object
3    self.each: { |e|
4      if: (predicate(e)) then: { result.add(e) } };
5    whenever: self extended: { |e|
6      if: (predicate(e)) then: { result.add(e) } };
7    whenever: self reduced: { |e| result.remove(e) };
8    whenever: self changed: { |e|
9      if: (!predicate(e)) then: { result.remove(e) } };
10   result.parent := self;
11   result };
```

The operator takes a predicate as argument. In line 2 we create a new collection that contains the results of applying the operator. In lines 3 and 4 we first apply the predicate to all elements already contained in the collection and add them if that application succeeds. In lines 5 and 6 we install a handler that applies the predicate to elements added to the collection and add them to the result collection if necessary. In line 7 we install a handler that removes elements from the result as they are removed from the collection. The handler in lines 8 and 9 reapplies the predicate to an element if its state was changed, possibly removing the element from the result if applying the predicate no longer succeeds.

This code clearly shows that the automatic updates of the result are incremental. The execution of the event handlers in lines 5, 7 and 8 concern the addition, removal or update of a single element in the result. In line 10 we register the parent of the result to be the collection over which we applied the operator. In line 11 we finally return the resulting collection.

Note that this code was was simplified for demonstration purposes. It does not deal with the possibility that an asynchronous operation is performed in the predicate application.

5 The weScribble Application

In this section, we validate ambient clouds in the implementation of a collaborative drawing application for the Android platform, called weScribble[3]. The application allows users to dynamically participate in drawing sessions with other people co-located. Aside from mobile Android devices and wireless ad hoc connections between these devices, no other infrastructure is assumed.

At startup, weScribble presents the user with a list of drawing sessions available in the environment with an indication of the amount of people drawing in each session. The user can either join a session or choose to create a new

[3] WeScribble is available from Google Play at `http://bit.ly/eOxpLg`.

one. A drawing session consists of a number of participants and a shared canvas on which they can draw. When a user joins a drawing session, the application synchronises the canvas with the existing participants to fetch the shapes that were already drawn and displays them on the screen. The user can then draw on the canvas of that session and the changes to the canvas are propagated to the other participants of the session whose canvas is updated accordingly. If a user temporarily disconnects from the network, he or she can keep drawing. Upon reconnection, those changes are synchronised with the other users in the session.

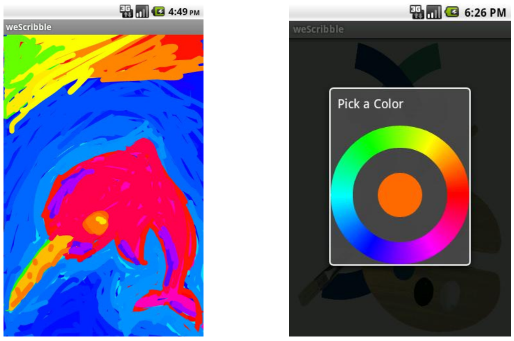


Fig. 3. The weScribble Android application

5.1 Ambient Clouds in weScribble

In this section, we illustrate the use of ambient clouds in the implementation of weScribble. weScribble uses ambient clouds for two different purposes: A drawing session consists of an ambient cloud that contains the shapes drawn by users. The users themselves are also contained in an ambient cloud.

First we collect the users that participate in our drawing session and extract their user names:

```
deftype Painter;
def painters := cloudOf: Painter
                where: {|p| equals(p←session, currentSession)};
names := painters.select: { |p| p←name }
```

We display the names of all discovered users in the GUI. Using the event-driven API we install two event handlers to show and hide names as they are added to or removed from the ambient cloud.

```
names.each: { |n| GUI.showInList(n) };
whenever: names extended: { |n| GUI.showInList(n) };
whenever: names reduced:  { |n| GUI.removeFromList(n) }
```

Users can choose to ignore shapes from other users by enabling a toggle in the GUI. We continue by defining an observable set containing the users that we choose to ignore.

```
def ignoredPainters := ObservableSet.new();
def ignore(painter) { ignoredPainters.add(painter) }
```

We now obtain the ambient cloud of shapes (represented by isolates) to display by joining the ambient cloud of all shapes with the difference of the ambient cloud of users and the users to ignore.

```
deftype Shapes;
def shapes := cloudOf: Shapes
              join: (painters.except: ignoredPainters)
                on: { |shape, painter| s.painterId == p.id }
```

Note here that adding users to the `ignoredPainters` set causes its shapes to disappear from the `shapes` ambient cloud. Finally we draw the relevant shapes and install event handlers to draw and hide shapes as the ambient cloud is updated.

```
shapes.each: { |s| GUI.draw(s) };
whenever: shapes extended: { |s| GUI.drawShape(s) };
whenever: shapes reduced: { |s| GUI.removeShape(s) };
whenever: shapes changed: { |s| GUI.redrawShape(s) };
```

5.2 Discussion

In the implementation of weScribble ambient clouds tackle the issues outlined in section 2 as follows.

- ***P1:*** Ambient clouds automatically maintain collections of co-located users and the shapes they created. The programmer is relieved from manually synchronising the contents of these collections with the network situation.
- ***P2:*** We used reactive standard query operators over ambient clouds to filter users based on the drawing session. We also associate users with their shapes, filtering out shapes of users we wish to ignore without having to manually update these results when users appear or disappear.
- ***P3:*** When users change the colour of their shapes, these changes are automatically propagated to the applications of other users by means of reactive isolates. If users change their nickname this is automatically reflected in the user lists of the other users using reactive objects.

6 Related Work

The problems around group abstractions for mobile applications is well established. However, most of the research focuses on group communication which is not the focus of this work. In this section, we discuss related work that focuses on organising remote objects in intermittently connected peer-to-peer applications.

Distributed Asynchronous Collections (DACs) [7] were originally devised as a way to marry publish/subscribe systems to traditional collection frameworks. They allow developers to subscribe to additions and removals that occur in the collections. However, DACs offer no support for tracking the connectivity of publishers that publish objects, so the programmer still has to track this manually.

Tuple spaces [8] allow distributed parties to publish tuples to a conceptually shared memory. LIME [9] even allows distinct tuple spaces to merge if users are close to each other and allows programmers to define reactions on the appearance or disappearance of tuples. Tuple spaces do not support the direct modification of tuples: tuples have to be removed first and new versions reinserted later. This requires application developers to write additional code to watch these remove/insert event pairs individually. Additionally, there is no support for creating a data structure from a tuple space, which forces programmers to update the derived collections manually.

Ambient references [10] allow discovering and communicating with homogenous groups of references to objects in the environment that change over time. This abstraction takes care of monitoring any disconnections and reconnections in the environment and even allows developers to take a snapshot of the current state. Ambient references do not allow programmers to react on objects, joining, leaving or being modified in the group. Additionally, they can not be composed, nor queried.

M2MI [11] introduces *handles* that denote a dynamic group of remote Java objects of the same interface and *omnihandles* that refer to all proximate objects of a certain interface. However, is not possible to react to state changes in the objects referred to by an omnihandle, nor is there support for querying.

Microsoft's Reactive Extensions for .NET [12] allows processing events by modelling them as streams of values on which standard query operators are defined. The difference with our work is that we define these standard query operators on object-oriented collections instead of event streams.

Reactive programming [6] is a paradigm that represents a program as a data flow graph based on the notion of *time-varying values*, which form nodes in the graph. If an operation is applied to a time-varying value, the operation is inserted as a node in the graph with a dependency edge to the time-varying value. When a time-varying value changes, dependent computations are automatically re-executed by propagating the change through the graph. We discuss the relation between reactive programming and our work in section 7.

7 Conclusion and Future Work

In this paper, we introduced ambient clouds: an object-oriented abstraction that automatically maintains collections of remote objects. These ambient clouds support reactive standard query operators that implicitly update their results as changes in the underlying ambient clouds occur. Together they relieve the programmer from re-implementing the common patterns when dealing with **(1)** volatile collections of remote objects, **(2)** querying and composing these collections and **(3)** propagating state changes to derived collections.

We have implemented a MANET application called weScribble using ambient clouds to illustrate how ambient clouds circumvent these problems in a fully functional collaborative drawing application.

As possible avenues for future work, we wish to further extend our reactive collection framework with other collection types such as a hash map or tree structures. Additionally, unlike our model, in reactive programming languages, there are no special reactive operations. Every operation that depends on a reactive value is implicitly lifted to the reactive level. Currently, on our reactive asynchronous collections, special query operators are used to process changes. We are integrating our collections into a reactive programming language to reduce manual lifting.

References

1. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. ACM Computing Survey 35, 114–131 (2003)
2. Van Cutsem, T., Mostinckx, S., Gonzalez Boix, E., Dedecker, J., De Meuter, W.: AmbientTalk: object-oriented event-driven programming in mobile ad hoc networks. In: SCCC 2007, pp. 3–12. IEEE Computer Society (2007)
3. Microsoft Corporation: The .NET standard query operators. Technical Specification (2006)
4. Meijer, E., Beckman, B., Bierman, G.: LINQ: reconciling object, relations and XML in the .NET framework, 706–706 (2006)
5. Cutsem, T.V., Mostinckx, S., Meuter, W.D.: Linguistic symbiosis between event loop actors and threads. Computer Languages, Systems & Structures 35, 80–98 (2009)
6. Bainomugisha, E., Lombide Carreton, A., Van Cutsem, T., Mostinckx, S., De Meuter, W.: A survey on reactive programming. ACM Computing Surveys (2012) (to appear)
7. Eugster, P.T., Guerraoui, R., Sventek, J.: Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 252–276. Springer, Heidelberg (2000)
8. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems 7, 80–112 (1985)
9. Murphy, A., Picco, G., Roman, G.C.: LIME: A middleware for physical and logical mobility. In: Proceedings of the 21st International Conference on Distributed Computing Systems, pp. 524–536. IEEE Computer Society (2001)
10. Van Cutsem, T., Dedecker, J., Mostinckx, S., Gonzalez Boix, E., D'Hondt, T., De Meuter, W.: Ambient references: addressing objects in mobile networks. In: OOPSLA 2006, pp. 986–997. ACM Press (2006)
11. Kaminsky, A., Bischof, H.P.: Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In: OOPSLA 2002, pp. 72–73. ACM Press (2002)
12. Microsoft Corporation: The reactive extensions for .NET (2013), `http://msdn.microsoft.com/en-us/data/gg577609`

Bandwidth Prediction in the Face of Asymmetry

Sven Schober[1], Stefan Brenner[2], Rüdiger Kapitza[2], and Franz J. Hauck[1]

[1] Institute of Distributed Systems, University of Ulm, Germany
{sven.schober,franz.hauck}@uni-ulm.de
[2] TU Braunschweig, Germany
{brenner,rrkapitz}@ibr.cs.tu-bs.de

Abstract. An increasing number of networked applications, like video conference and video-on-demand, benefit from knowledge about Internet path measures like available bandwidth. Server selection and placement of infrastructure nodes based on accurate information about network conditions help to improve the quality-of-service of these systems. Acquiring this knowledge usually requires fully-meshed ad-hoc measurements. These, however, introduce a large overhead and a possible delay in communication establishment. Thus, prediction-based approaches like Sequoia have been proposed, which treat path properties as a semimetric and embed them onto trees, leveraging labelling schemes to predict distances between hosts not measured before. In this paper, we identify asymmetry as a cause of serious distortion in these systems causing inaccurate prediction. We study the impact of asymmetric network conditions on the accuracy of existing tree-embedding approaches, and present *direction-aware embedding*, a novel scheme that separates upstream from downstream properties of hosts and significantly improves the prediction accuracy for highly asymmetric datasets. This is achieved by embedding nodes for each direction separately and constraining the distance calculation to inversely labelled nodes. We evaluate the effectiveness and trade-offs of our approach using synthetic as well as real-world datasets.

Keywords: Asymmetric bandwidth prediction, tree embedding.

1 Introduction

The performance of distributed multimedia applications largely depends on path properties like latency, packet-loss and bandwidth. A priori knowledge of these helps to improve the user-perceived quality of service by the adaption of application-specific variation points, like video resolution and codec, or modifying the communication structure by placing infrastructure nodes at beneficial locations in the network.

A naïve approach for acquiring path-property knowledge is to perform ad-hoc measurements. However, this has several disadvantages. First, it creates a huge overhead, as the required measurements grow quadratically with the node count. Second, it introduces delay as measurements cannot be performed in parallel, but have to be performed sequentially to avoid interferences.

Therefore, multiple prediction-based approaches have been proposed [1,4,5,8,10,16,18,21,25] which reduce the amount of required measurements by embedding hosts into a metric space and predicting unknown inter-host path properties using a distance function on the host's coordinates. Although predicting latency has been well studied in the literature, fewer approaches exist which are able to predict path bandwidth [8,10,16].[1] Tree metric spaces also have been proposed as targets for embedding network path metrics [1,21,25].

Most of these approaches require a symmetric distribution of the metric under consideration. However, this assumption is often violated [11,19], the most obvious example being the last-mile link of an endhost, often implemented with access technologies like ADSL[2] and DOCSIS[3] (Cable). Consequently, matrix factorization has been proposed to cope with asymmetry but needs clustering of nodes to reduce the rank of the distance matrices. The model proposed by Beaumont et al. [2] is also able to cope with asymmetry, but assumes the access link to determine the bottleneck bandwidth of each path. Xing et al. [27] propose embedding bandwidth in a set of ultrametric spaces, but their system is based on landmarks, and thus, not fully decentralizable.

To further motivate the need to cope with path asymmetry consider a *peer assisted streaming* system [14]. These systems try to minimize server load by selecting "close-by" peers as streaming sources. Figure 1a depicts a session between a server s and two clients c_1 and c_2 (active sources are rendered bold). Consider a scenario where c_1 has a highly asymmetric access link of 50 Mbit/s downstream and 2.5 Mbit/s upstream, common values for cable Internet, and c_2 has a symmetric though lower bandwidth link of 16 Mbit/s in each direction. We further assume for simplicity, that there is no bottleneck on the path between s and the clients. A new client c_3 joining this session has three choices s, c_1 and c_2 as its streaming source. Assuming s is already highly loaded, joining clients are forced to select other sources for streaming. In a system leveraging a symmetric bandwidth prediction component the choice will be node c_1 as this system averages up- and downstream in its prediction (cf. Figure 1b black arrows). However, c_2 obviously would be a better choice. Moreover, assuming the stream consumes bandwidth greater than 2.5 Mbit/s, selecting c_2 would lead to unacceptable performance of the streaming system. An asymmetry-aware streaming system would be able to select a peer based on its upstream bandwidth and in the presented case, select c_2 as its stream source (cf. Figure 1c).

In this paper we extensively study the impact of asymmetry on existing prediction approaches and present a technique, *direction-aware embedding* (DAE), effectively mitigating the negative effects of asymmetry on the prediction accuracy. By separating the upstream and downstream path characteristics and

[1] Note that this dicussion is independent of the type of bandwidth under consideration: capacity or available bandwidth.
[2] ITU-T G.992.5.
[3] ITU-T J.222.

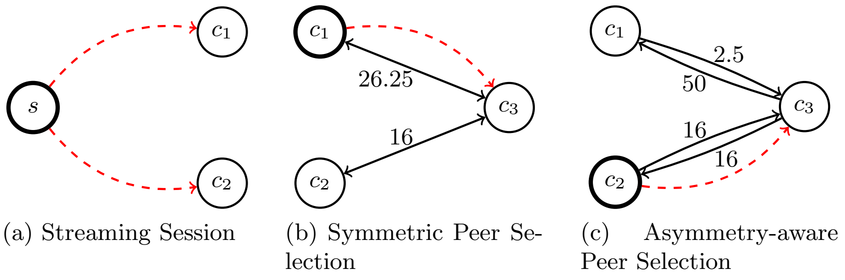


Fig. 1. Peer Selection Alternatives c_1 and c_2 in a Peer-assisted Streaming Scenario

embedding direction-labelled nodes onto prediction trees, we are able to significantly improve the prediction performance of previously presented tree-embedding schemes.

The rest of this paper is organized as follows. First, we present existing tree-embedding schemes in Section 2. Next, we present our approach in Section 3, which tackles this problem by applying our direction-aware embedding scheme. In Section 4 we evaluate the negative impact asymmetry has on the prediction accuracy of previous systems and the effectiveness of our novel approach. Finally, we discuss related work in Section 5 and conclude in the last section.

2 Background

Before we can present our contribution we first discuss the basic concepts of tree metrics, distance labeling and how both are applied in the two existing systems *Sequoia* [21] and its improved version by Song et al. [25].

In the scope of this paper, a *host* refers to a computer or Internet host, while a *node* denotes the respective entry on a tree. We also differentiate estimation and prediction. While we use the term *estimation* for sophisticated measurement techniques like pathChirp [22], the term *prediction* refers to reading tree distances off an existing prediction tree without further measurements.

2.1 Tree Metrics and Bandwidth

Following Ramasubramanian et al. [21] we understand a set of pairwise bandwidth measurements M as a tree metric, if there exists a tree T representing the measurements as distances between tree nodes with non-negative edge-weights such that $M \subseteq T$ and $d_M(a,b) = d_T(a,b)$ for all $a, b \in M$. While $d_M(a,b)$ represents the measured bandwidth from host a to b, $d_T(a,b)$ denotes the predicted bandwidth and accordingly the distance between the nodes on the tree.

In analogy to the triangle inequality, a necessary condition for a metric to be embeddable on a tree is the four-points condition (4PC): $d(w,y) + d(x,z) = d(w,z) + d(x,y)$ for the distances between four nodes w, x, y, z ordered by renaming such that $d(w,x) + d(y,z) \leq d(w,y) + d(x,z) \leq d(w,z) + d(x,y)$. Said in words, the two greater sums of distances between the nodes have to be equal, to embed them on a tree without distortion.

The key observation the authors of Sequoia make is that distance matrices as perceived in the Internet display a certain *treeness*. More formally, they satisfy a relaxed form of the 4PC, the so called ϵ-four-points condition (ϵ-4PC) for relatively small ϵ-values:

$$d(w,z) + d(x,y) \leq d(w,y) + d(x,z) + 2\epsilon \cdot min\{d(w,x), d(y,z)\} \tag{1}$$

Here, $\epsilon \in [0,1]$ characterizes how close a given metric is to a tree-metric ($\epsilon = 0$ is absolute treeness). Ramasubramanian et al. depict that bandwidth measurements on PlanetLab [3] show a high degree of treeness, with 80% of ϵ-values being less than 0.2.

The basic assumption of previous embedding approaches is that the source metric is at least a *semimetric*, i.e. it is assumed to be symmetric. However, in real-world network topologies this assumption is often violated [19,11]. Following Xing et al. [27], we define an asymmetry coefficient ζ_{ab}, which denotes the bandwidth asymmetry between two nodes a and b as follows:

$$\zeta_{ab} = \frac{|(d(a,b) - d(b,a)|}{d(a,b) + d(b,a)} \tag{2}$$

2.2 Sequoia

Ramasubramanian et al. present *Sequoia* [21], a system which is able to predict latency and bandwidth between its participants. It reduces the total number of measurements by embedding hosts on an edge-weighted tree, called *prediction tree*. Then, distances between arbitrary hosts can be read off that tree, by summing up the weight of the shortest path between them.

As bandwidth is not an additive but a concave metric,[4] a transformation has to be applied to the measured values. This is done for Sequoia in a linear fashion by subtracting the measured values from a large constant.

The prediction tree is constructed using an embedding procedure that selects two nodes (anchor and lever) from the existing tree by maximizing the so called *Gromov product* individually for each joining host. Concrete, the Gromov product $(b|c)_a$ allows the calculation of the intersection points of the incircle with the edges of a triangle Δ_{abc}:

$$(b|c)_a = \frac{1}{2}(d(b,a) + d(c,a) - d(b,c)) \tag{3}$$

In the context of prediction trees, b depicts the anchor, a the lever or "base node" and c the host to be embedded. Note, that the distances involving c have

[4] Path-bandwidth is defined by the weakest link of a path, while path-latency is the sum of individual link latency.

to be measured, whereas $d(b, a)$ can be read off the tree. Once anchor and base have been found, a virtual node t_c is inserted along the path p_{ba} at distance $(b|c)_a$ from the base node. Figure 2a displays a sample prediction tree after four node insertions.

A special *distance-labelling* scheme [20] for the nodes of a prediction tree allows the encoding of distances between any two hosts of an edge-weighted tree inside their respective labels. This enables the computation of distances between two hosts without knowledge of the complete prediction tree. In the context of this work, we follow the labelling scheme used by Song et al., where labels consist of a list of 3-tuples representing the anchor hierarchy and the distances between the nodes:

$$l_v = (a_0, d(a_0, t_v), d(t_v, v)), ... \tag{4}$$

Following this scheme we get the following distance labels for the nodes in the example tree depicted in Figure 2a: $l_a = (a, 0, 0)$, $l_b = (a, 0, 41)(b, 0, 0)$, $l_c = (a, 0, 41)(b, 3, 4)(c, 0, 0)$ and $l_d = (a, 0, 41)(b, 16, 13)(d, 0, 0)$. Node c for example is embedded on the tree using b as its anchor and its virtual node being at offset 3 on the path from b to a. Then, c is connected to its virtual node at distance 4.

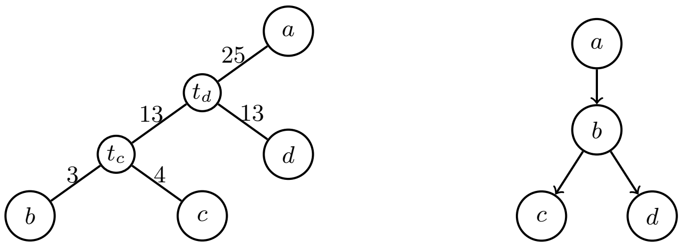
The first node of a tree can easily be identified by a label consisting of only one 3-tuple. For the other nodes the last tuple represents the node itself (with offset and distance $= 0$ because there is no distance to itself) while the other tuples represent distances to its anchor, its anchor's anchor and so forth. Given their labels, the tree distance between two hosts can be calculated easily [24]. Basically, considering $d_T(c, d)$ we get: $16 - 3 + 4 + 13 = 30$ for example.

In order to reduce the amount of conducted measurements the authors introduce an abstraction called *anchor tree*, wherein they capture the anchor relationships of joining nodes (which are also manifested in the distance labels). Figure 2b shows the anchor tree corresponding to the presented prediction tree. Since anchor trees must contain distance labels, an anchor tree can be transformed to a prediction tree and vice versa. Instead of searching the complete prediction tree for a node maximizing the Gromov product, the anchor search is guided by the anchor tree starting with the lever. A greedy search algorithm stops once no more progress can be made. Note, that this can result in a sub-optimal anchor selection, as only a subset of anchor candidates is considered depending on the algorithm's chosen path through the anchor tree.

The embedding order of hosts influences the overall distortion e_r, defined as the mean value of the relative prediction error for each path on the tree. The relative prediction error itself is defined as follows:

$$e_p = \frac{|d_T(a, b) - d_M(a, b)|}{d_M(a, b)} \tag{5}$$

The problem of prediction accuracy being influenced by the order of embedding the hosts has been addressed by the authors of Sequoia by constructing multiple prediction trees in parallel using different insertion orders. This way, when



(a) Prediction Tree (b) Anchor Tree

Fig. 2. Sample Instances of Core Sequoia Concepts

predicting distances they are able to remove outliers by choosing the median of distances predicted by the constructed trees.

2.3 Optimizations

Song et al. [25] improve on Sequoia basically in two ways: First, they decentralize the tree construction algorithm. To achieve this, they discard the idea of using multiple trees and use the structure of a single anchor tree to form an overlay network between the participating nodes. Afterwards, decentralization of the embedding algorithm is done by allowing the use of a random base node and starting the anchor search at the chosen base node. Second, they improve tolerance for datasets with less than perfect treeness which is true for most real-world datasets. To this end they modify the anchor selection algorithm to minimize the overall prediction error instead of maximizing the Gromov product, extend the search space on the anchor tree and modify the transformation to a rational function to avoid negative values when prediction values exceed the transformation constant. Furthermore, Song et al. propose an anchor search optimization which optimizes the chosen base and anchor by another pass of error minimization based on measurements that have already been taken.

3 Direction-Aware Embedding (DAE)

Obviously, Sequoia and Song's approach both are not designed to cope with asymmetric links. As these are a common phenomenon in the Internet, we identified them as a challenge for prediction accuracy. In order to cope with asymmetry, the key idea of our approach is to reflect the varying bandwidth properties of an asymmetric host by embedding it twice on the prediction tree. Once for its upstream and once for the downstream properties (cf. Figure 3). Distance computation then is only performed and valid between inversely directed representative nodes of two hosts.

Algorithm 1. DAE Embedding Procedure

```
Input: $j$: joiner, $d$: direction
Output: $l_j$: distance-label of joiner
 1: $b \leftarrow$ random node with direction $-d$
 2: $a \leftarrow null$, $e_{min} \leftarrow \infty$, $v_{base} \leftarrow b$
 3: while $a \neq v_{base}$ do
 4:     $C \leftarrow$ getCandidates$(v_{base}, d)$
 5:     for all $c \leftarrow C$ do
 6:         $l_{a-tmp} \leftarrow$ position$(j, c, b)$
 7:         $e_p \leftarrow$ relativeError$(l_{a-tmp})$
 8:         if $e_p < e_{min}$ then
 9:             $e_{min} \leftarrow e_p$, $a \leftarrow c$
10:         end if
11:     end for
12:     $v_{base} \leftarrow a$
13: end while
14: return $l_{a-tmp}$
```

Algorithm 2. DAE Positioning

```
Input: $j$: joiner, $a$: anchor, $b$: base
Output: $l_j$: new distance-label
 1: $\delta_o \leftarrow d_T(a, b) - (a|j)_b$
 2: $\delta_d \leftarrow d(j, b) - (a|j)_b$
 3: return appendLabel$(a, \delta_o, \delta_d)$
```

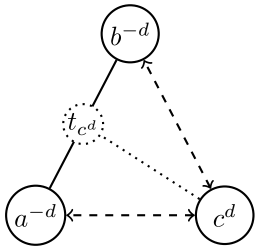


Fig. 3. Embedding Visualization

In order to embed a node on a DAE prediction tree, we modified the tree construction algorithm by first assigning a direction-label $d \in \{\uparrow, \downarrow\}$ to the host name. The embedding procedure itself then is executed twice in a similar way as Sequoia for upstream and downstream properties individually. However, in contrast to the existing approaches, the chosen base node and the anchor both have to be of inverse direction $-d$ to the joining node (cf. Figure 3 and Algorithm 1). Hence, a single host is represented by two nodes on the tree making use of a total of four different reference nodes (an anchor and a base node for each direction). For the upstream base and anchor, the corresponding upstream bandwidth starting from the joiner is measured and used as a distance for the embedding algorithm. The same is done for the opposite direction using the appropriate downstream measurements from the base and anchor to the joiner. This may result in the two nodes of a single host residing in totally different sections of the tree.

Our embedding procedure is depicted in Algorithm 1. Here, j denotes a joining node, a an anchor candidate, b the base node and d a direction-label. We choose a base node randomly and select an anchor using the anchor tree. Due to the inverse direction constraint, the anchor tree is a bipartite graph consisting of alternating up- and downstream nodes. Consequently, inappropriate nodes (wrong direction) have to be skipped during the anchor search. This is done by getCandidates (Line 4) which only selects valid nodes of the two-hop neighborhood[5] of the given v_{base}-node which is the starting point for each anchor search iteration. Following Song's approach, we choose a node minimizing the

[5] Two hops are needed because of the alternating directions on the anchor tree.

relative prediction error e_p (Lines 5-11), as defined in Equation 5. When no anchor can be found for a given base node because of non-available measurements, we choose another base node as a fallback. We also use the optimization procedure based on already measured links as proposed by Song et al. Note that nodes of the anchor tree might also have multiple children alike Song's approach.

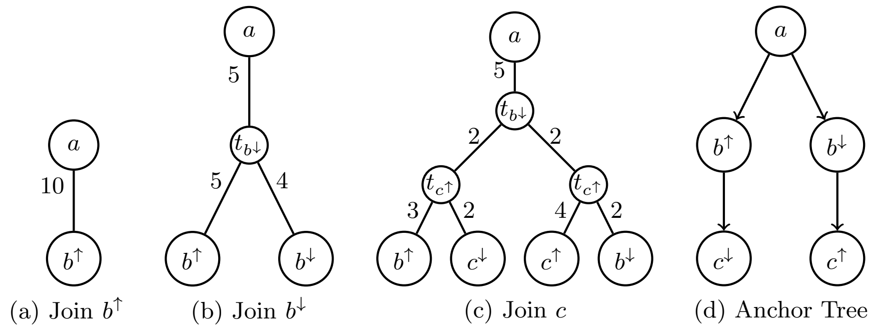


Fig. 4. DAE Example Tree Construction

Positioning of hosts on the tree is done similar to Sequoia (c.f. Algorithm 2). The Gromov product is used to calculate the position of a node with respect to its anchor. The distances needed for this calculation are measured $(d_M(j,b), d_M(j,a))$ and read off the tree $(d_T(a,b))$.[6] Then, the according offset δ_o and distance δ_d values are calculated and the node is embedded on the tree. This is done by copying the distance label of the anchor and altering it, appending a new 3-tuple containing offset and distance with respect to the anchor. A step-wise sample prediction tree construction using our DAE approach is depicted in Figure 4 with the corresponding anchor tree presented in Figure 4d.

Special care has to be taken when embedding the first node, as it is represented only by a single node. Since it bears no direction-label, it can act as a base node for both up- and downstream representative nodes. Also the second node, which will be one direction-labelled representative of the second host, has to be treated specially. This is due to the fact that there is no node in the tree, which could act as anchor, as the first cannot be base and anchor at the same time. Thus, the weight of the edge to the first node simply represents the measured distance in the corresponding direction (c.f. Figure 4a). Afterwards, the third node (the second representative of the second host) is embedded on the edge between the two existing nodes at the measured distance (c.f. Figure 4b). For the second host one can decide whether to embed its upstream or its downstream node first. We embed the node with the longer distance to the first node before the other one, as this avoids negative distances δ_d.

[6] Note that we also use the rational transformation introduced by Song et al. in order to transform bandwidth measurements.

In order to predict the path bandwidth from host a to b in our scheme, we first assign the implied direction-labels, $a^{\uparrow}$ and $b^{\downarrow}$ and then calculate the tree distance. Note that the tree distance between two nodes of the same direction (e.g. $d(a^{\uparrow}, b^{\uparrow})$) is meaningless as is the distance between the two representatives of a single host (e.g. $d(a^{\uparrow}, a^{\downarrow})$).

4 Evaluation

In the following, the benefit and properties of DAE especially in presence of asymmetric links are shown and evaluated. First, we describe our methodology, topologies and algorithm configuration. Then, we proceed to quantify the improvement of prediction accuracy achievable by DAE. We used two different topologies described below. We embed these using the three embedding approaches Sequoia, its enhancement by Song et al. (denoted as "Song" in the figures) and our DAE algorithm. Since there were no publicly available implementations of Sequoia and Song's approach, we implemented them based on the respective papers [21,25][7].

4.1 Methodology

When a dataset has been embedded, we calculate the relative embedding error for the complete prediction tree as the average value of individual link prediction errors. In this evaluation, the relative prediction error e_p of the path between two nodes a and b is calculated according to Equation 5. Note that the error calculation is also direction-aware.

It is also vital to define how the amount of executed measurements is counted for this evaluation. Since Sequoia and Song's approach both take the average value of the bidirectional measurements $a \rightarrow b$ and $b \rightarrow a$ between the hosts a and b, we count two measurements for each measured link. For DAE it is possible to make use of a measurement only for one direction of a link. Hence, counting both link directions individually is also appropriate for DAE.

4.2 Topologies

PlanetLab Topology: In order to compare our approach against the two existing tree-based approaches (Sequoia and Song's approach) we created this topology based on a measurement dataset between about 385 PlanetLab hosts. The snapshot[8] we used contained about 130,000 measurements acquired by the bandwidth estimation tools PathRate [6], PathChirp [22] and Spruce [26].

[7] Our implementations are available at `http://www.uni-ulm.de/en/in/vs/proj/ic2`

[8] Acquired 2010-08-28 at 4:57:51PT using S^3; `http://networking.hpl.hp.com/s-cube`

Synthetic Topology: We investigate the impact of asymmetry on the prediction accuracy of Sequoia, Song and DAE using a synthetic dataset, which allows the definition of a particular asymmetry ratio r_ζ. It is generated by creating a certain amount of hosts, defined by two parameters: "upstream" and "downstream". The downstream value for a node is set randomly, while the corresponding upstream value is calculated based on the downstream bandwidth as $r_\zeta \cdot bw_\downarrow$ for an asymmetry factor $r_\zeta \in (0, 1)$. The synthetic scenario is based on the assumption that link bandwidth only depends on the last mile of a particular path. Thus, the bandwidth between two hosts is given by the minimum value of the upstream of the sender and the downstream of the receiver of a transmission as $bw(a, b) = \min(a_\uparrow, b_\downarrow)$. Note, that as a consequence of our derivation scheme of a distance matrix from our model, we need to contaminate our synthetic topology with hosts of high and symmetric bandwidth, too.[9] Otherwise, high downstream bandwidth would not be noticeable since no upstream bandwidth is high enough. We call this "pseudo-symmetry."

4.3 Prediction Tree Algorithms

Recall, that *Sequoia* constructs multiple (k) trees to mitigate distortion due to insertion order.[10] Furthermore, when embedding the dataset using Sequoia, the bandwidth between two nodes is defined as the average value of the bidirectional measurements. In case a measurement is only available for one direction of a link we assume the other direction to be equal. Following the authors of Sequoia, we set the Gromov product to negative infinity if one of the measurements is still not available.

In our evaluation of *Song*, we only use a single tree as proposed in the corresponding paper to generate our error analysis. Furthermore, symmetrisation of measurements is also done for Song's approach.

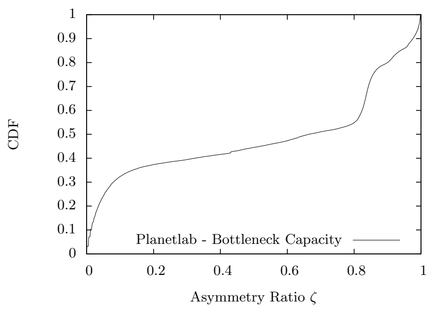
When embedding the dataset using DAE, bidirectional measurements are *not* averaged and, in contrast to the other approaches, missing measurements for only one direction are not replaced by the other direction but implicitly tolerated by our algorithm by using another anchor. Our algorithm also might make use of a unidirectional measurement value although the other direction is missing.

4.4 Accuracy of Prediction

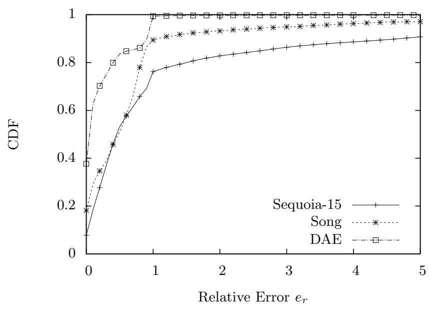
Figure 5a depicts the cumulative distribution of ζ-values in the PlanetLab topology. It is obvious that while roughly one third of hosts feature largely symmetric link conditions (small values on the x-axis), the other two thirds exhibit high values of asymmetry. Such a strong asymmetry on PlanetLab is surprising, as we expected hosts from research institutions to have good and symmetric connections. Asymmetry might be partially explained by varying points in time when the measurements were performed. Nevertheless, we conclude that asymmetry will be even stronger in settings comprising private Internet access links.

[9] We doted our topology with 33% high-bandwith, symmetric hosts.

[10] We set $k = 15$, as this provides 15% higher accuracy than $k = 10$.

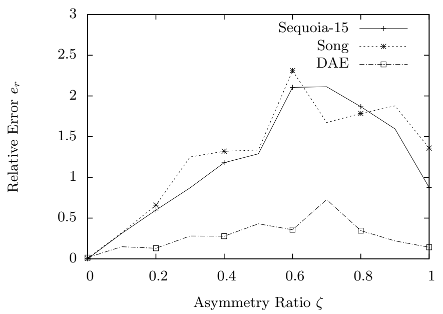


(a) Asymmetry Distribution

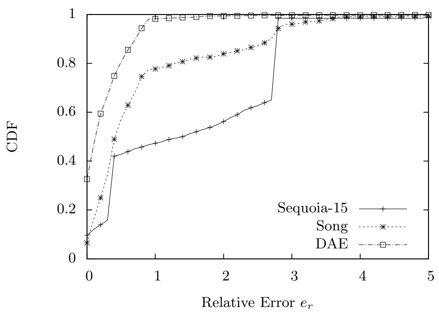


(b) Relative Link Prediction Error

Fig. 5. PlanetLab Topology



(a) e_p of all Tree Algorithms



(b) Relative Link Prediction Error

Fig. 6. Synthetic Dataset

When we feed this dataset to Sequoia and compare the predicted to the measured bandwidth, instead of the bidirectionally averaged values, we get the distribution depicted in Figure 5b. We see a heavy tail of huge prediction errors for $e_p > 1$. These would render the system practically unusable for peer selection purposes as motivated in the introduction.

As the PlanetLab dataset only represents a single mean ζ_{pl}-value, we studied the prediction performance with respect to varying ζ-values $\in [0, 1]$ using our synthetic topology. Figure 6a shows that increasing asymmetry has a severely negative impact on Sequoia's prediction performance. The advantage of DAE is proportional to the ratio and amount of asymmetry present in the dataset. There is no big advantage over Sequoia and Song for symmetric network conditions. In the presence of asymmetry, DAE explicitly takes this into account and allows a more accurate prediction.

In Figure 6a the error for various ζ-values is shown for each of the three tree algorithms. As can be seen, DAE significantly improves the accuracy of bandwidth prediction compared to Sequoia and Song. Song performs worse than Sequoia in this scenario, as it only constructs a single tree. We explain the increasing accuracy of Sequoia and Song's approach in Figure 6a for ζ-values above 0.7 by the measurement phenomenon pseudo-symmetry described in Chapter 4.2.

Considering the amount of measurements needed to construct the prediction trees, we found that Sequoia performed 84% of all $n(n-1)$ possible measurements for PlanetLab,[11] Song 13% and DAE 44%. We argue that DAE strikes a reasonable balance between accuracy and measurement traffic.

5 Related Work

Prediction of Internet path properties has been extensively studied. Especially the prediction of latency has been the focus of much attention in the past. Ng et al. [18] were the first to implement the idea of embedding inter-host latencies in an Euclidian space by assigning a coordinate to them, forming the research field of network coordinate systems. Their system is based on central landmark nodes to which common nodes measure their latency and use trilateration to locate themselves in the coordinate space. This approach has been decentralized in the widely known Vivaldi system [4], where Dabek et al. use a system of interconnected springs to model the location process of each node. Vivaldi's performance has been further improved by Elser at al. [7]. Common to these approaches is that the accuracy is highly susceptible to triangle inequality violations (TIVs). To tackle this problem embeddings into hyperbolic spaces [23] have been proposed. Furthermore, there is a line of research, which takes occurrences of TIVs as a hint for optimization potential, leveraging detour routing [15,9].

Embedding latencies is straightforward as this is an additive metric. Bandwidth on the other hand is a concave metric, and thus, does not lend itself for easy embedding in Euclidean spaces, as was shown by the authors of Sequoia [21]. Thus, systems have been devised to address this problem [8,10,16]. As bandwidth prediction under asymmetric bandwidth distributions is an even more challenging problem, Xing et al. [27] propose PathGuru, which embeds distance matrices in several ultra-metric spaces formed by pre-deployed landmarks. Maintaining incoming and outgoing bandwidth vectors, this system can predict asymmetric bandwidth distributions. However, the need for pre-deployed landmarks is a clear disadvantage.

There is a line of research based on matrix factorization, which was proposed to tackle the problem of triangle inequality violations. Mao et al. present the IDES system [17], a landmark-based approach which assigns each host an incoming and an outgoing vector. The distance between two hosts then is calculated by the scalar product of these vectors. The approach has later been decentralized by Liao et al. [13,12]. The core assumption common to these systems is that the distance matrix is of low rank and can be represented as the product of smaller matrices. In our problem domain, small rank corresponds to clustering of

[11] The relatively high amount of measurements results from using $k = 15$, as we get only about 25% of measurements when we use $k = 1$. Furthermore, as described in Section 4.1 we count measurements for both directions of a link individually.

nodes as nodes in a cluster will yield highly similar rows in the distance matrix. Though, tree-based approaches do not require such clustering.

Beaumont et al. study the "last-mile" model [2] first proposed by Liu et al. [14] and assume that perceived path bandwidth between hosts is solely determined by their access links. The authors consider scenarios where this is not the case as outliers, and cut them off using a percentile. Thus, they reduce potentially diverse path properties to single values for up- and downstream, which fails to capture situations where multiple hosts share a common bottleneck. In contrast, DAE allows bottlenecks to reside anywhere in the network.

6 Conclusion

In this paper, we presented *direction-aware embedding* (DAE), our approach for bandwidth prediction which is capable to cope with highly asymmetric bandwidth distributions. To the best of our knowledge, our approach is the only one simultaneously being fully decentralizable, independent of clustering and not assuming the bottleneck to reside on the last link. Opposed to existing tree-embedding approaches our scheme is able to maintain high prediction accuracy faced with asymmetric bandwidth distributions.

References

1. Abraham, I., Balakrishnan, M., Kuhn, F., Malkhi, D., Ramasubramanian, V., Talwar, K.: Reconstructing approximate tree metrics. In: Proc. of the 26th Ann. ACM Symp. on Princ. of Distr. Comp., pp. 43–52 (August 2007)
2. Beaumont, O., Eyraud-Dubois, L., Won, Y.J.: Using the last-mile model as a distributed scheme for available bandwidth prediction. In: Proc. of the 17th Int. Conf. on Par. Proc., pp. 103–116 (2011)
3. Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., Bowman, M.: Planetlab: an overlay testbed for broad-coverage services. SIGCOMM Comput. Commun. Rev. 33(3), 3–12 (2003)
4. Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: a decentralized network coordinate system. SIGCOMM Comput. Commun. Rev. 34(4), 15–26 (2004)
5. Donnet, B., Gueye, B., Kaafar, M.: A survey on network coordinates systems, design, and security. IEEE Comm. Surveys Tutorials 12(4), 488–503 (2010)
6. Dovrolis, C., Ramanathan, P., Moore, D.: Packet-dispersion techniques and a capacity-estimation methodology. Trans. on Netw. 12(6), 963–977 (2004)
7. Elser, B., Förschler, A., Fuhrmann, T.: Tuning vivaldi: Achieving increased accuracy and stability. In: Spyropoulos, T., Hummel, K.A. (eds.) IWSOS 2009. LNCS, vol. 5918, pp. 174–184. Springer, Heidelberg (2009)
8. Francis, P., Jamin, S., Jin, C., Jin, Y., Raz, D., Shavitt, Y., Zhang, L.: Idmaps: a global internet host distance estimation service. Trans. on Netw. 9(5), 525–540 (2001)
9. Haddow, T., Ho, S.W., Ledlie, J., Lumezanu, C., Draief, M., Pietzuch, P.: On the feasibility of bandwidth detouring. In: Spring, N., Riley, G.F. (eds.) PAM 2011. LNCS, vol. 6579, pp. 81–91. Springer, Heidelberg (2011)

10. Hu, N., Steenkiste, P.: Exploiting internet route sharing for large scale available bandwidth estimation. In: Proc. of the 5th ACM SIGCOMM Conf. on Internet Meas., p. 16 (2005)
11. Lakshminarayanan, K., Padmanabhan, V.N.: Some findings on the network performance of broadband hosts. In: Proc. of the 3rd ACM SIGCOMM Conf. on Internet Meas., pp. 45–50 (2003)
12. Liao, Y., Du, W., Geurts, P., Leduc, G.: Dmfsgd: A decentralized matrix factorization algorithm for network distance prediction. Trans. on Netw. PP(99), 1 (2012)
13. Liao, Y., Geurts, P., Leduc, G.: Network distance prediction based on decentralized matrix factorization. In: Proc. of the 9th IFIP TC 6 Int. Conf. on Netw., pp. 15–26 (2010)
14. Liu, S., Zhang-Shen, R., Jiang, W., Rexford, J., Chiang, M.: Performance bounds for peer-assisted live streaming. SIGMETRICS Perform. Eval. Rev. 36(1), 313–324 (2008)
15. Lumezanu, C., Baden, R., Levin, D., Spring, N., Bhattacharjee, B.: Symbiotic relationships in internet routing overlays. In: Proc. of the 6th USENIX Symp. on Netw. Sys. Des. and Impl., NSDI 2009, pp. 467–480 (2009)
16. Madhyastha, H.V., Isdal, T., Piatek, M., Dixon, C., Anderson, T., Krishnamurthy, A., Venkataramani, A.: iplane: an information plane for distributed services. In: Proc. of the 7th OSDI Conf., pp. 367–380 (2006)
17. Mao, Y., Saul, L., Smith, J.: Ides: An internet distance estimation service for large networks. J. on Sel. Areas in Comm. 24(12), 2273–2284 (2006)
18. Ng, T.S.E., Zhang, H.: Predicting internet network distance with coordinates-based approaches. In: Proc. IEEE INFOCOM, vol. 1, pp. 170–179 (2002)
19. Paxson, V.: End-to-end routing behavior in the internet. SIGCOMM Comput. Commun. Rev. 36(5), 41–56 (2006)
20. Peleg, D.: Proximity-preserving labeling schemes. J. Graph Theory 33(3), 167–176 (2000)
21. Ramasubramanian, V., Malkhi, D., Kuhn, F., Balakrishnan, M., Gupta, A., Akella, A.: On the treeness of internet latency and bandwidth. In: Proc. of the 11th Int. Joint Conf. on Meas. and Modeling of Comp. Sys., pp. 61–72 (2009)
22. Ribeiro, V., Riedi, R., Baraniuk, R., Navratil, J., Cottrell, L.: pathchirp: Efficient available bandwidth estimation for network paths. In: Passive and Active Meas. Workshop (2003)
23. Shavitt, Y., Tankel, T.: Hyperbolic embedding of internet graph for distance estimation and overlay construction. Trans. on Netw. 16(1), 25–36 (2008)
24. Song, S.: Decentralized pairwise bandwidth prediction, Unknown Date, `http://www.cs.umd.edu/Grad/scholarlypapers/papers/SukhyunSong.pdf`
25. Song, S., Keleher, P., Bhattacharjee, B., Sussman, A.: Decentralized, accurate, and low-cost network bandwidth prediction. In: Proc. IEEE INFOCOM, pp. 6–10 (April 2011)
26. Strauss, J., Katabi, D., Kaashoek, F.: A measurement study of available bandwidth estimation tools. In: Proc. of the 3rd ACM SIGCOMM Conf. on Internet Meas., pp. 39–44 (2003)
27. Xing, C., Chen, M., Yang, L.: Predicting available bandwidth of internet path with ultra metric space-based approaches. In: Proc. of IEEE GLOBECOM, pp. 1–6 (2009)

A Scalable Benchmark as a Service Platform

Alain Tchana[1], Noel De Palma[1], Ahmed El-Rheddane[1], Bruno Dillenseger[2], Xavier Etchevers[2], and Ibrahim Safieddine[1]

[1] Joseph Fourier University, LIG Laboratory (UJF/LIG), Grenoble, France
`firstname.lastname@imag.fr`
[2] Orange Labs, Grenoble, France
`firstname.lastname@orange.com`

Abstract. Load testing has always been a crucial and expensive activity for software companies. Classical solutions are a real burden to setup statically and their cost are prohibitive in terms of human and hardware resources. Cloud computing brings new opportunities to stress application scalability as load testing solutions can be provided on demand by the cloud. This paper describes a Benchmark-as-a-Service solution that scales automatically the load injection platform and eases its setup according to load profiles. Our approach is based on: (i) the virtualization of the Benchmarking platform to enable the injector's self-scalability, (ii) an online calibration mechanism to characterize injector capacity and impact on the benched application, (iii) a provisioning solution to scale the load injection platform sufficiently ahead of time. We also report experiments on a benchmark that shows the benefits in terms of cost and resources savings.

Keywords: Benchmarking as a service, Cloud.

1 Introduction

Load testing has always been a very crucial and expensive activity for Internet companies. Traditionally, it leverages a load injection platform capable of generating traffic according to load profiles to stress an application, a system under test or SUT for short, to its limits. Such solutions are a real burden to setup statically and their costs are prohibitive in terms of human and hardware resources.

Cloud computing brings new opportunities and challenges to test applications' scalability since it provides the capacity to deliver IT resources and services automatically on a per-demand, self-service (APIs) basis over the network. One characteristic is its high degree of automation for provisioning and on-demand management of IT resources (computation, storage and network resources) and services. IT resources can be provisioned in a matter of minutes rather than days or weeks.

Opportunities lay in the fact that load testing solution can be provided on demand as a service on the cloud. Such Benchmark-as-a-Service (BaaS) solution enables quite a number of benefits in terms of cost and resources. The cost of hardware, software and tools is charged on usage basis. The platform setup for the tests is also greatly simplified so that the testers can focus on their load injection campaign.

The challenge of Performance as a Service is to provide test teams with on-demand computing and networking resources, able to generate traffic on a SUT. Such test campaigns typically require more than a single load injection machine, to generate sufficient traffic (see Fig. 1). The issue is that the number of necessary load injection machines is not known in advance. It depends on the amount of resources consumed for generating and managing the requests and their responses, as well as on the target's global workload. The tester must empirically cope with these two risks:

– overloading the load injectors, causing scenarios not to behave as specified, and measures to be biased;
– wasting unnecessary resources.

For these reasons, we need a self-scalable load injection software making it possible to automatically adjust the number of load injection machines.

The contribution of this paper precisely addresses this challenge: We describe a BaaS solution (Section 3) which scales automatically the load injection platform. Besides the re-engineering of a load injection (Section 2) tool to enable self-scalability, the main concerns are (i) the injector's online calibration, (ii) the computation, based on from the load profile and the injector characterization, of the right amount of VMs and (iii) the control of their provisioning sufficiently ahead of time (Section 4 details these concerns). We also report experiments on the RUBiS [4] benchmark that shows the benefits in terms of self-scalability, including the cost reduction for long hours campaign (Section 5).

Section 6 and 7 present respectively the related work and the conclusion of this paper.

2 The CLIF Load Injection Framework

This work has been achieved in the context of the CLIF load injection framework which is a versatile load testing, open source software [2]. It is generic and extensible, in terms of target SUT protocols as well as resources to monitor. A workload scenario combines the definition of one or several virtual user (vUsers) behaviors, with the specification of the number of active vUsers as a function of time, called the *load profile.* A *behavior* is basically a sequence of requests interlaced with *think times* (i.e. periods of pause), enriched with conditional and loop statements, as well as probabilistic branches. These behaviors make use of plug-ins to support a variety of features, mainly injection protocols (HTTP, FTP, SIP. . .) and external data provisioning for request parameters variability.

As described in details in [1], CLIF's architecture is based on the Fractal component model [3], which easies its adaptation. *Load injector* and *probe* components are distributed through the network. The formers are responsible for

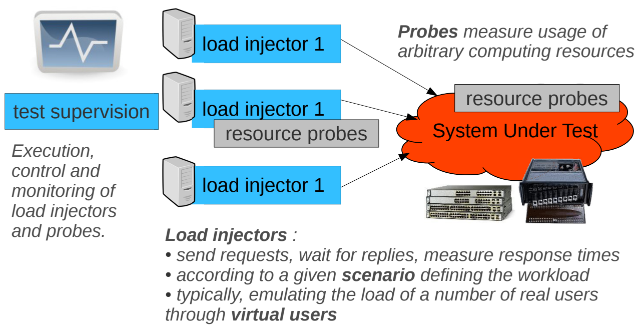

load injector 1
load injector 1
load injector 1
test supervision
Execution, control and monitoring of load injectors and probes.
resource probes
resource probes
System Under Test
Probes measure usage of arbitrary computing resources
Load injectors :
• send requests, wait for replies, measure response times
• according to a given scenario defining the workload
• typically, emulating the load of a number of real users through virtual users

Fig. 1. The big picture of a load testing infrastructure

generating the workload and measuring response times, while the latter measure the usage of given resources: CPU, memory, network adapter or equipment, database, middleware, etc.

Load injectors and probes are bound to a central control and monitoring component, namely the *Supervisor*, and a central *Storage* component that will collect all measures once a test execution is complete (Fig. 1). They are deployed on local or remote hosts. All these components are contained in the top-level, distributed *Clif Application (ClifApp) composite* component. The component based developement of CLIF facilitates its adaptation. We present in the nexts sections the implementation a scalable load testing framework based on CLIF.

3 BaaS Overview

This section describes the main components and design principles of our self-scalable Benchmarking-as-a-Service Platform (BaaSP) based on CLIF. The main purpose of the BaaSP is to minimize the cost of achieving the test in a cloud environment. This cost mainly depends on the number of virtual machines (VMs) used and their up time throughout the test. Since each CLIF injector is running on a separate VM, BaaSP proposes a testing protocol which attempts to reduce both the number of VMs used and their execution time. This protocol relies on dynamic addition/removal of CLIF injectors according to the variation of the submitted load profile. Roughly, instead of statically using an over sized number of VM injectors, the BaaSP dynamically adds or removes injectors during the test as needed by the workload. Besides, the BaaSP attempts to use an injector up to its maximum capacity before adding another one. Let us now present the self-scaling protocol we implement in the BaaSP:

1. Initial VMs allocation and systems deployment in the cloud: The first step is the deployment and the configuration of the CLIF benchmarking system, possibly including the system we want to test (the SUT). This latter is optional since the SUT can be deployed and configured a long time before

the BaaSP, with another deployment system. This phase includes the VMs allocation in the cloud. Note that the cloud platfom which runs the BaaSP can be different from the one, if any, running the SUT.
2. Calibration and planning. The calibration phase aims at knowing the maximum capacity of an injector VM. This knowledge will then allow to plan when to add/remove injectors during the test.
3. Test execution and injectors provisioning. The actual test starts with a minimal number of injectors. The execution of these latter (request injection) should follow the submitted load profile. The BaaSP adds/removes injector VMs according to the planning done in the previous stage.
4. Systems undeployment and VM deallocation. This phase is opposite to the first one. At the end of the test, the BaaSP automatically undeploys and frees all the VMs it has instantiated in the cloud.

Figure 2 presents the BaaSP architecture. It is organized as follows. A VM (called BaaSPCore) is responsible of orchestrating the test: initialization of each test phase (deployment, calibration, test launching and undeployment). The Calibrator component is responsible for evaluating the capacity of an injector, while the Planner plans when to add/remove injectors VM during benchmarking.

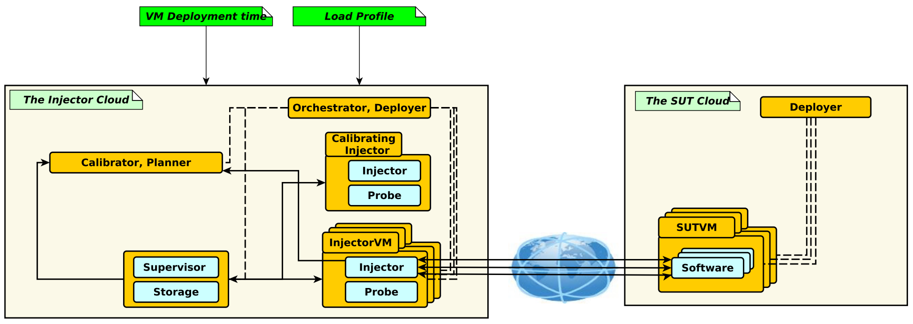


Fig. 2. Self-Scaling BaaS Architecture

4 Self-scaling Protocol

4.1 Calibrating with CLIF Selfbench

This phase aims at evaluating the load injection capacity of an injector VM, in terms of greatest number of clients it is able to emulate (vUsers).

In order to evaluate the capacity of an injector, the Calibrator uses a CLIF extension module called Selfbench [5]. Selfbench results from research work on automating performance modelling of black boxes. Part of this work consists in a self-driven workload ramp-up, looking for the maximum number of vUsers a SUT can serve until its resources are considered as insufficient.

Since Selfbench makes no assumption about the SUT capacity, it starts with a single vUser. From the response times and throughput the load injector gets, Selfbench computes the SUT's theoretical maximum capacity, with minimal assumptions in terms of parallel processing capability (single-threaded). Then, Selfbench increases the number of vUsers step-by-step, until reaching either the theoretical maximum capacity, or the SUT saturation limit. The number of steps is defined as a parameter. If the SUT is saturated, then the maximum number of vUsers it can serve has been reached. Otherwise, Selfbench makes a more optimistic assumption about the SUT's capacity, with a greater parallel processing capability, and runs a new step-by-step workload increase. The determination of steps duration combines theoretical results on queuing modeling and statistical considerations about the number of samples and their stability. The SUT saturation is defined as maximum or minimum thresholds on a number of load metrics, such as CPU usage, free memory or any other resource usage that a CLIF probe may monitor.

For the work we are presenting here, we use Selfbench in a slightly different way. The injector VM calibration is not based on detecting the SUT saturation but on detecting the injector VM saturation. Thus, the CLIF probes must be deployed at the injector VMs rather at the SUT side (even though it should be checked that the SUT is not saturating). At the end of its execution, Selfbench gives the number of vUsers reached before injector VM saturation (which represents the capacity of an injector).

4.2 Planning

Assuming that all injector VMs in the BaaSP have the same quantity of resources, then all injectors will have the same capacity (called InjMaxCapacity in the rest of this paper) as evaluated by the Calibrator. Based on this assumption and the time required to deploy injector VMs, the Planner is then able to plan injectors provisioning ahead of time for the given load profile. Let TTSVM be the deployment time function, which gives for a given number of VMs, the deployment time needed to start them in the cloud. This function is given by the operator of the BaaSP platform and depends on the cloud infrastructure utilized (In our case, we profiled our private cloud to configure this parameter as reported Section 4.3). The load profile (W) can be expressed as a discrete function of number of vUsers over the time: vUsers = W(t), means that the load profile requires "vUsers" to emulated the required workload at time t. Thus, the planning process can be expressed as a function: f(W,InjMaxCapacity,TTSVM). The Planner parses the load profile (W) and produces the provisioning rate (taking into account the deployment time TTSVM), according to the capacity of an injector VM (InjMaxCapacity). Thus, we are able to start injectors just ahead of time. In fact, instead of adding an injector at time t, the Planner will fire at time t - TTSVM. The planning algorithm returns a hash table

VMAt (key,value) where each "key" represents a time when the Planner should add/remove injectors.

Let VMAt[t1] = $InjAt_{t1}$ and VMAt[t2] = $InjAt_{t2}$, with t2 be the next key in VMAt following t1. If $InjAt_{t1} < InjAt_{t2}$, the Planner will add $InjAt_{t2}$ - $InjAt_{t1}$ injectors at time t2. Else, the Planner will remove $InjAt_{t1}$ - $InjAt_{t2}$ injectors. The algorithm we propose groups at a unique time, all injectors that will start in the same time frame. The time frame is defined as follows.

Another role of the Planner is to prepare the load profile which will be executed by added injectors during the benchmark. Alg. 1 is the algorithm used by the Planner to generate these load profiles. If Inj_Max represents the maximum number of injectors that can be simultaneously used during the test, the Planner generates Inj_Max of load profiles: W_i, $1 \leq i \leq Inj_{Max}$. The purpose of Alg. 1 is to generate all W_i. Then, when an injector "i" is added, it is configured to use a corresponding W_i. How a load profile is assigned to an injector is given by the algorithm. During the test, injectors which are running are ordered from 1 to currentNbInj, where currentNbInj is the current number of injectors. Thus, each injector i, $1 \leq i \leq CurrentNbInj$, runs the load profile W_i. When the Planner wants to add nbAdd of injectors, it sorts them from $currentNbInj + 1$ to $currentNbInj + nbAdd$. Each new injector j, $CurrentNbInj + 1 \leq j \leq CurrentNbInj + nbAdd$, will run the load profile W_j.

Finally, the Planner is implemented as a control loop. If "Timers" represents the set of keys (which are dates) of the hash table VMAT, then the Planner wakes up at each element in "Timers" and adjusts the number of injectors. The first entry of VMAt (VMAt[0]) represents the initial number of injectors, deployed before the beginning of the benchmarking process.

4.3 Injector Dynamic Provisioning

Injector Addition Protocol. The Planner initiates the addition of new injectors. Fig. 3 (1) summarizes the protocol we implement:

(a) The Planner asks the Deployer to create a number of new injector VM required at a given time in the existing environment. This request contains the load profile that the injectors will run.
(b) The Deployer asks the IaaS to start each new required VM in parallel.
(c) Each new VM is equipped with a deployment agent which informs the Deployer that it is up.
(d) The Deployer sends to each new injector its configuration, including the load profile it will run.
(e) Each new injector registers its configuration by contacting the ClifApp.
(f) ClifApp integrates the new injector configuration in its injector list and forwards this configuration to its inner component (supervisor...). After this, the ClifApp requests the added injectors to start their load injection according to the profile.

Algorithm 1. Injectors Workloads Planning

```
    In
      - VMAt[e]: Provisioning hash table
      - W[t]: The benchmark workload

    Out
      - $W_i$[t]: Generated workloads, similar to W[t]

Begin
 1: ForEach key e of VMAt do
 2:    If (e is the last key of VMAt)
 3:       exit
 4:    End If
 5:    next_e := next key of VMAt after e
 6:    For i from e to next_e do
 7:       For j from 1 to VMAt[e] do
 8:          $W_j$[i] := W[i]/VMAt[e]
 9:       End For
10:    End For
11: End For
End
```

Injector Removal Protocol. Like with the previous protocol, the Planner initiates the removal of injectors. The removal protocol we implement is presented in Fig. 3(2):

(a) The Planner asks the Deployer to stop the execution of a number of injector VMs.
(b) The Deployer asks the ClifApp to unregister the injectors from the ClifApp injector list. The ClifApp forwards this reconfiguration to its inner components (supervisor...) and requests the corresponding injectors to stop their load injection.
(e) Once the injector has been unregistered from the ClifApp, the Deployer is notified by the ClifApp that the corresponding VMs are no longer taken into account.
(f) Finally, the Deployer asks the IaaS to turn off the VM hosting the injector.

After the calibration and the planning phases, there are two ways to go on with the test. If the SUT needs to be stabilized before being in a usable state, then the calibration phase is considered as the stabilizer phase. Thus, the real test will go on immediately after calibration. Otherwise, the SUT is restarted before launching the test. In both cases, the Planner adapts the initial number of injectors according to the load profile and the injector VM capacity. The next section is dedicated to the evaluation of this protocol.

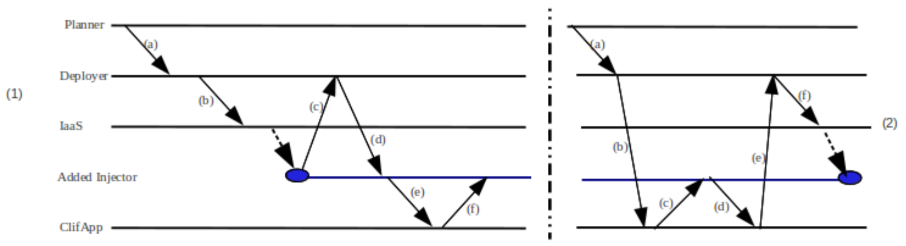


Fig. 3. (1) Example of adding an injector (initiated by the Planner); (2) Example of removing an injector (initiated by the injector itself)

5 Evaluation

5.1 Evaluation Context

The System under Test. The SUT is provided by RUBiS [4] (1.4.3 application version), a JEE benchmark based on servlets. RUBiS implements an auction web site modeled over eBay. It defines interactions such as registering new users, browsing, buying or selling items. For this evaluation, we submitted only browsing requests to the RUBiS application. We deploy the RUBiS open source middleware solution composed of: one Apache (2.2.14) web server (with Mod_JK 2 to connect to the application server), a Jakarta Tomcat (6.0.20) for servlets container (with AJP 13 as the connector), and a MySQL server (5.1.36) to host auction items (about 48 000 of items).

Cloud Environment. Our experiments were carried out using the Grid'5000 [10] experimental testbed (the French national grid). Grid'5000 is organized in "sites" (a site represents a city), which in turn are organized in clusters. For our experiments, we configure two Grid'5000's clusters (Chicon at Lille, north of France; and Pastel at Toulouse, south of France) to provide separately the SUT cloud and the Injector Cloud (as shown in the BaaSP architecture in Figure 2). The two clusters run OpenStack [9] in order to provide virtualized cloud. The virtualization system is KVM version 2.0.0. We start each RUBiS VMs with 1GB of memory while injectors and the others BaaSP VMs used 256MB of memory. Each VM is pinned to one processor. They run the same operating system as the nodes which host them, which is Linux Ubuntu 10.04 distribution with a 2.6.30 kernel, over a gigabit connection. In this environment, we have calibrated the deployment time TTSVM. This time has an asymptotic behavior. For example, the deployment time of 1 VM until 10 VMs is the same (100s) while it grows up from 11VMs to 20VMs (with a difference of 75s). For readability, we use in this section TTSVM instead of TTSVM[i] for $1 < i < 10$.

5.2 Evaluation Scenarios and Metrics

Workload Scenarios. Two workload scenarios have been experimented. These two workload scenarios summarize two situations corresponding to the worst case and a better case for our BaaSP system. Theoretically, each workload is designed to run in 1200 seconds.

The first workload scenario (Figure 4) represents a "simple" test workload W_s(t). This workload is composed of two phases: a ramp-up phase (W_s(nt)=nW_s(t)) followed by a ramp-down phase (W_s(nt)=$\frac{W_s(t)}{n}$), forming together a pyramidal workload. It needs addition/removal of single injector.

The second workload scenario (Figure 5) is more complex. It is composed of several kind of phases: gentler upward load, constant load, steep ramp-up load, steep ramp-down load, and gentle ramp-down load. **This kind of workload scenario shows how the BaaSP becomes more beneficial. In fact, unlike the first workload scenario, this second workload scenario needs sometime the addition/removal of more than one injector at once.**

The ultimate goal of our BaaSP is to minimize the cost of benchmarking an application in a cloud environment. Naturally, the main metric used in the evaluation is the cost of the test. We compare the cost of the test in two situations: static injectors deployment (called $Policy_0$) vs self-scaling injectors provisioning through our BaaSP. This second case was evaluated according to the following policies:

- $Policy_1$: injectors are dynamically added/removed without "just ahead time" provisioning.
- $Policy_2$: injectors are dynamically added/removed using a "just ahead time" provisioning strategy.

The cost of running the test in the cloud depends on both the duration of the test and the number of VM used during the test. In this three situations, the duration of the test includes: SUT and BaaSP startup, and the real test performing time. Calibrating time and injector addition/removal time are included when evaluating $Policy_1$ and $Policy_2$.

5.3 Evaluation Results

The evaluation results we describe in this section are classified in two categories: quantitative and qualitative. Quantitative evaluation depicts the actual results we obtained regarding SUT and cloud test bed environment behaviors' during the test. Besides, we analyze the behavior of each provisioning policy during the test. Regarding qualitative evaluation, it concerns the formalization of observations coming from the quantitative evaluation. Before describing these results, let us present the variables on which the qualitative evaluation is based:

Let $Cost_{tu}$ be the cost of running a VM (as described earlier) in the cloud, during a time unit TU. Let TTS be the time to start both the SUT and BaaSP VM. Let TTT be the theoretical time to run the benchmark workload (20 minutes in our case). Let nbVMRubis be the number of VM used to run RUBiS.

Let nbVMBaaSP be the number of VM used to run BaaSP components. Let nbInj be the maximum number of injectors used during the test (5 in our case). The cost of running the test in the cloud without dynamic provisioning ($Policy_0$), noted $Cost_0$, is given by the following formula:

$$Cost_0 = [nbInj{*}TTT{+}(nbVMRubis{+}nbVMBaaSP){*}(TTT{+}TTS)]{*}Cost_{tu}$$

Workload Scenario 1. We compare the cost of $Policy_0$ to the two others policies: $Policy_1$ (injectors addition is done without a "just ahead of time" provisioning), and $Policy_2$ (injectors addition is done in a "just ahead of time" provisioning).

Figure 4(1) presents actual results of the execution of this first workload scenario. It shows two types of curves: (a) the variation of the CPU load of the RUBiS database tier (remember that it is the bottleneck of our RUBiS configuration), and (b) the injector provisioning rate during the test. These curves are interpreted as follows:

– The behavior of the SUT follows the specified workload (pyramidal). We observe that this workload saturates the MySQL node in term of CPU consumption (100%) at the middle of the load profile.
– The execution of the test with $Policy_1$ has a wrong behavior while $Policy_0$ and $Policy_2$ follow the same behavior, which corresponds to what we expected (according to the workload scenario). In fact, $Policy_1$ extends the test duration more than the theoretical duration specified in the workload scenario: about 400s, corresponding to range (c) shown in Figure4(1).
– We observe long stairs during the upward phase with $Policy_1$ compared $Policy_2$ because the deployment time of a new injector is not anticipated by $Policy_1$ as it is with $Policy_2$. We don't observe the same phenomenon in the lowering phase because injectors' removal is immediate.
– As shown in the curves (b) in Figure4(1), we use up to five injectors VM for each workload scenario.

Remember that TTSVM is the time used by the BaaSP to add an injector (about 100s in our experiment). Let TTCal be the time used by the Calibrator to calibrate an injector (60s in our experiment) and InjMaxCapacity be the maximum capacity of an injector (40 vuser in our experiment). With $Policy_1$, as we said that the test runs more than the theoretical time. This time corresponds to the sum of the deployment time of all injectors added by the Planner during the test, (nbInj-1)*TTSVM.

Let TTSI be the time needed to saturate an injector during the ramp-up phase (120s in our experiment). With $Policy_1$, TTSI+TTSVM is the provisioning period during the ramp-up phase whereas it is TTSVM in the ramp-down phase. On the other hand, TTSVM is the provisioning period with $Policy_2$ in both ramp-up phase and ramp-down phase. Figure 4(2) shows the execution time of each injector using $Policy_0$, $Policy_1$ and $Policy_2$. Here are the formulas used to evaluate the cost of these different cases. Let $ExecTime_i$ be the execution time of injector i, and $ExecTime_{RubisBaaSP}$ be the execution time of RUBiS and BaaSP VM.

- $Cost_1 = Cost_0 + (\frac{nbInj(nbj-1)}{2}(TTSVM - 2TTSI) + TTCal + (nbVMRubis + nbVMBaaSP)[TTCal + (nbj-1)TTSVM]) * Cost_{tu}$ **(E1)**
- $Cost_2 = Cost_0 + [(nbVMRubis + nbVMBaaSP + 1)TTCal + nbInj * TTSVM - 2 * nbInj(nbj-1)TTSI] * Cost_{tu}$ **(E2)**

Regarding equations (E1) and (E2), dynamic provisioning is less expensive than static execution when:

- $Policy_1$: $TTSI \geq \frac{(nbVMRubis+nbVMBaaSP+1)TTCal}{nbInj(nbInj-1)} + \frac{(nbInj/2+nbVMRubis+nbVMBaaSP)(nbInj-1)TTSVM}{nbInj(nbInj-1)}$ (C1)

- $Policy_2$: $TTSI \geq \frac{(nbVMRubis+nbVMBaaSP+1)TTCal}{2*nbInj(nbInj-1)} + \frac{TTSVM}{2(nbInj-1)}$ (C2)

When conditions (C1) and (C2) are respected, $Cost_0 \leq Cost_1 \leq Cost_2$. In our experimental environment, we have: nbInj=5, nbVMRubis=3, nbVMBaaSP=2, TTS=250s, TTCal=60s, TTSVM=100s, and TTSI=120s. In this context, (C1) is not met whereas (C2) is. Then, $Cost_0 = 13250 * Cost_{tu}$, $Cost_1 = 14210 * Cost_{tu}$, and $Cost_2 = 9310 * Cost_{tu}$.

Workload Scenario 2. Fig.5 shows results for the second workload scenario. The interpretation of these results is similar to the previous workload scenario. Unlike the first scenario, we only evaluate the BaaSP when the "just ahead of time" provisioning is activated. Looking at the workload of this scenario, curve (a) of Fig. 5(1) shows injectors provisioning:

- An injector is added at time T1.
- Three injectors are simultaneously added at time T2. This is done according to the steep ramp-up phase occurs for a short time (from time 550s to 600s), which is less than TTSVM (refer to the planning algorithm).

Fig. 5(2) shows the execution time of this experiment (Fig. 5(2)(b)) in comparison to the static execution (Fig. 5(2)(a)). From the same variables we used in the previous section, the cost of this experiment ($Cost_3$) is evaluated as follows:

$Cost_3 = [(nbVMRubis + nbVMBaaSP) * (TTT + TTS + TTCal) + \sum_{i=1}^{nbInj} ExecTime_i] * Cost_{tu}$

$Cost_3 = 10570 * Cost_{tu}$

The value of $Cost_3$, in comparison to $Cost_0$ (which is always fix), shows that for some scenarios (long test campaign in preference), the gain of using dynamic injector provisioning becomes more interesting.

6 Related Work

Very few works are interesting on adaptive benchmarking tools. However, we study some work situated around this topic. Unibench [12] is an automated benchmarking tool. As our BaaSP, Unibench is able to deploy remotely both the

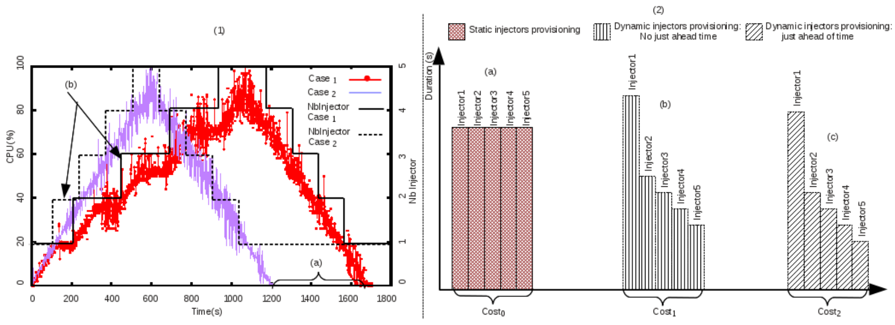


Fig. 4. Injectors provisioning in the BaaSP when running the first scenario

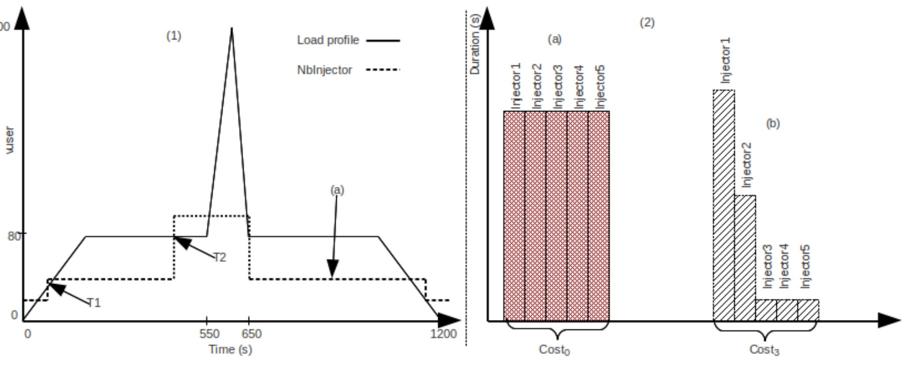


Fig. 5. Injectors provisioning in the BaaSP when running the second scenario

SUT and benchmarking components in a cluster. It is adaptive in the way that it is able to identify a modification in the SUT and then achieve another benchmarking process related to this modification. To do this, Unibench is supposed to know the source code and the programing language of the SUT. Unlike our BaaSP, the SUT is not considered as a black box. [13] presents research challenges for implementing benchmarking tools for self-adaptative systems. Except the definition of metrics and some principles to be considered when defining workload, it does not care about the self-adaptation of the benchmarking tool itself.

CloudGauge [14] is an open source framework similar to ours. It uses the cloud environment as the benchmarking context. Unlike our BaaSP which evaluate a SUT running in the cloud, CloudGauge SUTs' is the cloud and its capability for VM consolidation. It dynamically injects workloads to the cloud VM and adds/removes/migrate VM according to the fluctuation of the workload. As Selfbench [5] (the calibration system we used), it is able to adjust itself the workload during the benchmarking process. Indeed, like our BaaSP, users can define a set of workloads for benchmarking. Since the SUT is the cloud, injectors are deployed into VM. There is no separation between injectors nodes and SUT nodes. Thus, unlike our BaaSP, there is no need to dynamically create injectors node as injectors and SUT share the same VMs. Regarding the architecture of CloudGauge, we observe some similarity with ours. For example, CloudGauge defines an orchestrator called *Test Provisioning* which is responsible to orchestrate the benchmarking process. Other tools such as VSCBenchmark [15] and VMark [16] are comparable to CloudGauge. They allow the definition of dynamic workload for VM consolidation benchmarking in a cloud environment.

As far as we know, there is no open source benchmarking framework with comparable characteristic as ours. However, there is some proprietary and commercial tools. Looking at the marketing speech of BlazeMeter [17], it provides same features as ours (except SUT deployment): dynamic injectors allocation and de-allocation in the cloud in order to reduce test cost. It is an evolution of the JMeter [11] tool for cloud platform. Since it is proprietary, there is no technical and scientific description of BlazeMeter. Therefore, it becomes difficult to really compare its functionalities to ours. NeoLoad [18] is another tool similar to BlazeMeter. It allows deployment of injectors in a cloud environment for benchmarking an application. It is able to integrate new injectors throughout the benchmarking process. However, this integration should be initiated by the administrator by planning. NeoLoad does not implement itself an automated planning component, as we did.

7 Conclusion

This paper explores Cloud Computing features to ease application benchmarking and to stress their scalability. Load testing solution can be provided on demand in the cloud and can benefit from self-scalability.

We describe a Benchmark-as-a-Service solution that provides a number of benefits in terms of cost and resources savings. The cost of hardware, software and tools is charged on a pay-per-use basis and platform setup is also greatly simplified. The self-scalability property of the platform eases the benchmarking process and lower the cost for long hours campaign since it does not require to statically provision the platform which is prohibitive in terms of human and hardware resources. Resource provisioning is minimized while ensuring load injection according to a given profile. Our experiments based on the RUBiS benchmark show the benefits in terms of cost reduction for long hours testing campaigns. As for as we know, our Benchmark-as-a-Service platform is the only one that scales automatically the resources used for load injection.

As a future work we plan to add a new mode to our platform. With this mode, load profiles are not required any more. It aims at automatically provisioning and controlling load injection resources until saturating the SUT. The difficulty here is to stress progressively an application near its limits while preventing thrashing. This requires a fine grain load injection control and provisioning.

Acknowledgment. This work is supported by the French Fonds National pour la Societe Numerique (FSN) and Poles Minalogic, Systematic and SCS, through the FSN Open Cloudware project.

References

1. Dillenseger, B.: CLIF, a framework based on fractal for flexible, distributed load testing. Annals of Telecommunications 64(1-2), 101–120 (2009)
2. The CLIF Project, `http://clif.ow2.org` (visited on February 2013)
3. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: An Open Component Model and Its Support in Java. In: International Symposium on Component- Based Software Engineering (CBSE), Edinburgh, UK (2004)
4. Amza, C., Cecchet, E., Chanda, A., Cox, A.L., Elnikety, S., Gil, R., Marguerite, J., Rajamani, K., Zwaenepoel, W.: Specification and implementation of dynamic web site benchmarks. In: IEEE Annual Workshop on Workload Characterization, Austin, TX, USA (2002)
5. Harbaoui, A., Salmi, N., Dillenseger, B., Vincent, J.: Introducing queuing network-based performance awareness in autonomic systems. In: ICAS, Cancun, Mexico (2010)
6. Tchana, A., Temate, S., Broto, L., Hagimont, D.: Autonomic resource allocation in a J2EE cluster. In: Utility and Cloud Computing, Chennai, India (2010)
7. Amazon Web Services, Amazon EC2 auto-scaling functions, `http://aws.amazon.com/fr/autoscaling/` (visited on February 2013)
8. Righscale web site, `http://www.rightscale.com` (visited on February 2013)
9. Openstack web site, `http://openstack.org/` (visited on February 2013)
10. Grid'5000 web site, `https://www.grid5000.fr` (visited on February 2013)
11. The Apache Software Foundation, Apache JMeter, `http://jmeter.apache.org/` (visited on February 2013)
12. Rolls, D., Joslin, C., Scholz, S.-B.: Unibench: a tool for automated and collaborative benchmarking. In: ICPC, Braga, Portugal (2010)
13. Almeida, R., Vieira, M.: Benchmarking the resilience of self-adaptive software systems: perspectives and challenges. In: SEAMS, Waikiki, Honolulu, HI, USA (2011)
14. El-Refaey, M.A., Rizkaa, M.A.: CloudGauge: a dynamic cloud and virtualization benchmarking suite. In: WETICE, Larissa, Greece (2010)
15. Jin, H., Cao, W., Yuan, P., Xie, X.: VSCBenchmark: benchmark for dynamic server performance of virtualization technology. In: IFMT, Cairo, Egypt (2008)
16. Makhija, V., Herndon, B., Smith, P., Roderick, L., Zamost, E., Anderson, J.: VMmark: a scalable benchmark for virtualized systems, Technical Report VMware-TR-2006-002, Palo Alto, CA, USA (September 2006)
17. BlazeMeter,Dependability benchmarking project, `http://blazemeter.com/` (visited on February 2013)
18. Neotys, NeoLoad: load test all web and mobile applications, `http://www.neotys.fr/` (visited on February 2013)

Failure Analysis and Modeling in Large Multi-site Infrastructures

Tran Ngoc Minh and Guillaume Pierre

IRISA / University of Rennes 1, France
{minhtn,guillaume.pierre}@irisa.fr

Abstract. Every large multi-site infrastructure such as Grids and Clouds must implement fault-tolerance mechanisms and smart schedulers to enable continuous operation even when resource failures occur. Evaluating the efficiency of such mechanisms and schedulers requires representative failure models that are able to capture realistic properties of real world failure data. This paper shows that failures in multi-site infrastructures are far from being randomly distributed. We propose a failure model that captures features observed in real failure traces.

1 Introduction

Large computing infrastructures such as Grids and Clouds have become indispensable to provide the computing industry with high-quality resources on demand. However, failures of computing resources create an important challenge that these infrastructures must address. Failures cause a reduction of the total system capacity, and they also negatively impact the reliability of applications making use of the resources. Understanding failures from a statistical point of view is therefore essential to design efficient mechanisms such as checkpointing and scheduling in Grids and Clouds.

This paper presents a comprehensive analysis of failure traces from five large multi-site infrastructures [10]. We focus on simultaneity, dependence and multiplication features. Simultaneity measures the extent to which multiple failures or multiple recoveries happen at the same time. Dependence means that times between failures have short- and long-term autocorrelations. Finally, multiplication captures the fact the times between failures are not smoothly distributed but rather occur at multiples of specific durations. Similar features were not present in previous studies of clusters, peer-to-peers and web/dns servers. We therefore believe that they are characteristic of large multi-site systems. This analysis enables us to model failures and generate realistic synthetic failure scenarios that can be used for further studies of fault-tolerance mechanisms. The advantage of a failure model is that it enables us to tune parameters as we wish, which is not possible when replaying a trace.

This analysis is based on five traces of node-level failures from the Failure Trace Archive [10]. These traces, described in Table 1, can be considered as representative of both Grid and Cloud infrastructures. For example, GRID'5000 which is currently used to serve Cloud users is a good representative of low-level Cloud infrastructures. However, since these traces were collected when serving Grid jobs, obviously virtualization is another source of failures in Clouds that we do not consider.

This paper is organized as follows. Sections 2, 3 and 4 respectively analyse the simultaneity, dependence and multiplication properties in failure traces. Section 5 proposes a failure model that Section 6 validates with real world data. Finally, Section 7 discusses related work and Section 8 concludes the paper.

Table 1. Details of failure traces used in our study

ID	System	Nodes	Period, Year	Res.[1]	#(Un)availability Events
MSI1	CONDOR-CAE	686	35 days, 2006	300	7,899
MSI2	CONDOR-CS	725	35 days, 2006	300	4,543
MSI3	CONDOR-GLOW	715	35 days, 2006	300	1,001
MSI4	TERAGRID	1,001	8 months, 2006-2007	300	1,999
MSI5	GRID'5000	1,288	11 months, 2006	5	419,808

2 Simultaneity of Failures and Recoveries

Let T be a set of N ordered failures: $T = \{F_i | i = 1 \ldots N \text{ and } F_i \leq F_j \text{ if } i < j\}$, where F_i denotes the time when a failure i occurs. Each failure i is associated with an unavailability interval U_i, which refers to the continuous period of a service outage due to the failure, and the time $R_i = F_i + U_i$ indicates the recovery time of the failure. For a group of failures T' that is a subset of T, we consider a failure i as a simultaneous failure (SF) if there exists in T' any failure $j \neq i$ such that j and i happen at the same time, otherwise we call i a single failure. Similarly, we also consider i as possessing a simultaneous recovery (SR) if there exists in T' any failure $k \neq i$ such that k and i recover at the same time, otherwise i possesses a single recovery.

Assigning T' as a whole failure trace, we calculate the fractions of SFs and SRs in real multi-site systems, which are shown in the first and second rows of Table 2. As we can see, simultaneous failures and recoveries are dominant in all cases. We do a further analysis to check how many simultaneous failures possess SRs. This is done by determining all groups of SFs, i.e. all failures that occur at the same time are gathered into one group, and calculating the number of SRs for each group. We then average and show the result in the third row of Table 2. We conclude that most of the simultaneous failures recover simultaneously.

Considering each trace separately, failures of MSI1 and MSI2 almost occur and reoperate concurrently, so it is not surprising when most of the simultaneous recoveries belong to SFs. But this does not apply for MSI5, where only 69%

[1] Res. is the trace resolution. For instance, a node failure at time t with resolution 5 seconds means that the actual failure time was between $t - 5$ and t.

Table 2. Fractions of SFs (R1) and SRs (R2) in real systems, calculated for the whole trace. The third row (R3) shows fractions of SRs, calculated for groups of SFs.

	MSI1	MSI2	MSI3	MSI4	MSI5
R1	97%	93%	75%	73%	95%
R2	98%	97%	81%	75%	95%
R3	97%	98%	94%	93%	69%

resources with SF become available simultaneously despite of a large number of SFs and SRs (95%). This is due to failures that occur but do not recover at the same time with some failures, instead they recover concurrently with other failures. In contrast with MSI5, MSI3 and MSI4 only exhibit around 70%-80% SFs and SRs but a large number of SRs are from SFs. One can think of the resolution of the traces as the reason that causes SFs and SRs. However, we argue that the resolution is not necessarily a source of the simultaneity feature since the resolution is relatively small and cannot cause such a large number of SFs and SRs. A more plausible reason that causes a group of nodes to fail at the same time is that the nodes share a certain device/software whose failure can disable the nodes. For example, the failure of a network switch will isolate all nodes connected to it. Its recovery will obviously lead to the concurrent availability of the nodes.

We believe that the simultaneity feature is common in data-center-based systems. We therefore argue that fault-tolerant mechanisms or failure-aware resource provisioning strategies should be designed not only to tolerate single node failures, but also massive simultaneous failures of part of the infrastructure.

3 Dependence Structure of Failures

We now deal with a set of times between failures $\{I_i\}$, whose definition is based on the set of ordered failures $T = \{F_i\}$ in Section 2. We determine a time between failures (TBF) as $I_i = F_i - F_{i-1}$ and hence it is easy to represent $\{F_i\}$ by $\{I_i\}$ or convert $\{I_i\}$ to $\{F_i\}$. This section examines the dependence structure of $\{I_i\}$[2]. The term "dependence" of a stochastic process means that successive samples of the process are not independent of each other, instead they are correlated. A stochastic process can exhibit either short or long range dependence (SRD/LRD) as shown by its autocorrelation function (ACF). A process is called SRD if its ACF decays exponentially fast and is called LRD if the ACF has a much slower decay such as a power law [3]. Alternatively, the Hurst parameter H [7], which takes values from 0.5 to 1, can be used to quantitatively examine the degree of dependence of a stochastic process. A value of 0.5 suggests that the process is either independent [1] or SRD [11]. If $H > 0.5$, the process is considered as LRD, where the closer H is to 1, the greater the degree of LRD.

[2] In terms of statistics, we consider $\{I_i\}$ as a stochastic process.

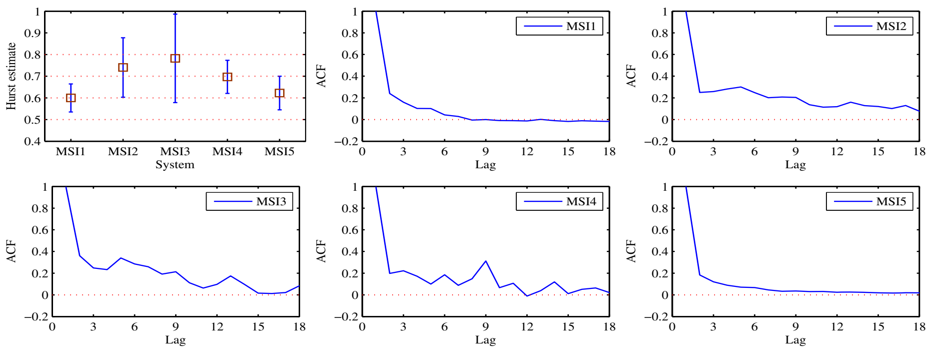


Fig. 1. Hurst parameter and autocorrelation functions of TBFs in real traces

Figure 1 shows the dependence feature. To quantitatively measure how TBFs are autocorrelated, we estimate the Hurst parameter of real TBF processes. The estimation is done with the SELFIS tool [9]. As there are several available heuristic estimators that each has its own bias and may produce a different estimated result, we chose five estimators (*Aggregate Variance, R/S Statistic, Periodogram, Abry-Veitch* and *Whittle*) and computed the mean and the standard deviation of their estimates. For all cases real TBFs are indeed LRD because all Hurst estimates are larger than 0.5. The most noticable point focuses on MSI2, MSI3 and MSI4, which result in H around 0.7 to 0.8. It shows that the TBFs of these traces are largely autocorrelated. This is confirmed by observing their ACFs in Figure 1, which decay slowly and determine their LRD feature. The LRD of MSI1 and MSI5 are not very strong since their estimated Hurst parameters are larger but not very far from 0.5. In particular, the ACF of MSI1 decays though not exponentially but quickly to 0, thus one can also consider it as SRD or in our case, we call it as exhibiting weak LRD.

It is important to capture the LRD feature in modeling because it may significantly decrease the computing power of a system by the consecutive occurrence of resource failures. In particular, if simultaneous failures happen with LRD, a system will become unstable and it is hard to guarantee quality-of-service requirements. Fault-tolerant algorithms should therefore be designed for correlated failures to increase the reliability.

4 Multiplication Feature of Failures

An interesting feature of failure traces is the distribution of times between failures. However, we have seen that the vast majority of failures are simultaneous. This would result in several TBFs with value 0 as shown in Table 3. The occurence of a large number of zeroes in a TBF process makes it difficult to fit TBFs to well-known probability distributions. Therefore, instead of finding a best fit for the whole TBF process, we remove zeroes out of the process and only try to fit TBFs that are larger than 0, so-called positive TBFs or PTBFs.

Table 3. The fraction of zero values in TBF processes

MSI1	MSI2	MSI3	MSI4	MSI5
93%	91%	71%	65%	90%

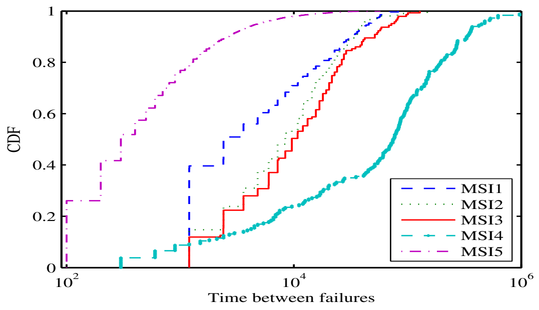


Fig. 2. Cumulative distribution functions of real PTBFs

Figure 2 shows cumulative distribution functions (CDFs) of PTBFs of all failure traces. For each trace, the PTBFs are fitted to the following five distributions: Generalized Pareto (GP), Weibull (Wbl), Lognormal (LogN), Gamma (Gam) and Exponential (Exp). The maximum likelihood estimation method [12] is used to estimate parameters for those distributions in the fitting process, which is done with a confidence level $\gamma = 0.95$ or a significance level $\alpha = 1 - \gamma = 0.05$. For each distribution with the estimated parameters in Table 4, we use a goodness-of-fit test, called Kolmogorov-Smirnov (KS test) [3], to assess the quality of the fitting process. The null hypothesis of the KS test is that the fitted data are actually generated from the fitted distribution. The KS test produces a p-value that is used to reject or confirm the null hypothesis. If the p-value is smaller than the significance level α, the null hypothesis is rejected, i.e. the fitted data are not from the fitted distribution. Otherwise, we can neither reject nor ensure the null hypothesis.

Table 5 shows that PTBFs of MSI1 and MSI5 cannot be fitted well to any distribution candidate since all p-values are equal to 0. The reason lies in Figure 2, where we can easily observe staircase-like CDFs in the two traces. This shape indicates that the data tend to distribute around some specific values. Further analysing PTBFs of MSI1 and MSI5, we find that most of them are multiples of so-called basic values. As shown in Table 6, MSI1 has a basic value of 1200 seconds as 100% of its PTBFs are multiples of 1200[4]. We refer to this property as the multiplication feature of failures. Other traces show similar behavior.

Although MSI2, MSI3 and MSI4 exhibit this feature and their CDFs also have staircase-like shapes, their PTBFs still can be fitted to some distributions. Table 5 indicates that Gam and Exp are suitable for MSI2 where Exp is the best. Though GP is the best for MSI3, its PTBFs can be fitted to any

[4] We consider a as being a multiple of b if $|a/b - round(a/b)| < 0.005$.

Table 4. Parameters of distributions estimated during the fitting process. a, b, μ, σ indicate shape, scale, mean and standard deviation, respectively.

	GP(a, b)	Wbl(a, b)	LogN(μ, σ)	Gam(a, b)	Exp(μ)
MSI1	0.7 4653	0.8 8548	8.4 1.3	0.7 15415	10490
MSI2	0.2 11546	0.9 13540	9 1.2	1 14596	14034
MSI3	0.3 14028	0.9 17272	9.2 1.2	0.9 21013	18420
MSI4	0.5 72854	0.7 97029	10.6 2	0.5 235811	126615
MSI5	0.7 445	0.7 838	6 1.3	0.6 2112	1227

Table 5. P-values of fitting PTBFs, obtained from the KS test. Those larger than the significance level $\alpha = 0.05$ are in gray boxes.

	GP	Wbl	LogN	Gam	Exp
MSI1	0	0	0	0	0
MSI2	0.03	0.03	0.04	0.07	0.08
MSI3	0.29	0.19	0.17	0.26	0.09
MSI4	0	0.08	0	0.29	0
MSI5	0	0	0	0	0

distribution candidate. Finally, Gam should be the best choice for MSI4 besides Wbl. Different from MSI1 and MSI5, the staircases in the CDFs of MSI2, MSI3 and MSI4 are relatively small, so have the CDFs be possible to fit the distribution candidates. In contrast, MSI1 and MSI5 focus their PTBFs on their basic value (see Figure 2) and hence the PTBFs are hard to fit the tested distributions. As there is a consensus among MSI2, MSI3 and MSI4, we suggest that the Gamma distribution can be used as a marginal distribution-based model for PTBFs, where zeroes can be added to form a complete TBF process. However, this would be a simple model that is able to capture neither the dependence nor the multiplication feature and hence its representativeness is limited.

It is hard to explain why PTBFs exhibit the multiplication feature. One possible cause is that this is an artifact of the trace resolution. For example, MSI4 has a resolution of 300 seconds so all TBFs in the trace are multiples of 5 minutes even if the actual failures did not exhibit this property. However, this explanation does not fully explain the phenomenon in the other four traces since their resolutions are different from their basic values. Therefore in addition to the resolution, there may be other causes that we did not discover due to limited available information in each trace. We argue that this would be an interesting information to be added in the Failure Trace Archive [10]. Since almost PTBFs in all five traces are multiples of a basic value, it is essential to take this feature into account in our failure models.

5 Failure Modeling

This section presents a model for times between failures that is able to capture all the practical features analysed in previous sections, including the simultaneity,

Table 6. Basic values and fractions of PTBFs that are multiples of a basic value

	MSI1	MSI2	MSI3	MSI4	MSI5
Basic value (s)	1200	1200	1200	300	100
Fraction	100%	99%	99%	100%	100%

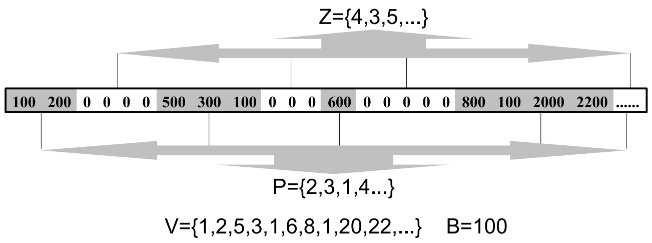


Fig. 3. Illustration of how the model gathers information from its input

the dependence and the multiplication. The model described in Algorithm 1 receives a TBF process $\{I_i\}$ as its input and produces a synthetic TBF process $\{S_i\}$ with those three features, which can be converted into a sequence of failure events used in performance study.

5.1 General Model

Our failure model in Algorithm 1 consists of three steps. Firstly, we extract necessary information from the TBF process input $\{I_i\}$, where the extraction is explained in Figure 3. As we indicated in Table 3, the TBF process of a failure trace contains a large number of zeroes due to the simultaneity feature, it is reasonable to set up a 2-state model: $\{I_i\}$ goes to state-0 if its value is zero, otherwise it is with state-1. Once $\{I_i\}$ falls into a state, we will determine how long it remains in the state before switching to the other state. For example with a TBF process in Figure 3, we form for state-0 a set Z that contains the lengths of all zero sequences. With respect to state-1, we produce a similar set P with the lengths of all PTBF sequences. We also determine the basis value B of the TBF process that will be used later for the multiplication feature. Furthermore, all PTBFs are collected, divided by B and stored into a set V. With Z, P, V, B, we gather enough information and finish the first step of the model. As the second step, we find the best fitted marginal distribution for Z, P and V, denoted by $DistZ$, $DistP$ and $DistV$, respectively. The fitting methodology will be presented later in Section 5.2.

As the last step, we generate $\{S_i\}$ through a main loop. We initialize by randomly picking a state. Then, we sample a value r, which indicates how many TBFs should be created in this state, by using $DistZ$ or $DistP$, depending on the state. With r, the dependence structure of $\{S_i\}$ can be controlled as similar as that of $\{I_i\}$. If the state is state-0, we generate a sequence of r zero values and switch to state-1. The zero sequence helps to create simultaneous failures and

Algorithm 1. The failure model

```
Input: a TBF process $\{I_i\}$.
Output: a synthetic TBF process $\{S_i\}$.

$[Z, P, V, B] = ExtractInfo(\{I_i\})$; // Extract necessary information from input
$DistZ = Fit(Z)$; $DistP = Fit(P)$; $DistV = Fit(V)$; // Find fitted distributions
$state = random(\{0, 1\})$; $N = 0$; // Initialize

repeat
  if $state = 0$ then
    $r = round(Sampling(DistZ))$;
    $S_{N+1} \dots S_{N+r} = 0$;
  else
    $r = round(Sampling(DistP))$;
    for $j = 1$ to $r$ do
      $S_{N+j} = round(Sampling(DistV)) * B$
                                  $+[-Res * UniF]$; // Optional
    end for
  end if
  $N = N + r$;
  $state = 1 - state$;
until $N + 1 \geq$ desired number of failures;
```

hence helps to capture the simultaneity feature for $\{S_i\}$. If the state is state-1, we generate a sequence of r PTBFs, each is formed by sampling $DistV$ and multiplying with B to obtain the multiplication feature[5]. Then, we switch to state-0 and continue the loop until the desired number of failures is achieved. Indeed, the model operates similarly as a 2-state Markov chain [4], where there is no probability for a state to switch to itself.

5.2 Fitting Methodology

In order to find the best fits for Z, P and V sets, we also apply the maximum likelihood estimation method and the KS test on the five well-known distribution candidates as described in Section 4. Since data are hard to fit any distribution if they contain some specific values that are dominant over other values, as illustrated when we fit PTBFs of MSI1 and MSI5 in Section 4, we carefully check if this happens with the Z, P and V sets. In Table 7, we list top four values that appear in the sets with their frequency. As we can see in most

[5] Since failures are reported with a resolution, our model also allows to generate "actual" failure times if one needs by subtracting each generated PTBF an amount of $Res * UniF$, where Res is the resolution and $UniF$ is the uniform distribution in the range $[0, 1]$.

cases, values 1 and 2 are dominant over other values. Therefore, we remove values 1 and 2 out of the fitting process. Furthermore since the applied distribution candidates support a non-negative value domain and we already remove the two smallest values out of the sets, which results in 3 as the new smallest value, we decide to shift all remaining values of the sets by 3 units to ease the fitting process. In summary, let X be any from the Z, P and V sets, we will find the best fit for the set $Y = \{y = x - 3 | x \in X \setminus \{1, 2\}\}$.

Table 7. Top four values (ordered) appear in the Z, P and V sets. Reading format: a value above and a frequency in percentage below, correspondingly.

	Z	P	V
MSI1	(1 2 8 6)	(1 2 3 4)	(1 2 3 4)
	(17 8 6 5)	(60 21 6 5)	(40 11 5 4)
MSI2	(1 2 3 25)	(1 2 4 3)	(1 2 3 4)
	(30 14 7 5)	(37 12 9 7)	(15 9 7 7)
MSI3	(1 2 9 3)	(1 3 10 2)	(1 2 5 3)
	(45 9 9 5)	(26 17 13 4)	(12 10 6 6)
MSI4	(1 2 3 6)	(1 2 3 6)	(1 2 3 5)
	(43 13 5 5)	(28 15 10 10)	(4 3 2 1)
MSI5	(1 2 3 4)	(1 2 3 4)	(1 2 3 4)
	(24 12 9 6)	(63 19 8 4)	(26 16 10 6)

Table 8. P-values of fitting Y sets, obtained from the KS test. Those larger than the significance level $\alpha = 0.05$ are in gray boxes.

Trace/Set	GP	Wbl	LogN	Gam	Exp
MSI1/Z	0.06	0.01	0.00	0.00	0.00
MSI1/P	0.00	0.00	0.00	0.00	0.00
MSI1/V	0.06	0.00	0.00	0.00	0.03
MSI2/Z	0.56	0.02	0.00	0.01	0.20
MSI2/P	0.59	0.01	0.00	0.00	0.67
MSI2/V	0.11	0.00	0.00	0.00	0.07
MSI3/Z	0.50	0.41	0.06	0.34	0.20
MSI3/P	0.23	0.03	0.01	0.02	0.23
MSI3/V	0.59	0.00	0.00	0.00	0.23
MSI4/Z	0.06	0.12	0.01	0.07	0.20
MSI4/P	0.41	0.00	0.00	0.00	0.41
MSI4/V	0.05	0.00	0.00	0.00	0.00
MSI5/Z	0.00	0.00	0.00	0.00	0.00
MSI5/P	0.00	0.00	0.00	0.00	0.00
MSI5/V	0.00	0.00	0.00	0.00	0.00

Table 8 shows the results of fitting Y sets, which indicate that GP seems to be the best fitting candidate. In 11/15 cases, GP results in p-values larger than the significance level $\alpha = 0.05$. Therefore in these cases, the null hypothesis that Y sets are from the GP distribution cannot be rejected. In addition, though Exp can also be a good candidate, its p-values are smaller than those of GP in most cases. Hence, we suggest that GP should be the best choice for fitting Y sets. As all distribution candidates result in p-values equal to 0 in the other four cases, we additionally use the KS statistic, also produced by the KS test, to select the best distribution. From Table 9, we again confirm that GP should be a suitable choice for fitting Y sets since its KS statistics are smallest, except for the P set of MSI5. Hence for generality of the model, we propose and use GP as the fitting distribution for Y sets in our study, where the estimated parameters of GP are shown in Table 10.

In conclusion, let X be any from the Z, P and V sets, the fitted distribution $DistX$ of X is determined by the following parameters: percentage of value 1 in X (p_1), percentage of value 2 in X (p_2) and GP parameters (a, b). To sample a value x from $DistX$, we first sample a value pr from the uniform distribution over the range $[0, 1]$. If $pr \leq p_1$, we assign $x = 1$, else if $p_1 < pr \leq p_1 + p_2$, $x = 2$. Otherwise, $x = g + 3$, where g is sampled from the GP distribution with parameters (a, b).

Table 9. KS statistics of fitting Y sets, obtained from the KS test

Trace/Set	GP	Wbl	LogN	Gam	Exp
MSI1/P	0.32	0.40	0.41	0.44	0.32
MSI5/Z	0.14	0.27	0.39	0.28	0.22
MSI5/P	0.36	0.33	0.35	0.33	0.44
MSI5/V	0.17	0.32	0.40	0.31	0.28

Table 10. Estimated parameters of GP(a, b), where a and b indicate shape and scale

	MSI1	MSI2	MSI3	MSI4	MSI5
Z	0.33 20.82	0.45 46.35	0.45 19.19	-0.75 27.87	0.87 8.04
P	-0.17 1.82	-0.08 5.77	0.02 5.31	0.10 3.87	12.54 0.00
V	0.11 12.10	0.23 9.21	0.24 12.53	0.35 302.96	1.07 3.97

6 Validation of the Model

We present in this section our experiments to validate our model. We apply the model to all the traces in Table 1 to generate synthetic failures. The quality of these synthetic failures is evaluated by comparing with the real data. Our evaluation focuses on the simultaneity feature, the dependence structure and the marginal distribution of TBFs. Evaluating the multiplication feature is not necessary because it is guaranteed when we generate PTBFs by sampling the distribution $DistV$ and multiplying with the basic value B (see Algorithm 1).

6.1 Simultaneous Failures

Table 11 describes fractions of simultaneous failures produced by the model. It can be seen that our model controls well this feature since the fractions are close to those of the real data. The quality of generating this feature depends on fitting the Z sets. Successfully fitting the sets, as shown in Tables 8 and 9, helps to control well the number of zeros generated in a TBF process. In case of MSI5, though the p-value of fitting the Z set to GP is 0, its KS statistic is small. Thus, the fitting is acceptable, resulting in a good control of the simultaneity.

Table 11. Fractions of simultaneous failures produced by the model

	MSI1	MSI2	MSI3	MSI4	MSI5
Data	97%	93%	75%	73%	95%
Model	96%	90%	79%	70%	97%

Table 12. Compare the Hurst parameter between the model and the data, presented as *mean* ± *standard deviation* of the five estimators

	MSI1	MSI2	MSI3	MSI4	MSI5
Data	0.60 ± 0.06	0.74 ± 0.14	0.78 ± 0.20	0.70 ± 0.08	0.62 ± 0.08
Model	0.62 ± 0.05	0.76 ± 0.06	0.76 ± 0.18	0.68 ± 0.06	0.54 ± 0.04

6.2 Dependence Structure

We evaluate the long range dependence feature both via observing an autocorrelation function and estimating the Hurst parameter. The estimation is done similarly as presented in Section 3, i.e. using the SELFIS tool with the five estimators, namely *Aggregate Variance, R/S Statistic, Periodogram, Abry-Veitch* and *Whittle* [9]. As we can see in Figure 4, the autocorrelation of the model fits well to that of the real data, except for the case of MSI5. This is in accordance with the quantitative results of estimating the Hurst parameter in Table 12. It is not strange when the model does not fit MSI5 since we cannot find good fitting distributions for the Z, P and V sets of MSI5 as shown in Section 5.2. In contrast for the other cases, the fitting step of the model is well done and thus, the model is able to generate autocorrelated failures.

6.3 Marginal Distribution

One of the first aspects often received the attention of researchers when they analyse or model failures is the marginal distribution. It can be seen that our model is highly representative since it can capture realistic observed features of failures, namely the simultaneity, the dependence and the multiplication. However, its representativeness is even better if it can fit the marginal distributions of real times between failures. Indeed, this is confirmed in Figure 4 where we draw

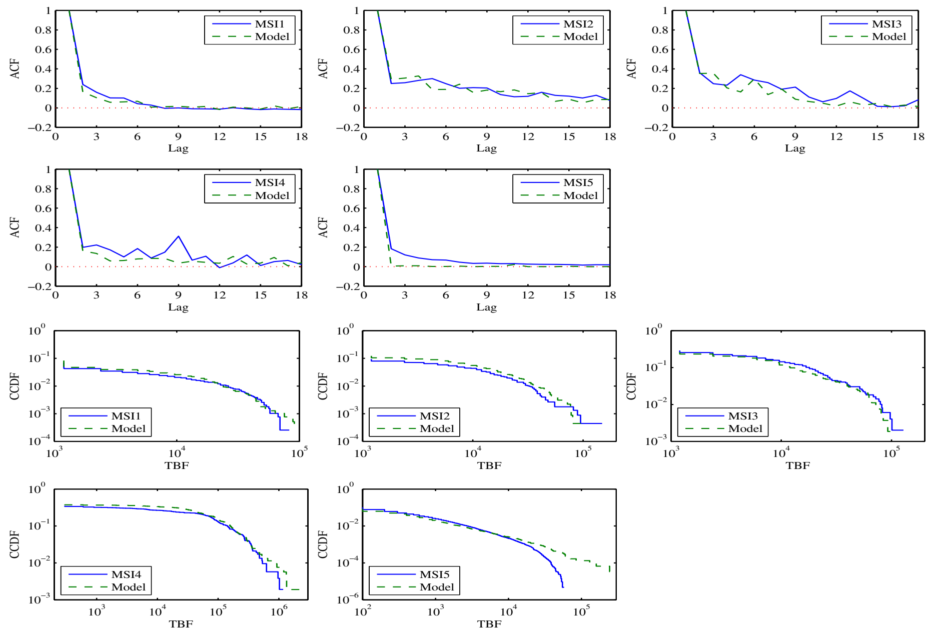


Fig. 4. Fitting autocorrelation functions and marginal distributions of TBFs

the complementary cumulative distribution functions (CCDFs) of synthetic and real TBFs. The figure shows that our model fits the marginal distribution feature well in four cases. For MSI5, the fitting step does not work finely, which makes the generated TBFs not able to fit the real TBFs well. Nevertheless, the fitting quality of MSI5 is acceptable since the ugly fitting part only occurs when TBFs are larger than 10,000 seconds, which just occupy $\sim 0.2\%$ number of TBFs.

7 Discussion and Related Work

Our study demonstrates that Grid and Cloud infrastructures exhibit properties of simultaneity, dependence and multiplication that must be modeled to accurately capture the characteristics of such systems. Interestingly, the same features are not necessarily present in other types of large-scale systems such as desktop grids and P2P systems. Table 13 measures the occurence of these features in a number of systems, and highlights systems which clearly exhibit them. Only 2/14 systems exhibit all three features. We so argue that it is essential to develop a specific failure model for systems such as Grids and Clouds.

Many studies have been dedicated to analysing and modeling failures [2, 5, 6, 8, 13–19]. However, most of them focus on servers, high performance clusters, peer-to-peer systems, etc. The few studies dedicated to multi-site systems [6, 8, 18] did not concentrate on modeling the observed features. In [8], a failure model is developed based on fitting real data to distribution candidates, but none of

Table 13. The simultaneity (S), the dependence (D) and the multiplication (M) features in other systems, expressed by the fraction of simultaneous failures, the Hurst parameter and the basic value, respectively. Grey boxes indicate systems which exhibit the corresponding feature clearly (S <50% and $D < 0.6$ are not considered).

System	Type	S	D	M
UCB	Desktop Grid	5%	0.47	No
MICROSOFT	Desktop Grid	100%	0.52	3600
LRI	Desktop Grid	31%	0.56	No
DEUG	Desktop Grid	5%	0.70	No
NOTRE-DAME (host availability)	Desktop Grid	90%	0.69	960
NOTRE-DAME (CPU availability)	Desktop Grid	99%	0.58	960
PLANETLAB	P2P	67%	0.65	900
OVERNET	P2P	100%	0.50	1200
SKYPE	P2P	100%	0.48	1800
SDSC	HPC Cluster	18%	0.53	No
LANL	HPC Cluster	15%	0.76	60
PNNL	HPC Cluster	42%	0.55	100
WEBSITES	Web Server	1%	0.63	No
LDNS	DNS Server	10%	0.56	No

the features observed in this study is captured. Moreover, this model is designed specifically only for GRID'5000 and it is not clear whether it would work for other systems. The model proposed by Yigitbasi et al. [18] studies peaks of failures but not times between failures. Although it studies autocorrelation, it is in terms of failure rates and is not taken into account in their model. Finally, the authors of [6] showed that failures often occur closely in time, in so-called group of failures. The concept of group of failures is close to the simultaneity feature in this paper, and we believe that groups of failures occur due to the vast majority of simultaneous failures as shown in Table 2. Hence, modeling simultaneous failures is more accurate than modeling groups of failures since the information about the times between failures in a group could not be recovered once they are grouped for modeling. Furthermore, the dependence and multiplication features are not taken into account by [6], possibly because it aims at other large-scale systems than multi-site infrastructures.

8 Conclusions and Future Work

This paper demonstrated that failures exhibit simultaneity, dependence and multiplication features, which can have a significant impact on system performance. We proposed a failure model that can capture these features and help research on fault-tolerance mechanisms. The Gamma distribution alone may be used as a marginal distribution-based model for PTBFs. However, the model from Section 5 offers much more precision with respect to these three features. It is also flexible as the parameters of GP distributions can easily be tuned. Finally, it addresses the issue of trace resolution and generates "actual" failure times that are not affected by the resolution. Our future work includes adding the unavailability attribute and using the model to associate failure-awareness for enhancing scheduling performance or resource provisioning in Clouds.

Acknowledgements. This work was partially funded by the Harness project of the Seventh European Framework Programme (FP7/2007-2013) under grant agreement number 318521 and by the Netherlands Organization for Scientific Research (NWO) in the context of the Guaranteed Delivery in Grids project.

We would like to acknowledge Lex Wolters for his initial contribution to this work. We regret that he passed away on March 2012.

References

1. Beran, J.: Statistics for Long-Memory Processes. Chapman & Hall (1994)
2. Chu, J., Labonte, K., Levine, B.N.: Availability and Locality Measurements of Peer-to-Peer File Systems. In: ITCom (2002)
3. Feitelson, D.G.: Workload Modeling for Computer Systems Performance Evaluation., Book Draft, Version 0.32 (2011)
4. Feller, W.: An Introduction to Probability Theory and Its Applications (1950)
5. Gainaru, A., Cappello, F., Snir, M., Kramer, W.: Fault Prediction under the Microscope: a Closer Look into HPC Systems. In: SC (2012)
6. Gallet, M., Yigitbasi, N., Javadi, B., Kondo, D., Iosup, A., Epema, D.: A Model for Space-Correlated Failures in Large-Scale Distributed Systems. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part I. LNCS, vol. 6271, pp. 88–100. Springer, Heidelberg (2010)
7. Hurst, H.E.: Long Term Storage Capacity of Reservoirs., Trans. ASCE (1951)
8. Iosup, A., et al.: On the Dynamic Resource Availability in Grids. In: GRID (2007)
9. Karagiannis, T., et al.: A User-Friendly Self-Similarity Analysis Tool (2003)
10. Kondo, D., et al.: The Failure Trace Archive: Enabling Comparative Analysis of Failures in Diverse Distributed Systems. In: CCGRID (2010)
11. Lillo, F., Farmer, J.: The Long Memory of the Efficient Market (2004)
12. Myung, J.: Tutorial on Maximum Likelihood Estimation. J. Math Psy. (2003)
13. Nurmi, D., Brevik, J., Wolski, R.: Modeling Machine Availability in Enterprise and Wide-Area Distributed Computing Environments. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 432–441. Springer, Heidelberg (2005)
14. Oppenheimer, D., Ganapathi, A., Patterson, D.A.: Why do Internet Services Fail, and What Can Be Done about It? In: USITS (2003)
15. Pecchia, A., Cotroneo, D., Kalbarczyk, Z., Iyer, R.K.: Improving Log-Based Field Failure Data Analysis of Multi-Node Computing Systems. In: DSN (2011)
16. Sahoo, R.K., Squillante, M.S., Sivasubramaniam, A., Zhang, Y.: Failure Data Analysis of a Large-Scale Heterogeneous Server Environment. In: DSN (2004)
17. Schroeder, B., Gibson, G.A.: A Large-Scale Study of Failures in High-Performance-Computing Systems. In: DSN (2006)
18. Yigitbasi, N., Gallet, M., Kondo, D., Iosup, A., Epema, D.: Analysis and Modeling of Time-Correlated Failures in Large-Scale Distributed Systems. In: GRID (2010)
19. Zheng, Z., et al.: 3-Dimensional Root Cause Diagnosis via Co-Analysis (2012)

Evaluating the Price of Consistency in Distributed File Storage Services

José Valerio, Pierre Sutra, Étienne Rivière, and Pascal Felber

University of Neuchâtel,
Switzerland

Abstract. Distributed file storage services (DFSS) such as Dropbox, iCloud, SkyDrive, or Google Drive, offer a filesystem interface to a distributed data store. DFSS usually differ in the consistency level they provide for concurrent accesses: a client might access a cached version of a file, see the immediate results of all prior operations, or temporarily observe an inconsistent state. The selection of a consistency level has a strong impact on performance. It is the result of an inherent tradeoff between three properties: consistency, availability, and partition-tolerance. Isolating and identifying the exact impact on performance is a difficult task, because DFSS are complex designs with multiple components and dependencies. Furthermore, each system has a different range of features, its own design and implementation, and various optimizations that do not allow for a fair comparison. In this paper, we make a step towards a principled comparison of DFSS components, focusing on the evaluation of consistency mechanisms. We propose a novel modular DFSS testbed named FlexiFS, which implements a range of state-of-the-art techniques for the distribution, replication, routing, and indexing of data. Using FlexiFS, we survey six consistency levels: linearizability, sequential consistency, and eventual consistency, each operating with and without close-to-open semantics. Our evaluation shows that: *(i)* as expected, POSIX semantics (i.e., linearizability without close-to-open semantics) harm performance; and *(ii)* when close-to-open semantics is in use, linearizability delivers performance similar to sequential or eventual consistency.

1 Introduction

Distributed file storage services (DFSS) offer a unified filesystem view of unstructured distributed data stores. As for any distributed storage service, the expected properties of a DFSS are consistency, availability, and tolerance to partitions. The CAP theorem [1] states that a distributed storage system can fully support at most two of these three properties simultaneously. Partition tolerance is usually considered essential for a DFSS, as data centers may be temporarily disconnected in a large-scale distributed setting and such events must be supported. As a result, developers of DFSS usually decide on a tradeoffs between availability and consistency.

Historically, DFSS designs have focused on providing the POSIX strong model of consistency [2]. The need for planetary-scale and always available services, and thus the shift to geographically distributed platforms and cloud architectures, has led DFSS designs to focus more heavily on availability and weaker consistency levels. By introducing caching mechanisms and the *close-to-open* semantics, the Andrew filesystem (AFS) [3] was one of the first systems to offer a consistency level weaker than POSIX. Most operations in systems such as HDFS [4] or GoogleFS [5] are sequentially consistent. However, both systems are built around a central metadata server. Schvachko [6] recently pointed out that this approach is inherently not scalable because the metadata server cannot handle massive parallel writes and the physical limits of a central metadata server design hits the petabyte barrier, i.e., the system cannot address more than 10^{15} bytes. On the other hand, flat storage systems like Cassandra or Dynamo [7, 8] propose an even weaker consistency level: eventual consistency. This relieves designers from the need for a central metadata server. Some systems [9–11] implement a filesystem interface on top of an eventually consistent storage system. However, to the best of our knowledge, none has gained wide acceptance.

Enabling further research on DFSS to scale and break the petabyte barrier requires developers to understand and be able to compare systematically the multiple components of a design. These components include data distribution and replication (and associated consistency guarantees), request routing, data indexing and querying, or access control. Performing a fair comparison of these aspects as supported by existing DFSS implementations is difficult because of their inherent differences. Indeed, these systems propose not only different filesystem consistency levels (FSCLs), but they also feature different base performance and optimization levels, which largely depend on the programming language, environment, runtime, etc.

In this paper, we make a step toward allowing the systematic comparison of DFSS designs. We instantiate our approach by isolating and evaluating the impact of the FSCL on performance. We make the following contributions:

- We depict the construction of a filesystem on top of a regular key/value store with the simple addition of the *compare-and-swap* primitive.
- We present a clear typology of the different FSCLs and categorize existing implementations accordingly.
- We compare empirically FSCLs by instantiating them into a novel DFSS testbed named FlexiFS. Our testbed is modular and implements a range of state-of-the-art techniques for the distribution, replication, routing, and indexing of data.

Our main findings are the following: *(i)* as expected, POSIX semantics (i.e., atomicity without close-to-open semantics) harms performance; and *(ii)* when close-to-open semantics is in use, atomicity delivers performance similar to sequential or eventual consistency.

The remainder of this paper is organized as follows: We describe the design of FlexiFS in Section 2. Section 3 introduces FSCLs and their corresponding implementations in FlexiFS. We present several benchmark results that evaluate each level in Section 4. Section 5 surveys related work. We close in Section 6.

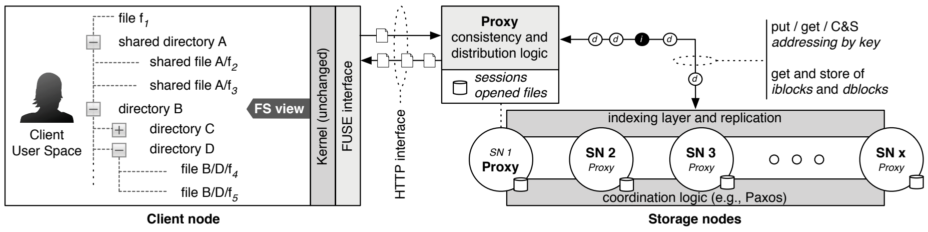


Fig. 1. General Architecture of FlexiFS

2 Testbed Design

FlexiFS is a distributed file storage service (DFSS) offering a transparent filesystem interface. Figure 1 illustrates its general architecture. FlexiFS has been built in a modular way to allow evaluation of different choices of DFSS designs. A typical deployment contains two sets of nodes: *storage nodes* implement a distributed flat storage layer, while *client nodes* present a filesystem abstraction to the users, and store files and folders hierarchies on the storage nodes.

A client node accesses the filesystem through a *filesystem in user space* (FUSE) implementation [12]. FUSE is a loadable kernel module that provides a filesystem API to the user space. It lets non-privileged users create a filesystem without writing any kernel code. In FlexiFS, each access to the filesystem is transformed into a Web service request and routed toward a *proxy* node that acts as an entry point to the distributed storage. The proxy redirects requests to the adequate storage node(s), which store or return data blocks.

2.1 Proxy

The role of the proxy is to hide both the topology of the distributed storage and the operation logic from the client. In FlexiFS, any storage node can act as a proxy. When a client executes an operation and contacts a proxy via its Web service interface, the proxy accesses the underlying storage system, executes the operation, and returns the result to the client.

The storage layer is essentially a key/value store extended with a *compare-and-swap* primitive. Devising a filesystem on top of this interface is a contribution of our work. The operations of the interface are as follows:

– $put(k, v)$: writes the value v for key k,
– $get(k)$: returns the data stored for key k,
– $C\&S(k, u, v)$: executes a compare-and-swap on key k with old value u and new value v.[1]

Depending on the FSCL in use in FlexiFS, the semantics of the above interface may change. For instance, $C\&S()$ is not atomic under eventual consistency. We detail how FlexiFS implements this interface in Section 3.

[1] $C\&S(k, u, v)$ checks whether the stored value is still u, and if so, replaces u by v; in any case the old value is returned.

2.2 Distributed Storage Layer

FlexiFS is modular and decouples the filesystem logic from the actual storage. The storage layer supports data indexing and provides the API described in the previous section. Due to its modular design, FlexiFS is able to use different indexing and storage layers, such as a multi-hop DHT or a central server. Below, we detail the design common in flat storage layers [7, 8] that we use in this paper.

FlexiFS's indexing and storage layer is a simple yet efficient one-hop DHT structured as a ring that relies on consistent hashing to store and locate data. Figure 1 presents its general architecture. It supports the following features:

Routing. For performance reasons and in order to reduce noise in our experiments, we have chosen a one-hop routing design, i.e., every node knows all other nodes in the ring.

Elasticity. Upon joining, a node chooses a random identifier along the ring and fetches the ring structure from some other DHT node. It then informs its two direct neighbors that it is joining.

Storage. FlexiFS uses consistent hashing to assign blocks to nodes [13] with replication factor r: a block with a key k is stored at the r nodes whose identifiers follows k on the ring.

Failure Detection. Each node periodically checks the availability of its closest successor on the ring, and repair mechanisms are triggered upon a lack of response within a timeout.

A gossiping mechanism spreads topological changes throughout the ring. Each node notifies its closest neighbor whenever it learns about a leave/join event. If the time to spread a message along the ring is shorter than the time between two leave/join events, this mechanism is guaranteed to maintain the ring topology. In our experience, such an assumption is reasonable for a deployment size of less than a few hundred storage nodes (our typical testbed size).[2]

2.3 Filesystem

Like most contemporary DFSS, FlexiFS decouples metadata from data storage. For each file, an inode block (`iblock` hereafter) contains the metadata information about the file, e.g., size and user/group ownership. One or more data blocks (`dblock`) hold the content of the file.

FlexiFS provides several hooks to tune how files are stored. Figure 2 illustrates our current design: `dblocks` are of constant and configurable size. The current size of `dblocks` is 128 kB, corresponding to the default maximal block size for the FUSE interface. `iblocks` simply list the `dblocks` of the corresponding files. Compared to the typical redirection-based architecture of Unix, the two above mechanisms help in reducing the network overhead [5].

Both `iblocks` and `dblocks` are represented as elements of the same key/value store, where they get replicated according to the different consistency models.

[2] The probability that faults partition the ring is small. Indeed, we note that at the considered scale, the mean time between failures divided by the number of nodes is orders of magnitude smaller than the time to spread a message throughout the ring.

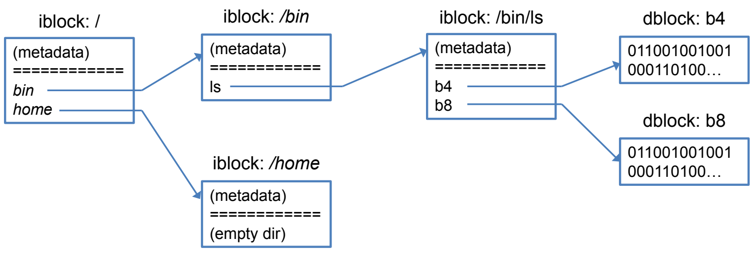


Fig. 2. Example of a filesystem structure stored in FlexiFS

```
int(∗ create )(const char ∗, mode_t, struct fuse_file_info ∗);
int(∗ open )(const char ∗, struct feuse_file_info ∗);
int(∗ read )(const char ∗, char ∗, size_t, off_t, struct fuse_file_info ∗);
int(∗ write )(const char ∗, const char ∗, size_t, off_t, struct fuse_file_info ∗);
int(∗ close)(const char ∗);
int(∗ rename )(const char ∗, const char ∗);
int(∗ statfs )(const char ∗, struct statvfs ∗);
```

Fig. 3. Excerpt of the FUSE interface implemented by FlexiFS

Only `iblocks` are mutable. The key of a `dblock` is equal to the hash of its content. This ensures good balancing of the data across storage nodes in order to deliver aggregate performance and increased fault tolerance. In case of an `iblock`, the proxy generates a unique key at creation time.

2.4 File Operations

FlexiFS implements the complete FUSE interface. We present an excerpt in Figure 3 and detail below the most important operations.[3]

Create. Upon the creation of a file (or directory), the proxy first stores a corresponding `iblock` in the distributed storage. Then, it executes *C&S*() on the parent directory to add the file. Performing a *C&S*() operation ensures that no two clients create the same file concurrently. If the file was concurrently created, the proxy returns an error to the client.

Open. To open an existing file (or directory), the proxy follows the graph structure of the filesystem and invokes the *get*() operation to retrieve the corresponding `iblock` from the storage interface. Once the `iblock` is fetched, the proxy checks that permissions are correct and returns an appropriate value to the client.

Read. The proxy first retrieves the `iblock` from the storage system. Once the `iblock` is known, the proxy also knows all the `dblocks` attached to it. Hence, to retrieve the content of the file, the proxy fetches the `dblocks` from the storage system by invoking the *get*() operation in parallel.

[3] Because an open file may acquire multiple reference (e.g., after a *fork*()), there is no *close*() operation in the FUSE interface, but instead *flush*() and *release*(). The former is called every time a descriptor referencing the file is closed; the latter once all file descriptors are closed and all memory mappings are unmapped. To simplify exposition, we shall consider in this paper that *close*() is part of the FUSE interface.

Write. The proxy first retrieves the `iblock` of the file and produces the new `dblocks`. It uses the storage layer's $put()$ operation to insert (in parallel) the new `dblocks` in the distributed storage. Notice that because `dblocks` are content-addressed and immutable, every modification that produces a `dblock` leads to the creation of a new `dblock` with a different key. Then, the proxy uses $C\&S()$ to update the `iblock` corresponding to a file. If the `iblock` changed meanwhile, the proxy has to recompute (if necessary) the `dblocks`, as well as an updated version of the `iblock`; then it re-executes $C\&S()$. This last sequence of operations is executed until the $C\&S()$ succeeds. Since a $write()$ operation may access any offset of the file, the above mechanism is necessary to avoid a lost update phenomena when two clients concurrently write the file.

Close. Upon the closing of a file, the proxy checks that the file still exists. If the file does not exist, the proxy returns an error to the client.

Rename. If the source and target parent directories are the same, the proxy attempts updating the `iblock` of the parent directory. In case they are different, the proxy first tries adding the file to the target directory, then it attempts removing the file from the source directory. All attempts are perform with $C\&S()$. If $C\&S()$ fails at some point, the proxy returns an error to the client.[4]

3 Consistency

An important design aspect for a DFSS is defining the semantics of sharing, i.e., how clients accessing simultaneously the same file observe modifications by other clients. FlexiFS has several built-in sharing semantics and corresponding implementations, which we describe in the remainder of this section. Additional semantics can be easily added thanks to FlexiFS's modular design.

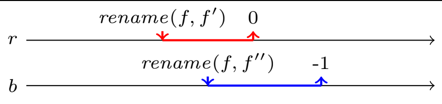
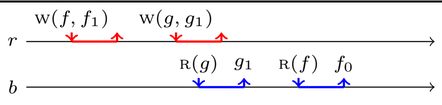
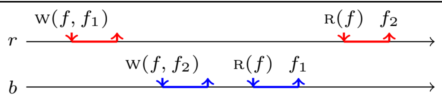
3.1 Overview

FlexiFS classifies the semantics of sharing with (i) the consistency level that governs the FUSE interface; and (ii) the use (or not) of the close-to-open semantics. The combination of these two parameters defines a filesystem consistency level (FSCL). In what follows, we list the various consistency levels FlexiFS supports at the FUSE interface and their respective implementation. Further, we introduce the close-to-open semantics.

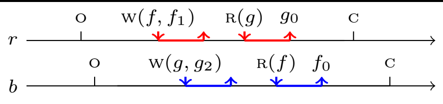
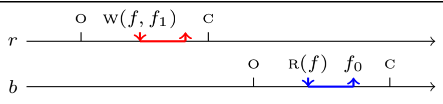
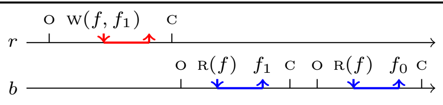
3.2 Consistency of the FUSE Interface

The three operations available at the storage layer implement accesses to the filesystem. As a consequence, the consistency of the distributed storage governs consistency of the FUSE interface. The FUSE interface is linearizable (resp. sequentially, eventually consistent) when operations at the storage level are linearizable (resp. sequentially, eventually consistent).

[4] Even if $C\&S()$ is atomic, the renamed file might end up in both source and target directories. Renaming is strictly atomic in POSIX semantics. We note however that such a behavior is admissible in certain systems (e.g., Win32).

Consistency of File Operations	Example	Implementations
Linearizability		POSIX [2]
Sequential Consistency		Sprite [14], GoogleFS [5], HDFS [4]
Eventual Consistency		Pastis [11]

(a) *Without* close-to-open semantics

Consistency of File Operations	Example	Implementations
Linearizability	 O W(f, f_1) R(g) g_0 C r O W(g, g_2) R(f) f_0 C b	AFS [3], NFS [15]
Sequential Consistency		SinfoniaFS [16]
Eventual Consistency		Coda [17], Ivy [10]

(b) *With* close-to-open semantics

Fig. 4. Filesystem Consistency Levels. *(operation* W(f, v) *means a write to file* f *with value* v*;* R(f) *is a read on* f*;* O *and* C *respectively opens and closes all files accessed during the session).*

Linearizability. The most powerful synchronization level for processes in a distributed environment is obtained through the use of atomic, or *linearizable*, objects [18]. A linearizable object is a shared object that provides the illusion of being accessed locally. More precisely, this consistency level states that each operation takes effect instantaneously at some point in real time, between its invocation and response.

Figure 4(a) presents an execution of linearizable operations. The blue (b) client renames file f to f'. Concurrently, the red (r) client renames file f to f''. Since operations are linearizable, one of the two accesses must fail.

To implement linearizability in FlexiFS, we use Paxos [19], which provides a consensus primitive for unreliable nodes. On top of consensus, we implement a replicated state machine executing the three operations listed in Section 2.1. Notice that, because `dblocks` are immutable, they are trivially linearizable. Hence, to improve performance, we execute a simpler algorithm in that case: operations *put*() and *get*() access a majority of replicas, respectively storing and fetching the content from it.

Sequential Consistency. Under sequential consistency, "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [20]. Sequential consistency is weaker than linearizability. In particular, this consistency level is not composable [18]: even if each file is sequentially consistent, the filesystem as a whole is not sequentially consistent (this is also called the *hidden channel* problem). We illustrate this issue in Figure 4(b) (middle). In this figure, accesses with respect to file f are sequentially consistency, and similarly this property holds for g. However, the execution (i.e., when we consider both f and g as a whole) is not sequentially consistent.

FlexiFS implements sequential consistency using primary replication. For each `iblock`, a primary replica is elected. Upon a $put(k, v)$ call, the primary for key k sends to all replicas the value v and then waits until a majority of replicas acknowledges the reception before returning to the proxy. To execute a $get(k)$ call, the proxy accesses any replica of k that contains the version it previously read, or a newer version (this applies only to `iblocks`). To execute $C\&S(k, u, v)$, the primary tests locally if the old value equals u. If it is the case, it executes a $put(k, v)$ and returns the old value to the proxy. A perfect failure detector [21] ensures the safety and liveness of the above mechanisms.

Eventual Consistency. Under eventual consistency [22], there must exist a monotonically growing prefix of updates on which correct replicas agree. Since there is no assumption on the time of convergence, eventual consistency does not offer any guarantee on the return value of non-stable operations (that do not belong to the common prefix).

Figure 4(b) (bottom) depicts a run under eventual consistency. In this figure, both b and r clients write then read file f. Because the r client reads version f_2 while the b client reads version f_1, no linearization of the four operations can satisfy the returned values.

We implement eventual consistency in FlexiFS using version vectors and the "last writer wins" approach [23]. This optimistic replication schema works as follows: Each version of the `iblock` is timestamped with a version vector. Upon updating the value of an `iblock` (via $put()$ or $C\&S()$), the proxy contacts one of the `iblock` replicas. This replica atomically increments its local vector clock, timestamps the `iblock` with it, and returns to the proxy. Replicas then converge using an anti-entropy protocol. If two versions of some `iblock` are concurrent, we apply the "last writer wins" approach. Concurrent operations are totally ordered according to the identifier of the replicas that emitted them. Upon a read access, a proxy simply returns a version stored at some replica.

3.3 Close-to-Open Semantics

Under POSIX semantics, almost all file operations shall be linearizable [2, page 58]. In particular, a read shall see the effects of all previous writes performed on the same file. The CAP impossibility result [1] tells us that such a constraint hinders the scalability of a DFSS.

By introducing the notion of *file session*, close-to-open semantics [24] aim at reducing the amount of synchrony required to access a shared file. A file session is a sequence of read and/or write operations enclosed between an *open* and a *close* invocation. It has the following properties [25]:

(1) Writes to an open file are visible to the client but are invisible to remote clients having the same file opened concurrently.
(2) Once the file is closed, changes are immediately visible to sessions that are starting afterwards.

Since operations execute in isolation and either all writes or none execute, these sharing semantics are close to the familiar notion of transaction. Notice, however, that close-to-open semantics apply the last writer wins rule: two concurrent updates do not abort, one of them is simply overwritten.

The above definition of close-to-open consistency has been formulated with atomicity in mind. One can actually combine close-to-open semantics with sequential consistency and eventual consistency as well. For sequential consistency, rule (2) is replaced by:

(2a) There exists a sequential ordering of the sessions such that (i) for every read in a session, there exists a matching write prior to it, and (ii) reads are causally ordered on the same client.

For eventual consistency, rule (2) is replaced by:

(2b) If at some point in time no more changes occurs, then eventually all sessions observe the same state of the file.

Figure 4(b) illustrates the combination of close-to-open semantics with linearizability, sequential, and eventual consistency. Obviously, if a single read or write operation is executed per session, the consistency level per operation defines how sessions behave. In other words, close-to-open semantics have no effect (e.g., middle row in Figure 4(b)). Now, when a client executes multiple operations or opens multiple files at the same time, the filesystem is neither linearizable nor sequentially consistent. For instance, execution depicted at the top row in Figure 4(b) is admissible. Under close-to-open semantics, linearizability is stricter than sequential consistency, which is itself stricter than eventual consistency (last two rows in Figure 4(b)).

When FlexiFS implements the close-to-open semantics, the proxy is stateful and keeps track of the files opened by each of its clients. Therefore, a client has to access the same proxy during a file session. In more details, upon a successful call to *open*(f) the proxy records the `iblock` of f. This `iblock` is used for all the operations during a file session: a *read*() operation accesses the `dblocks` indexed by the `iblock`, and a write operation changes only the cached version. When the client closes file f, the proxy stores the `iblock` of f using *put*(). It can then forget that f was opened by the client.

3.4 Consistency of the Filesystem

A filesystem consistency level (FSCL) is obtained by the combination of a consistency level governing file operations and the use or not of the close-to-open semantics. This leads to six *different* FSCLs. In Figure 4, we list in the last column one or more matching implementations for each level.

POSIX semantics is obtained when file operations are atomic and close-to-open is not supported. To implement sequential consistency, Sprite [14] relies on a cache consistency manager while GoogleFS [5] and HDFS [4] make use of a leasing mechanism. NFS [15] implements the consistency level offered in Andrew Filesystem [3]: the close-to-open semantics is respected and metadata operations are atomic. Sinfonia [16] supports mini-transactions, a generalized form of compare-and-swap operation. To advocate for this paradigm, the authors of Sinfonia built a filesystem: SinfoniaFS. This filesystem implements sequential consistency with close-to-open semantics. Ivy [10] and Pastis [11] implement an eventually consistent DFSS, respectively, with and without close-to-open semantics.

4 Evaluation

In this section, we present experimental results obtained using FlexiFS, where we observe empirically and in isolation the tradeoffs between sharing semantics and performance in DFSS designs.

4.1 Experimental Settings

All tests were performed on a cluster of 8-core virtualized Xeon 2.5 Ghz servers running Ubuntu 12.04 GNU/Linux and connected by a 1 Gbps switched network. We use 3 to 7 servers for the storage layer and one client. Our implementation uses the Lua programming language (`http://www.lua.org/`) and leverages the Splay framework and libraries [26]. Bindings to the FUSE C APIs employ the `luafuse` library (`http://code.google.com/p/luafuse/`). The FlexiFS implementation is modular and easy to modify. The conciseness of Lua and the use of Splay allow the whole implementation to be less then 2,000 lines of code (LOC). In particular, the code to support each FSCL is very concise and easy to extend, e.g., 62 LOC for sequential consistency, and 160 LOC for eventual consistency.

In what follows, we explore the impact of each FSCL on the cost of `iblock` operations, and file operations. We also investigate the impact of the replication factor on performance. All experimental results are averaged over 10^3 operations, and we present standard deviations when appropriate.

4.2 Benchmarks

Metadata Operations. In Figure 5(a), we experiment the insertion of a novel `iblock` in the storage system when both the FSCL and the size of the inserted block vary. For the sake of comparison, results are normalized by the time required for executing a *noop*() RPC operation carrying a payload that equals the size of the block.

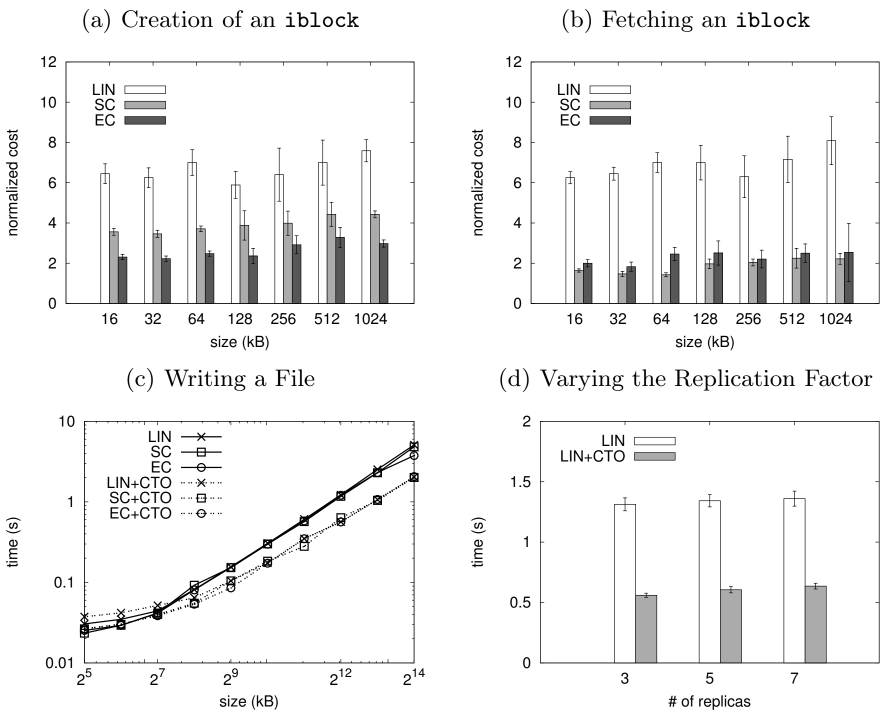


Fig. 5. Evaluating the Price of Consistency. *(LIN, SC and EC stand for Linearizable, Strong Consistency and Eventual Consistency FSCL models, respectively. CTO stands for close-to-open semantics).*

We observe in Figure 5(a) that eventual consistency is the cheapest FSCL as it costs around 2 times more than the baseline RPC call. This is expected since a call to *C&S*() under eventual consistency requires 2 roundtrips: one to go from the client to the proxy, then one to go from the proxy to a replica. Sequential consistency costs 4 roundtrips: once the primary is reached, the update must reach a quorum of replicas. For linearizability, 2 more roundtrips for the "propose" phase of the Paxos algorithm are executed, leading to 6 times the baseline cost.

Our second experiment evaluates the cost of fetching an `iblock` from the file storage. We report the results in Figure 5(b). Under linearizability, a *get*() operation has an identical cost to a *C&S*() operation, since both operations go through the replicated state machine. On the other hand, the cost of sequential consistency is reduced because the proxy can access any replica to fetch the `iblock` content. Therefore, performance is in that case identical to eventual consistency.

File Operations. Figure 5(c) depicts the time required to write a complete file at the client side using the FUSE interface. Both scales in this figure are logarithmic. Eventual consistency is the fastest FSCL when either *(i)* close-to-open semantics is not used, or *(ii)* there is a single `dblock` to write, i.e., less than 128 kB in our settings. The POSIX semantics of sharing (i.e., linearizability without close-to-open semantics) performs the worst in this respect. When the size of the file reaches 128 kB, the use of close-to-open semantics leads to better performance (between 2 and 3 times faster). Below 128 kB, close-to-open semantics pays the cost of the necessary *open*() and *close*() operations. All FSCLs offering close-to-open semantics reach similar performance when more than 4 MB of data are written. Read operations over a file (not reported here) follow a similar pattern.

Impact of the Replication Factor. Our last experiment measures the impact of the replication factor on performance. In this experiment, a client writes a file of 4 MB under linearizability. We vary both the replication factor of FlexiFS, and the use or not of the close-to-open semantics. Figure 5(d) depicts our results. In this figure, we observe that increasing the replication factor has a small impact on performance: below 3% without close-to-open semantics, and 7% with. Paxos is the most demanding consistency control algorithm we have implemented. Thus, this result shows that the filesystem consistency level is contributing more than the replication factor to DFSS performance.

5 Related Work

Several papers discuss the performance, consistency, and semantics tradeoffs in DFSS designs. The Andrew file system (AFS) [3] introduced caching mechanisms and the close-to-open semantics for both files and directories. This was inspired by earlier designs such as Locus [27], which relied on a strict—but costly and inefficient—POSIX semantics. Since its second version, the Network File System (NFS) [15] also implements the close-to-open semantics; its fourth version distinguishes data from metadata management, a separation that has been adopted by all DFSS designs since then.

OceanStore [28] is a flat data storage system that provides both eventually consistent and atomic operations. OceanStore follows a design close to the eventually serializable data storage [22]: an application may emit two types of file operations, *weak* and *strong*. A weak operation is tentative and executes on any replica; a strong operation waits until replicas agree on some total ordering of the operations. GoogleFS [5] uses a central server for storing metadata. CFS [9] builds a single-user file system by storing content-addressable blocks in the Chord DHT [29]. Ivy [10] extends this design by allowing a predefined group of users to access a shared file system. Content blocks are stored in the Chord DHT and each writer maintains its own modification log, implementing read-your-write semantics and eventual consistency. In [30], the authors propose to build centralized metadata storage services for DFSS, providing linearizability guarantees using the Paxos [19] consensus algorithm while maintaining high availability.

The authors of Pastis [11] compare close-to-open against read-your-write semantics. Levy [25] surveys DFS designs and four different types of file sharing semantics: POSIX, close-to-open, immutable files, fully transactional semantics, and survey corresponding implementations. The use of a modular framework for evaluating design choices and establishing performance/tradeoffs in systems software design has been successfully used in various domains. Examples include virtual machines construction [31] or CORBA-based Middleware [32].

6 Conclusion

This paper depicts a study of the impact of a filesystem consistency level (FSCL) on the performance of a distributed filesystem service (DFSS). While the FSCL offered by a DFSS has fundamental impact on performance, it is difficult to systematically evaluate this impact in isolation from other design aspects, due to the design and implementation diversity of existing systems. This paper presents FlexiFS, a framework for the systematic evaluation of DFSS aspects. In more details, we depict a filesystem interface to users and leverage a set of servers implementing a fully distributed storage layer for both data and metadata. We implement three forms of consistency: linearizability, sequential consistency and eventual consistency, together with and without close-to-open semantics. Remarkably, a DFSS providing all these FSCLs can be supported with the simple addition of a *compare-and-swap* primitive to a regular key/value store. Our experimental results establish that linearizability under the close-to-open semantics is a sound design choice and a good compromise between operational semantics and performance, while illustrating the tradeoffs offered by the other design options. FlexiFS has a modular design and we plan on investigating further aspects of DFSS pertaining to indexing, client interaction, and semantics. We also plan on releasing FlexiFS as a part of the open-source SPLAY framework [26].

Acknowledgements. This work is sponsored in part by European Commission's Seventh Framework Program (FP7) under grant agreement No. 318809 (LEADS), and the Swiss National Foundation under agreements No. 200021-127271/1 and CRSII2-136318/1.

References

1. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51–59 (2002)
2. IEEE, The Open Group: Standard for Information Technology-Portable Operating System Interface (POSIX) System Interfaces (2004)
3. Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J.: Scale and performance in a distributed file system. ACM Trans. Comput. Syst. 6(1) (February 1988)
4. The Apache Software Foundation: The Hadoop Distributed File System (2012)
5. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. In: SOSP (2003)
6. Shvachko, K.V.: HDFS Scalability: The Limits to Growth. USENIX login 35(2) (April 2010)
7. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44(2) (April 2010)
8. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: SOSP (2007)

9. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. In: SOSP (2001)
10. Muthitacharoen, A., Morris, R., Gil, T.M., Chen, B.: Ivy: a read/write peer-to-peer file system. In: OSDI (2002)
11. Busca, J.-M., Picconi, F., Sens, P.: Pastis: A highly-scalable multi-user peer-to-peer file system. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 1173–1182. Springer, Heidelberg (2005)
12. File System in User Space, http://fuse.sourceforge.net/
13. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: STOC (1997)
14. Nelson, M.N., Welch, B.B., Ousterhout, J.K.: Caching in the sprite network file system. ACM Trans. Comput. Syst. 6(1) (February 1988)
15. Sun Microsystems, Inc.: NFS: Network file system protocol specification. RFC 1094, Network Information Center, SRI International (March 1989)
16. Aguilera, M.K., Merchant, A., Shah, M., Veitch, A., Karamanolis, C.: Sinfonia: a new paradigm for building scalable distributed systems. In: SOSP (2007)
17. Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.: Coda: A highly available file system for a distributed workstation environment. IEEE Trans. Comput. 39(4) (April 1990)
18. Herlihy, M., Wing, J.: Linearizability: a correcteness condition for concurrent objects. ACM Trans. Prog. Lang. and Sys. 12(3) (July 1990)
19. Lamport, L.: The part-time parliament. ACM Trans. Comp. Sys. 16(2) (1998)
20. Lamport, L.: How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Trans. Comput. 46(7), 779–782 (1997)
21. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996)
22. Fekete, A., Gupta, D., Luchangco, V., Lynch, N., Shvartsman, A.: Eventually-serializable data services. Theoretical Computer Science 220 (1999)
23. Saito, Y., Shapiro, M.: Optimistic replication. Computing Surveys 37(1) (2005)
24. Saltzer, J.H., Kaashoek, M.F.: Principles of Computer System Design: An Introduction. Morgan Kaufmann Publishers Inc. (2009)
25. Levy, E., Silberschatz, A.: Distributed file systems: concepts and examples. ACM Comput. Surv. 22(4) (December 1990)
26. Leonini, L., Rivière, E., Felber, P.: SPLAY: Distributed systems evaluation made simple. In: NSDI (2009)
27. Walker, B., Popek, G., English, R., Kline, C., Thiel, G.: The LOCUS distributed operating system. In: SOSP (1983)
28. Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.: Oceanstore: An architecture for global-scale persistent storage. In: ASPLOS (2000)
29. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans. Net (1) (February 2003)
30. Stamatakis, D., Tsikoudis, N., Smyrnaki, O., Magoutis, K.: Scalability of replicated metadata services in distributed file systems. In: Göschka, K.M., Haridi, S. (eds.) DAIS 2012. LNCS, vol. 7272, pp. 31–44. Springer, Heidelberg (2012)
31. Geoffray, N., Thomas, G., Lawall, J., Muller, G., Folliot, B.: VMKit: a substrate for managed runtime environments. In: VEE (2010)
32. PolyORB Middleware Technology, http://libre.adacore.com/tools/polyorb

An Effective Scalable SQL Engine for NoSQL Databases

Ricardo Vilaça, Francisco Cruz, José Pereira, and Rui Oliveira

HASLab - High-Assurance Software Laboratory
INESC TEC and Universidade do Minho
Braga, Portugal
{rmvilaca,fmcruz,jop,rco}@di.uminho.pt

Abstract. NoSQL databases were initially devised to support a few concrete extreme scale applications. Since the specificity and scale of the target systems justified the investment of manually crafting application code their limited query and indexing capabilities were not a major impediment. However, with a considerable number of mature alternatives now available there is an increasing willingness to use NoSQL databases in a wider and more diverse spectrum of applications and, to most of them, hand-crafted query code is not an enticing trade-off.

In this paper we address this shortcoming of current NoSQL databases with an effective approach for executing SQL queries while preserving their scalability and schema flexibility. We show how a full-fledged SQL engine can be integrated atop of HBase leading to an ANSI SQL compliant database. Under a standard TPC-C workload our prototype scales linearly with the number of nodes in the system and outperforms a NoSQL TPC-C implementation optimized for HBase.

Keywords: SQL, NoSQL, Cloud Computing, Middleware.

1 Introduction

With cloud-based databases as part of platform-as-a-service offerings, such as Google's BigTable [7], Amazon's DynamoDB [10], and Yahoo!'s PNUTS [8], HBase [12] and Cassandra [19], NoSQL databases become attractive for a larger and more diverse set of applications.

However, their lack of SQL support represents a major hurdle for a wider adoption. This arises at different levels due to the prevalence of SQL as the standard, widely mastered and efficient query language for databases. Most web scale applications are purposely kept SQL-based for their core data management [23]. Any application at least two, three years old is directly or indirectly (e.g. on top of an object-relational mapping) based on an SQL interface and its migration is usually not straightforward. A large number of tools and middleware coupled to SQL have been developed and matured over the years and are currently at the basis of most application development frameworks.

It is therefore not surprising that SQL compliance has been one of the most requested additions to the Google App Engine platform [15].

This state of affairs has sparked a number of proposals for middleware that exposes higher-level query interfaces on top of the barebones key-value primitives of NoSQL databases. Many of these aim at approximating the traditional SQL abstraction, ranging from shallow SQL-like syntax for simple key-value queries (e.g. CQL for Cassandra, Phoenix [24], PIQL [4]) to the translation of analytical queries into map-reduce jobs [1–3, 20]. However, due to the complexity of SQL existing solutions are limited to a subset of the language, thus not allowing to leverage exiting SQL applications and tools.

In this paper, we present a *distributed query engine* (DQE) for running SQL queries on top of a NoSQL database, while preserving its scalability and schema flexibility. The DQE allows to combine the expressiveness and performance of a Relational Database Management System (RDBMS) with the scalability and schema flexibility of a NoSQL database. DQE is a transaction-less database [17], preserving the isolation semantics provided by the underlying NoSQL database.

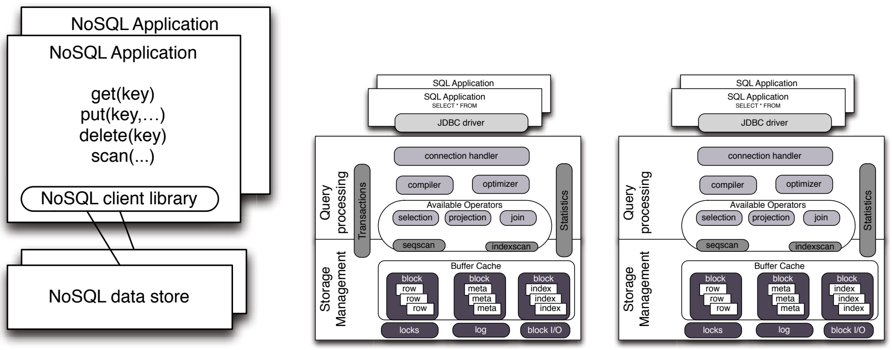
Enabling scalable SQL processing atop of a NoSQL database poses several architectural challenges to the query engine [26, 27]. On one hand, traditional RDBMS architectures include several legacy components such as on-disk data structures, log-based recovery, and buffer management, that were developed years ago but are not suited to modern hardware. Those components impose an huge overhead to transaction processing [17] limiting its scalability. On the other, large scale NoSQL databases have a simple data model, using a simple key-value store or at most variants of the entity-attribute-value (EAV) model [22] which strongly limit the expressiveness of data representation. Therefore, a first challenge consists in addressing the impedance mismatches [21] while supporting SQL queries. Moreover, it implies mapping relational tables and indexes to the NoSQL database tables in such way that the processing capabilities of the database are fully exploited. The other major challenge regards the basic key-value store interface of NoSQL databases which only allows applications to insert, query, and remove individual tuples or, at most, issue range queries based on the primary key of the tuple. These range queries allow for fast iteration over ranges of rows and also allow to limit the number and what columns are returned. However, NoSQL databases do not support partial key scans, but SQL index scans must perform equality and range queries on all or part of the fields of the index.

Contributions. This paper makes the following three contributions. First, we propose an architecture that allows to combine the expressiveness and performance of RDBMS with the scalability of a NoSQL database. The resulting query processing component is stateless regarding application data and can thus be seamlessly replicated and embedded in the client application. Then, we show how the basic key-value operations and data models can be mapped to scan operators within a traditional (SQL enabled) RDBMS. And finally, we describe a complete implementation of DQE with full SQL support, using Apache Derby's [11] query engine and HBase as the NoSQL database.

Roadmap. The remainder of the paper is structured as follows. Section 2 introduces the proposed architecture of the DQE. Section 3 describes how it is implemented using Apache Derby components and HBase. Section 4 presents the experimental evaluation. Section 5 compares our approach to other proposals for query processing on a NoSQL database, and Section 6 concludes the paper.

2 Architecture

The proposed architecture is shown in Figure 1(c), in the context of a scalable NoSQL and a traditional RDBMS. A major motivation for a NoSQL database is scalability. As depicted in Figure 1(a), a typical NoSQL database builds on a distributed setting with multiple nodes of commodity hardware. By adding more nodes to the system we can increase both the overall performance and capacity of the system and also its resilience through data replication. By allowing clients to directly contact multiple fragments and replicas, the system can scale also in terms of clients connected. To make this possible, NoSQL databases provide a simple data model as well as basic querying and searching capabilities, that allow applications to insert, query, and remove individual items or, at most, issue range queries based on the primary key of the item [28].



(a) NoSQL architecture. (b) Centralized SQL RDBMS. (c) Distributed query engine (DQE).

Fig. 1. Data management architectures

In sharp contrast, a RDBMS is organized as tables (also called relations). We seldom are concerned with the storage structure but instead express queries in a high-level language, invariably SQL. SQL allows applications to realize complex operations and processing capabilities, such as filtering, joining, grouping, ordering and counting.

Our current proposal builds on rewriting the internal architecture of a typical RDBMS, by reusing some of its components and adding new components atop of a NoSQL database. To understand how components can be reused we examine the internals of a RDBMS roughly splitting it in a *query processor* block and a *storage manager* block (Figure 1(b)).

The query processor is responsible for offering an SQL based API to applications, and for translating the application queries, through compilation and execution, to the underlying storage manager.

The architecture proposed in Figure 1(c) reuses a number of components from the SQL query processor (shown in light gray). In detail, these are: the JDBC driver and client connection handler; the compiler and the optimizer, and a set of generic relational operator implementations. These components can be shielded from changes, as they depend only on components that are reimplemented (shown in medium gray) providing the same interfaces as those that, in the RDBMS, embody the centralized storage functionality (shown in dark gray) and that is removed from our architecture. The components to be reimplemented are the following:

– A mapping from the relational model to the data model of a NoSQL database. This includes: atomic data types and their representation, representation of rows and tables, and representation of indexes.
– A mapping from the relational schema to that of the NoSQL database, which allows data to be interpreted as relational tables (see Section 3);
– Implementation of sequential and index scan operators. This includes: matching the interface and data representation of the database and leveraging the indexing and filtering capabilities in the NoSQL database to minimize data network traffic;

The proposed architecture has the key advantage of being stateless regarding application data. Data manipulation language (DML) statements (SELECT, INSERT, UPDATE and DELETE) can be executed without any coordination among different DQE instances. As a result, the system should retain the scale-out capabilities of the supporting NoSQL database.

In addition, this architecture also offers the possibility to take advantage of the flexible schema exposed by the underlying NoSQL database. That is, each application applies its own view of the schema over the NoSQL database.

3 Implementation

The current DQE prototype was built by reusing Apache Derby components and uses HBase as the NoSQL database. In the following, we start with an overview of both systems.

HBase is a key-value based distributed data storage system based on Bigtable [7]. In HBase, data is stored in the form of HBase tables (HTable) that are multi-dimensional sorted maps. The index of the map is the row's key, column's name, and a timestamp. Columns are grouped into column families. Column families must be created before data can be stored under any column key in that family. Data is maintained in lexicographic order by row key. Finally, each column can have multiple versions of the same data indexed by their timestamp.

A read or write operation is performed on a row using the row-key and one or more column-keys. Update operations on a single row are atomic, i.e. concurrent writes on a single row are serialized. Any update performed is immediately visible to any subsequent reads. HBase exports a non-blocking key-value interface on the data: put, get, delete, and scan operations.

HBase closely matches the scale-out properties assumed for NoSQL databases. HTables are horizontally partitioned in regions that are assigned to RegionServers nodes. In turn, each region is stored as an appendable file in the distributed file system, Hadoop File System (HDFS) [25] based on GFS [13]. By default, HBase uniformly distributes data among all available nodes in the systems.

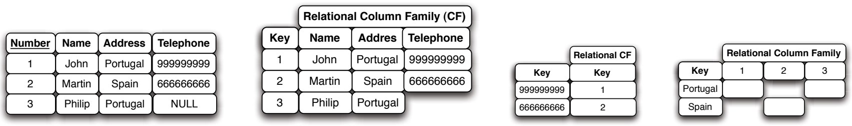
Apache Derby is an open source relational database implemented entirely in Java. Derby has a small footprint, about 2.6 megabytes for the base engine and an embedded JDBC driver. In addition, it is easy to install, deploy and use.

Besides providing a complete implementation of SQL and JDBC, Derby has the advantage of already providing an embedded mode, which eases its use as a middleware layer.

The store layer of Derby is split into two main areas, access and raw. The access layer presents a conglomerate (table or index)/row based interface to the SQL layer. It handles table scans, index scans, index lookups, indexing, sorting, locking policies, transactions, isolation levels. The access layer sits on top of the raw store, which provides the raw storage of rows in pages in files, transaction logging, transaction management. Following the architecture proposed in the previous chapter, the raw store layer was removed in our prototype and some components of the access layer were replaced.

3.1 Prototype Components

The system is composed of the following layers: (i) query engine, (ii) storage and (iii) file system. Applications issue SQL requests to any query engine node. The query engine node communicates with storage nodes, executes queries and returns the results to applications. We use Derby components to implement the query engine. We reuse its query processing sub-system, both the compiler and the optimizer components. Two new operators for index and sequential data scans have been added to the set of Derby's generic relational operators. These operators leverage HBase's indexing and filtering capabilities to minimize the amount of data that needs to be fetched. The SQL advanced operators such as JOIN and aggregations are not supported by HBase and are implemented at the query engine. The query engine translates the user queries into some appropriate put, get, delete, and scan operations to be invoked on HBase. Each HBase region is stored as an appendable file in the distributed file system. The distributed file system could be implemented using any scalable file system that supports append-only files such as HDFS.



	Relational Column Family (CF)		
Key	Name	Addres	Telephone
1	John	Portugal	999999999
2	Martin	Spain	666666666
3	Philip	Portugal	

Number	Name	Address	Telephone
1	John	Portugal	999999999
2	Martin	Spain	666666666
3	Philip	Portugal	NULL

	Relational CF
Key	Key
999999999	1
666666666	2

	Relational Column Family		
Key	1	2	3
Portugal			
Spain			

(a) Relational Table (b) Primary Key HTable (c) Unique Index HTable (d) Non-unique Index HTable

Fig. 2. Data Model Mapping

3.2 Relational-Tuple Store Mapping

Relational tables and secondary indexes are mapped to the HBase's data model. We have adopted a simple mapping from a relational table to a HTable. There is a one-to-one mapping where the HBase row's key is the relational primary key (simple or compound) and all relational columns are mapped into a single column family. Since relational columns are not multi-valued, each relational column is mapped to a HTable column. The schema of relational tables is rigid, i.e., every row in the same table must have the same set of columns. However, the value for some relational columns can be NULL and thus an HTable column for a given row exists only if its original relational column for that row is not NULL.

A secondary index of the relational model is mapped into an additional HTable. The additional table is necessary so that data is ordered by the indexed attributes. For each indexed attribute a HTable row is added and its row's key is the indexed attribute. For unique indexes the row has a single column with its value being the key of the matching indexed row in the primary key table. For non-unique indexes there is one column per matching indexed row with the name of the column being the matching row's key. Figure 2(a) depicts a relational table example. The column Number is the primary key and the table has two additional indexes: one unique index on attribute Telephone and a non-unique index on column Address. Therefore, the mapping will have three HTables: base data — Figure 2(b), unique index on column Telephone — Figure 2(c), and non-unique index on column Address — Figure 2(d).

Due to the simple mapping from the relational data model to HBase, the user can take advantage of the flexible schema exposed by the underlying NoSQL database and directly use its data in HBase for simple queries or complex map-reduce jobs.

3.3 Reducing Data Transfer

In order to reduce network traffic between the query engine and HBase, the implementation of sequential and index scan operators takes advantage of the indexing and filtering capabilities of HBase.

For index scans data is maintained ordered by one or more columns. This allows to restrict the desired rows for a given scan by optionally specifying the start and the stop keys. In a relational table each column is typed (e.g., char, date, integer, decimal, varchar) and data is ordered according to the natural order of the indexed column data type. However, row keys in HBase are plain byte arrays and neither Derby or HBase byte encoding preserve the data type's natural order. In order to build and store indexes in HBase maintaining the data type's order we need to map row keys into plain bytes in such a way that when HBase compares them the order of the data type is preserved. This mapping has been implemented for integer, decimal, char, varchar and date types. As indexes may be composite, besides each specific data type encoding, we also needed to define a way to encode multiple indexed columns in the same byte array. We do so by simply concatenating them from left to right, according to the order they are defined in the index using a pre-defined separator.

In HBase the start and stop keys of a scan must always refer to all the columns defined in the index. However, when using compound indexes the DQE may generate scans using subsets of the index columns. Indeed, an index scan can use equality conditions on any prefix of the indexed columns (from left to right) and at most one range condition on the rightmost queried column. In order to map these partial scans, the default start and stop keys in HBase are not used but instead the scan expression is run through HBase's BinaryPrefixComparator filter.

The aforementioned mechanisms reduce the traffic between the query engine and HBase by only retrieving the rows that match the range of the index scan. However, a scan can also select non-indexed columns. A naive implementation of this selection would fetch all rows from the index scan and test the relevant columns row by row. In detail, doing so on top of HBase would require a full table scan, which means fetching all the table rows from the different regions and possible different RegionServers. The full table would therefore be brought to the query engine instance and only then discard those rows not selected by the filter. To mitigate this performance overhead, particularly for low selective queries, that this approach may incur, the whole selection is pushed down into HBase. This is done by using the SingleColumnValueFilter filter to test a single column and, to combine them respecting the conjunctive normal form, using the FilterList filter. The latter represents an ordered list of filters that are evaluated with a specified boolean operator FilterList.Operator.MUST PASS ALL (AND) or FilterList.Operator.MUST PASS ONE (OR).

3.4 Metadata

Derby must also store information about the in-memory representation of tables and index, conglomerates, in a persistent manner.

For this we use a special HBase table, ConglomerateInfo, with information for all available conglomerates. ConglomerateInfo has a single column family, MetaInfo, and data for each conglomerate is stored in a row whose key is the conglomerate's identifier. Three columns are used to save information: value, a byte array with the encoding of all information; name, the name of the index or table to which this conglomerate matches; and size, to store the estimate of the current size (number of rows) of the conglomerate.

The information stored in ConglomerateInfo is mainly modified by Data Definition Language (DDL) statements. However, the size attribute is modified by any update DML statement (INSERT, UPDATE or DELETE) and therefore this attribute may change frequently. Thus, we used a simple mechanism to update the value in a distributed and efficient manner. The size of the table/index is used only for the query optimizer to decide the best query plan and therefore don't need to be the most recent value. In most RDBMS, this value is maintained probabilistically manner and thus we update its value in each query engine instance asynchronously.

Each query engine instance maintains the last estimate it has for the global size (shared by all instances), updates it accordingly to the local changes (when some update DML statement occurs), and periodically updates the value stored in ConglomerateInfo (shared by all instances) and refresh its local estimate. For, this it maintains the delta of the size after the last update to ConglomerateInfo, and in the next update it increments or decrements the size stored in ConglomerateInfo with the delta value, using a special HBase method for this purpose (incrementColumnValue). Then, it resets the delta value for the next time window.

4 Evaluation

The evaluation of our prototype addresses two performance aspects: the overhead imposed by the DQE in terms of added latency, and the system scale-out in terms of the achieved throughput by increasing the number of nodes.

4.1 Overhead

We measure the increased latency of the system resulting from using the DQE instead of a standard HBase client.

Test Workload. We used a workload typical of NoSQL databases, Yahoo! Cloud Serving Benchmark [9]. YCSB was designed to benchmark the performance of NoSQL databases under different workloads. It has client implementations for several NoSQL databases and a JDBC client for RDBMS. We have used the HBase and JDBC clients without any modifications.

Experimental Setting. The machines used for the experiments had a 2.4 GHz Dual-Core AMD Opteron(tm) Processor, with 4GB of RAM and a local SATA hard-disk. The machines were interconnected by a switched Gigabit local area network.

For the experiments 2 machines were used: one to run the workload generator, using an embedded connection to the modified Derby; and another running HBase. HBase was run in standalone mode, meaning that the HBase master and HBase RegionServer were collocated in the same machine using the local filesystem.

Table 1. Overhead results (ms)

Workload	1 thread, 100 tps		50 threads, 100 tps	
Operation	HBase	DQE	HBase	DQE
Insert	0.58	0.93	1.04	1.98
Update	0.51	1.3	2.66	3.1
Read	0.53	0.79	1.63	1.7
Scan	1.43	2.9	4.64	6.1

The YCSB database was populated with 100,000 rows (185MB) and the workload consisted of 1,000,000 operations. The proportion for the different types of operations was 60% reads, 20% updates and 20% scans. The operations were distributed uniformly over the database rows. The size of the scan operator was also a uniform random number between 1 and 10. We measured two runs where each client had 1 and 50 threads, respectively. In both, the target throughput was 100 operations per second.

Results. The average latency (in milliseconds) for the YCSB workload is shown in Table 1, for the standard HBase client and DQE.

The results for the insert operations correspond to the loading of the database while other operations are due to the mix of operations generated by the workload itself.

The results show that for all types of operations the query engine can be embedded in the application with an overhead totally in line with the base figures. The additional latency is due to the SQL processing and required marshalling/unmarshaling. Moreover, the overhead of DQE decreases with the increasing number of concurrent clients (threads).

4.2 Scale-Out

We measured the increased throughput of the system when varying the number RegionServer nodes from 1 to 30.

Test Workload. For the evaluation of the scale-out we used the load of an industry standard on-line transaction processing SQL benchmark, TPC-C.[1] It mimics a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. The warehouses are hotspots of the system and the benchmark defines 10 client per warehouse.

TPC-C specifies five transactions: NewOrder with 44% of the occurrences; Payment with 44%; OrderStatus with 4%; Delivery with 4%; and StockLevel with 4%. The NewOrder, Payment and Delivery are update transactions while the others are read-only. The traffic is a mixture of 8% read-only and 92% update transactions and therefore is a write intensive workload.

[1] Since the current system does not include a transaction manager, all transactional contexts of the benchmark are simply discarded.

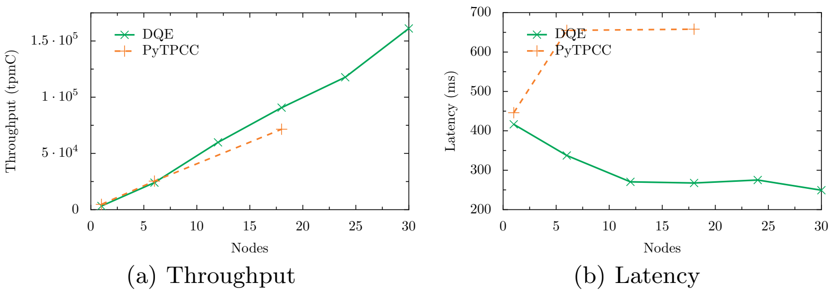


Fig. 3. TPC-C results

We have used both an existing SQL implementation,[2] without modifications, to drive the DQE, and an existing NoSQL implementation optimized for HBase.[3] Briefly, in the NoSQL implementation, TPC-C columns are grouped into column families, named differently for optimization, and the data storage layout has been optimized.

Experimental Setting. We ran all the experiments on a cluster of 42 machines with 3.10GHz GHz Dual-Core i3-2100 CPU, with 4GB of RAM and a local SATA hard-disk. The machines were interconnected by a switched Gigabit local area network.

The TPC-C workload has run from a varying number of machines. For our proposal, we varied the number of client machines from 1 to 10, each running 150 client threads. Each client machine as also running a DQE instance as a middleware layer.

One machine was used to run the HDFS namenode, HBase Master and Zookeper [18]. The remaining machines were RegionServers, each configured with a heap of 3GB, and also running a HDFS DataNode instance.

The TPC-C database was populated according to the number of Region-Servers, ranging from 5 warehouses, for a single RegionServer, to 150 warehouses, for 30 RegionServers. All TPC-C tables, were partitioned and distributed so there were 5 warehouses per RegionServer each handling a total of 50 clients. With 150 warehouses, the size of the database was about 75GB.

Results. The system throughput for a varying configuration of 1, 6, 12, 18, 24, and 30 RegionServers is depicted in Figure 3(a). The results show that DQE presents linear scalability. This is mainly due to the scale independence of the query processing layer and the scalability of the NoSQL database layer.

Furthermore, while with the DQE the system has a slightly lower throughput up tp 6 RegionServers its scalability is better than that of HBase under the NoSQL TPC-C load generator. This can be mainly attributed to the optimizations achieved by Derby's query engine that take advantage of relational

[2] BenchmarkSQL - `http://sourceforge.net/projects/benchmarksql/`

[3] `https://github.com/apavlo/py-tpcc/wiki/HBase-Driver`

operators, filtering and secondary indexes, while manual optimizations and denormalization still incur on greater complexity resulting in a greater overhead and worse scalability.

In this specific case, the query engine can greatly restrict the amount of data retrieved from HBase by, as previously explained, also taking advantage of HBase filters to select non-indexed columns. As a matter of fact, the network traffic when using the DQE is much lower than using the NoSQL TPC-C implementation. In fact, this prevented us from getting results for PyTPCC, the implementation optimized for HBase, with more than 18 RegionServers due to the saturation of the network. The latency figures in Figure 3(b) reflect the problem very clearly.

5 Related Work

Existing NoSQL databases rely on simplified and heterogenous query interfaces, that constitute a barrier on their adoption. Projects like BigQuery [1], Hive [2] or Tenzing [20] try to mitigate this constraint by providing an interface based on SQL over a MapReduce framework and a key/value store, but are mainly intended for data warehousing and analytical purposes.

BigQuery is a Google web service, built on top of BigTable. As query language, it uses a SQL dialect, that is a variation of the standard. It only offers a subset of the standard operators like selection, projection, aggregation and ordering. Joins and other more complex operators are not supported. In addition, data is immutable once uploaded to BigTable. Hive is built on top of Hadoop, a project that encompasses HDFS and the MapReduce framework. Hive also defines a simple SQL-like query language to query data but it offers more complex operators such as equi-joins, which are converted into MapReduce jobs, and unions. Likewise, Tenzing relies on a MapReduce framework to provide a SQL query execution engine, offering a mostly complete SQL implementation. The Hadapt commercial system[4] (previously HadoopDB [3]) is also an analytical driven database, but it takes a slightly different approach by providing a hybrid system. Like Hive, it uses an SQL interface over the MapReduce framework from Hadoop, but replaces the HDFS layer with a cluster of single-node relational databases.

Like Hadapt, the CloudDB [16] project is a hybrid system. However, it supports both OLAP and OLTP by providing three types of data storage systems: a relational database, a NoSQL database and a database oriented for OLAP. Data is stored in the database, according to the guarantees of data consistency required by the user.

Similarly to our approach, PIQL [4] and Megastore [5] propose an architecture with higher-level processing functionality via a database library.

In PIQL, the application issues queries in a new declarative language that is based on a subset of SQL, but extended with new statements and some new operators to always achieve a predictable performance independently of the

[4] `http://www.hadapt.com/`

database size (i.e. scale-independence). However, there are several restrictions on the supported operations. For instance, a *table scan* is not scale-independent and has to be appended with a *limit* statement to bound the results. While this is done to achieve predictable performance it is a major impediment to run legacy applications.

MegaStore is built on top of BigTable and implements some of the features of RDBMS, such as secondary indexing. Nonetheless, join operations must be implemented at the application side. Therefore, applications must be written specifically for MegaStore using its data model and queries are restricted to scans and lookups.

The use of a library-centric component to offer higher-level processing functionality in our prototype is similar to the architecture of PIQL, Megastore, as well as that by Brantner et al. [6]. However, our architecture allows to combine the expressiveness and performance of RDBMS, taking advantage of its query optimizer and full SQL support. This allows our proposal to run existing SQL applications and tools while retaining the scale-out capabilities of the NoSQL database.

On a different perspective, other approaches make use of object mapping tools that allow to bypass the database lower level interfaces. By using [14], the user has at her disposal the generic object interfaces like JPA and JDO that allow her to use NoSQL databases in an almost transparent way, leveraging the knowledge already existent in the area. These solutions have also the advantage of aiding the migration of existent solutions based on object to relation mappers allowing the mix of different types of databases under the same code base.

6 Conclusions and Future Work

We presented a distributed query engine allowing to execute ANSI compliant SQL queries on top of a NoSQL database. The query engine allows to combine the expressiveness and performance of a Relational Database Management System with the scalability and schema flexibility of a NoSQL database.

The developed prototype offers a standard JDBC client interface and can be embedded in the client application as a middleware layer. It is stateless with respect to application data making its replication straightforward. For the execution of data management language statements it does not require any kind of coordination which prevents any undesirable impact on the scalability of the underlying NoSQL database. The feasibility of the approach is demonstrated by the performance results obtained with the YCSB and TPC-C benchmarks.

Moreover, the comparison with a NoSQL TPC-C implementation optimized for HBase shows that the presented prototype, through the use of the full-fledged query engine from Apache Derby achieves impressive performance.

Acknowledgment. This work is part-funded by: ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project Stratus/FCOMP-01-0124-FEDER-015020; and European Union Seventh Framework Programme (FP7) under grant agreement n° 257993 (CumuloNimbo).

References

1. BigQuery: Google (2011), http://code.google.com/apis/bigquery/
2. Hive: Hive (2011), http://hive.apache.org/
3. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proc. VLDB Endow. 2, 922–933 (2009)
4. Armbrust, M., Curtis, K., Kraska, T., Fox, A., Franklin, M.J., Patterson, D.A.: PIQL: success-tolerant query processing in the cloud. Proc. VLDB Endow. 5(3), 181–192 (2011)
5. Baker, J., Bondç, C., Corbett, J.C., Furman, J.J., Khorlin, A., Larson, J., Léon, J.M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In: CIDR (2011)
6. Brantner, M., Florescu, D., Graf, D., Kossmann, D., Kraska, T.: Building a database on S3. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, pp. 251–264. ACM, New York (2008), http://doi.acm.org/10.1145/1376616.1376645
7. 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: OSDI 2006 (2006)
8. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s hosted data serving platform. Proc. VLDB Endow (2008)
9. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SoCC 2010 (2010)
10. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: SOSP 2007 (2007)
11. Foundation, A.S.: Apache derby (2013), http://db.apache.org/derby/
12. George, L.: HBase: The Definitive Guide. O'Reilly Media (2011)
13. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. SIGOPS Operating Systems Review 37(5), 29–43 (2003)
14. Gomes, P., Pereira, J., Oliveira, R.: An object mapping for the Cassandra distributed database. In: Inforum (2011)
15. Google: Cloud SQL: pick the plan that fits your app. (May 2012), http://googleappengine.blogspot.pt/2012/05/cloud-sql-pick-plan-that-fits-your-app.html
16. Hacigümüs, H., Tatemura, J., Hsiung, W.P., Moon, H.J., Po, O., Sawires, A., Chi, Y., Jafarpour, H.: CloudDB: One Size Fits All Revived. In: Proceedings of the 2010 6th World Congress on Services (2010)
17. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: OLTP through the looking glass, and what we found there. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, pp. 981–992. ACM, New York (2008), http://doi.acm.org/10.1145/1376616.1376713

18. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC 2010, pp. 11–11. USENIX Association, Berkeley (2010), `http://dl.acm.org/citation.cfm?id=1855840.1855851`
19. Lakshman, A., Malik, P.: Cassandra - A Decentralized Structured Storage System. In: LADIS 2009 (2009)
20. Lin, L., Lychagina, V., Wong, M.: Tenzing: A SQL implementation on the MapReduce framework. Proceedings of the VLDB Endowment 4(12), 1318–1327 (2011)
21. Meijer, E., Bierman, G.: A co-relational model of data for large shared data banks. ACM Queue 9(3), 30:30–30:48 (2011), `http://doi.acm.org/10.1145/1952746.1961297`
22. Nadkarni, P., Brandt, C.: Data Extraction and Ad Hoc Query of an Entity-Attribute-Value Database. Journal of the American Medical Informatics Association 5(6), 511–527 (1998)
23. Rys, M.: Scalable SQL. ACM Queue: Tomorrow's Computing Today 9(4), 30 (2011)
24. SalesForce.com: Phoenix: A SQL layer over HBase (May 2013), `https://github.com/forcedotcom/phoenix`
25. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2010, pp. 1–10. IEEE Computer Society, Washington, DC (2010), `http://dx.doi.org/10.1109/MSST.2010.5496972`
26. Stonebraker, M., Cattell, R.: 10 rules for scalable performance in 'simple operation' datastores. Commun. ACM 54(6), 72–80 (2011), `http://doi.acm.org/10.1145/1953122.1953144`
27. 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: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB 2007, pp. 1150–1160. VLDB Endowment (2007), `http://dl.acm.org/citation.cfm?id=1325851.1325981`
28. Vilaça, R., Cruz, F., Oliveira, R.: On the expressiveness and trade-offs of large scale tuple stores. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2010. LNCS, vol. 6427, pp. 727–744. Springer, Heidelberg (2010), `http://dx.doi.org/10.1007/978-3-642-16949-6_5`

EZ: Towards Efficient Asynchronous Protocol Gateway Construction

Yérom-David Bromberg[1], Floréal Morandat[1], Laurent Réveillère[1], and Gaël Thomas[2]

[1] LaBRI University of Bordeaux, France
[2] LIP6-INRIA University of Pierre et Marie Curie

Abstract. Over the past decade, we have witnessed the emergence of a bulk set of devices, from very different application domains interconnected *via* Internet to form what is commonly named Internet of Things (IoT). The IoT vision is grounded in the belief that all devices are able to interact seamlessly with each other anytime, anyplace, anywhere. However, devices communicate *via* a multitude of incompatible protocols, and consequently drastically slow down the IoT vision adoption. Gateways, that are able to translate one protocol to another, appear to be a key enabler of the future of IoT but present a cumbersome challenge for many developers. In this paper, we are providing a framework called **EZ** that enables to generate gateways for either C or Java platform without requiring from developers any substantial understanding of either relevant protocols or low-level network programming.

1 Introduction

In the new era of the Internet Of Thing (IoT), interoperability is a key challenge. Over the last years, a promising solution that gained success to address interoperability issues is to use gateways that translate back and forth messages among heterogeneous protocols [1–5]. The design of such gateways [1–5] does not take as a first class priority the specific needs of the IoT context. First, gateways must scale in the large, i.e., they must efficiently manage both bulk sets of messages and simultaneous message translation processes. Second, gateways need to be as pervasive as possible: they must run on highly heterogeneous software environments. Current gateways only target low-level C code and are not adequate in the IoT context where Java is also a mainstream language.

In this paper, we propose `EZ` a new gateway compiler for the `z2z` language [2] that relies on the event paradigm. We choose the `z2z` language as it has already proven to be adequate to describe gateways. To solve performance issues `EZ` defines a workflow of handlers, which communicate with events. Further, to take into account software heterogeneity, `EZ` is able to compile the `z2z` language to both C and Java code, ready to be plugged into respective runtimes.

The remainder of this paper is structured as follows. Section 2 introduces our approach to generate from high level specifications our next generation gateways for either C or Java environment. Section 2.2 is focused on the internal design of our next generation gateways. In particular, it presents our new asynchronous runtime system for building scalable gateways in either C or Java along with EZ. Section 3 presents the performance evaluation of our asynchronous gateways. Finally, Section 4 reviews related research works and Section 5 concludes the paper with a discussion of future research directions.

2 EZ Approach

In a way similar to z2z, zebu [2,6], our approach is based on generative programming. More specifically, EZ reuses the z2z domain specific language to specify generated gateways (Figure 1, ❶). However, EZ introduces a new compiler enabling the generation of asynchronous code in either C or Java to be linked into an adequate and efficient event-based oriented runtime system (Figure 1, ❷, ❸) that fulfills the requirements of IoT. In other terms, developers only have to replace the z2z compiler by the EZ one to generate event-based gateways. These gateways use exclusively non-blocking system calls to perform asynchronous I/O network operations.

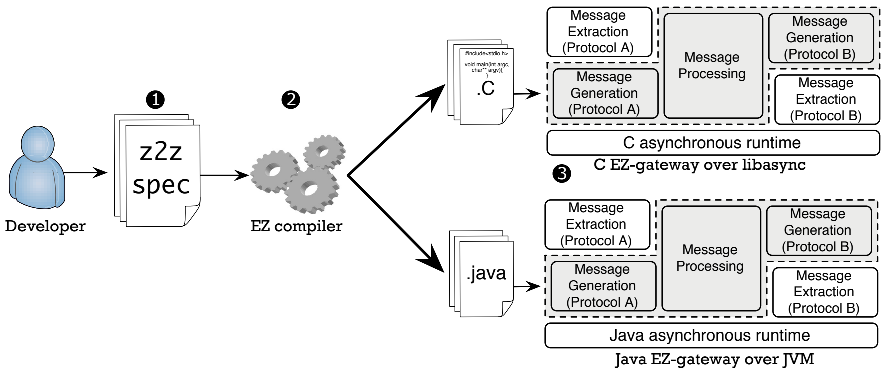

❶
❷
❸
Developer
z2z spec
EZ compiler
.C
.java
Message Extraction (Protocol A)
Message Generation (Protocol A)
Message Processing
Message Generation (Protocol B)
Message Extraction (Protocol B)
C asynchronous runtime
C EZ-gateway over libasync
Java asynchronous runtime
Java EZ-gateway over JVM

Fig. 1. EZ approach to generate asynchronous multi-platform gateway

2.1 Z2z Domain Specific Language

Our EZ approach has been design especially to be fully compliant with the z2z language that has been proved to be adequate to describe gateways in high level manner by hiding to developers low-level network and system codes.

Z2z Language. The z2z language provides facilities for defining three types of modules: the *Protocol Specification* (PS) modules used to describe the network protocol behaviors, the *Message Specification* (MS) modules used to describe message structures, and the *Message Translation* (MT) modules used to describe the message translation logic. More precisely, a PS module provides information about various properties of the interaction with the network, such as the transport protocol used, whether requests are sent in unicast or multicast, and whether responses are received synchronously or asynchronously. It also specifies how to dispatch a received request to a specific handler for processing. A MS module defines the useful information to be extracted from incoming messages, i.e., *message views*. It also defines the structure of new messages to be created, i.e., *message templates*. A template contains a message view that describes "holes" to be filled by the translation logic when creating a new message. Finally, a MT module is specified using a dedicated C-like syntax and consists of a set of handlers, one for each kind of relevant incoming requests, as indicated by the protocol specification. It provides domain-specific operators for manipulating and constructing messages, for sending requests and returning responses, and for managing session state across requests.

Z2z Front-End Compiler. Our new `EZ` compiler reuses the `z2z` front-end that deals with the scanning and the parsing of the language and performs consistency checks across the PS, MS and MT modules. For instance, the front-end checks that the MT module defines a handler for each kind of message that should be handled by the gateway and that each handler has an appropriate return type according to the PS module. It guaranties that all fields and values have been appropriately initialized and used across the different modules. Further, the front-end compiler performs as well a data-flow analysis of the message translation code to detect erroneous specifications and ensure the generation of safe code.

2.2 Asynchronous EZ Runtime

One key contribution of the `EZ` approach is to be able to generate in a transparent manner either C or Java gateways. Specifically, both C and Java based gateways use runtimes implemented with an event-based programming paradigm built on top of the libasync [7] event-driven library for the C version and the new NIO.2 API provided by the JDK7 for the Java version.

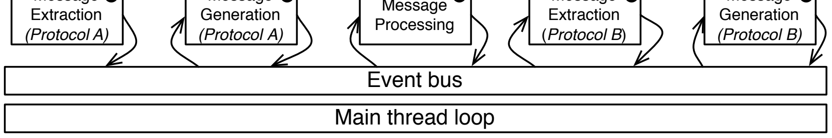

Message ❶ Extraction (Protocol A)
Message ❷ Generation (Protocol A)
Message ❸ Processing
Message ❶ Extraction (Protocol B)
Message ❷ Generation (Protocol B)
Event bus
Main thread loop

Fig. 2. Event-based processing chain

Gateways are decomposed into key functional building blocks, such as message extraction, processing, and generation. Each of these building blocks registers their interests into network `I/O` events (See Figure 2,❶,❷,❸). Further, each building block may additionally interact with each other *via* the use of events that they can generate by themselves. For instance, events for either notifying the arrival of a recognized and fully parsed message, for indicating that no data can be read anymore, or for notifying network `I/O` error, are asynchronously dispatch to the building blocks that have declared their interest in these events. Incoming messages that traverse the event-based processing chain are then decomposed into multiple events that are serially dispatched one at a time by a main thread loop. As a consequence, the pipeline, contrary to our previous work [1–5], is not anymore shared among a pool of threads avoiding so mutex lock contentions and increasing inherently performances.

Event-Based Message Processing. As opposed to `z2z` for instance, `EZ` treats each operation performed by the translation logic, that leads to an `I/O` access, as an asynchronous call and thus is compiled by the `EZ` compiler so as to produce a continuation. When the operation is completed, an event is generated and caught by the runtime system that resumes the associated continuation to resume the processing. This strategy results in the creation of many continuations for a handler.

Event-Based Message Extraction. In contrast to the thread model that accumulate data to reconstruct the corresponding message before processing it, `EZ` processes messages even if there are not yet fully parsed. When no more data can be read, because of message fragmentation for example, the parser is paused, the current state is saved and a callback is attached to the availability of new data. When the associated event occurs, the callback is invoked and execution continues in the previously saved state. A soon as a fragment or a message field is recognized by the parser an event is immediately triggered. This event is then stored in the message view to be processed by the other building blocks such as the message processing one. With `EZ`, message parsing drastically limits memory consumption because a message does not need to be entirely saved in memory before starting its parsing. Instead, in C based version of the runtime, two contiguous memory pages are used as a circular buffer, accounting for about 8KB of memory, whereas in the Java version of the runtime two buffers are used and flipped when the first one is exhausted. This only works thanks to the scattered read facility of Java network API. In the either C or Java runtimes, dynamic memory allocation is only required for filling values of the message view when fields are recognized.

Event Based Error Management. When an event occurs, the associated callback function is executed by the runtime system. However, events corresponding to `I/O` operations may never occur if the underlying operation fails to complete, leading thus to memory leaks difficult to address. To overcome this issue, C and Java `EZ` runtimes attach timeout on each event. If the corresponding timer expires before the occurrence of the event, the runtime frees associated resources to prevent memory leaks.

3 Evaluation

To assess the scalability of `z2z` and `EZ` gateways, we have implemented the SMTP/HTTP and HTTP/STMP gateways described previously. Our performance experiments are carried out on a Dell Intel® Xeon® server powered by 4 processors of 8 hyper-threaded cores clocked at 2.2 GHz. We use the multi-threaded SMTP test client and server distributed with Postfix to stress the generated gateways. C code is compiled using gcc 4.7.2 and Java code executes on HotSpot server 1.7. For our experiments, a given number of simultaneous clients (up to 30) send 1000 messages of 10KB in a closed loop[1] to a SMTP server through a tunneling application. Figure 3 shows the response time for each gateway, as well as the native protocol communication costs (`direct`). Since gateways are I/O intensive, a certain amount of clients are required to get satisfying throughput. The event-based gateways (`EZ-C` and `EZ-Java`) clearly outperform the thread-based gateways (`z2z`).

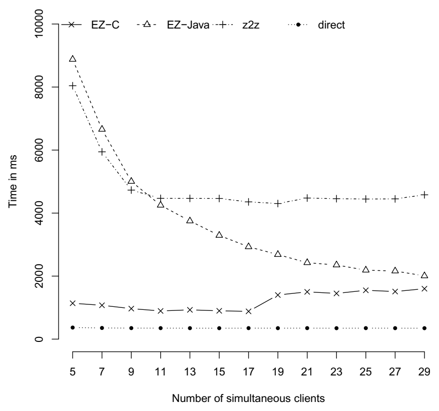


Fig. 3. Time to send 1000 message of 10kB by N clients

4 Related Work

There have been a bulk set of different approaches to protocol interoperability, for instance to name a few, ReMMoC [8], RUNES [9], MUSDAC [10], BASE [11], INDISS [1], Starlink [3] and Enterprise Service Buses [12]. Compared to `EZ`, these approaches have three major weak points: neither they address the difficulty of gateway development nor they tackle the scalability issue, and nor they target both C and Java environment. The closest approach to ours is z2z [2], which constitutes the first generative approach for building gateways. However, z2z generated gateways exhibit poor scalability in the face of the increasing load generated by clients and the increasing size of messages. Furthermore, z2z only targets C environment and is therefore less pervasive than a Java based version.

Scalability has been a major concern in the field of Web servers as these systems must efficiently operate on thousands of files and connections. However, Web servers are CPU intensive oriented while gateways are I/O intensive. Thus experiences on Web server architecture should not be granted for gateways.

5 Conclusion and Future Work

Network protocol gateways are a key enabler of the future of IoT but present a cumbersome challenge for developers. In particular there are three main challenges that need to be overcome to build gateways: (i) providing an easy way to build gateways by hiding to developers intricacies of low-level network and

[1] A client sends a new message only when it receives an acknowledgement from the server for the reception of the previous message.

system code, (ii) tackling the scalability issue,(iii) beeing pervasive, i.e. multi-platform compliant (i.e. C or Java based). As a future direction, multicore architectures are today a reality in all kinds of computing systems, ranging from powerful servers to desktop environments and embedded systems. However current systems and applications are unable to fully exploit these new architectures. Taking advantage of multicore hardware is thus today one of the most important scientific challenges in the systems domain. We are investigating the extension of EZ to support multicore architectures. Another potential research direction is to raise on our ongoing research work on hardware accelerated parsers to speed up drastically messages processing in embedded platforms [13].

References

1. 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)
2. 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)
3. Bromberg, Y.D., Grace, P., Reveillere, L.: Starlink: Runtime interoperability between heterogeneous middleware protocols. In: ICDCS, pp. 446–455 (2011)
4. Issarny, V., Bennaceur, A., Bromberg, Y.D.: Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In: SFM, pp. 217–255 (2011)
5. Rodrigues, P., Bromberg, Y.-D., Réveillère, L., Négru, D.: ZigZag: A middleware for service discovery in future internet. In: Göschka, K.M., Haridi, S. (eds.) DAIS 2012. LNCS, vol. 7272, pp. 208–221. Springer, Heidelberg (2012)
6. Burgy, L., Reveillere, L., Lawall, J., Muller, G.: Zebu: A language-based approach for network protocol message processing. IEEE Transactions on Software Engineering 37(4), 575–591 (2011)
7. Mazières, D.: A toolkit for user-level file systems. In: USENIX-ATC, pp. 261–274 (2001)
8. Grace, P., Blair, G.S., Samuel, S.: A reflective framework for discovery and interaction in heterogeneous mobile environments. SIGMOBILE Mob. Comput. Commun. Rev. 9(1), 2–14 (2005)
9. Costa, P., Coulson, G., Mascolo, C., Mottola, L., Picco, G.P., Zachariadis, S.: Reconfigurable component-based middleware for networked embedded systems. International Journal of Wireless Information Networks 14(2), 149–162 (2007)
10. Raverdy, P.G., Issarny, V., Chibout, R., de La Chapelle, A.: A multi-protocol approach to service discovery and access in pervasive environments. In: MobiQuitus, pp. 1–9 (2006)
11. Becker, C., Schiele, G., Gubbels, H., Rothermel, K.: Base: A micro-broker-based middleware for pervasive computing. In: PERCOM, p. 443 (2003)
12. Chappell, D.: Enterprise Service Bus. O'Reilly (2004)
13. Mercadal, J., Réveillère, L., Bromberg, Y.D., Gal, B.L., Bissyandé, T.F., Solanki, J.: Zebra: Building efficient network message parsers for embedded systems. Embedded Systems Letters 4(3), 69–72 (2012)

Autonomous Adaptation of Cloud Applications

Everton Cavalcante[1], Thais Batista[1], Frederico Lopes[1], André Almeida[1], Ana Lúcia de Moura[2], Noemi Rodriguez[2], Gustavo Alves[1], Flavia Delicato[3], and Paulo Pires[3]

[1] UFRN – Federal University of Rio Grande do Norte, Natal, Brazil
evertonrsc@ppgsc.ufrn.br, thais@ufrnet.br,
{fred.lopes,gustavoalvescc}@gmail.com, andre.almeida@ifrn.edu.br
[2] PUC-Rio – Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil
{amoura,noemi}@inf.puc-rio.br
[3] UFRJ – Federal University of Rio de Janeiro, Rio de Janeiro, Brazil
{fdelicato,paulo.f.pires}@gmail.com

Abstract. Cloud-based applications are built from services offered by distinct third-party cloud providers. Most cloud-related information (service properties such as price, availability, response time, etc.) can change at any time during application execution. Therefore, it is essential to support the adaptation of applications in such dynamic conditions in order to ensure that the cloud services currently provided to deployed applications adhere to the established requirements. In this paper we present an autonomous adaptation process for cloud-based applications by replacing a service by an alternative one that fulfills the application needs. We discuss the factors that trigger an adaptation in a Cloud Computing scenario and describe the adaptation process within *Cloud Integrator*, a service-oriented middleware platform for composing, executing, and managing services provided by different cloud platforms.

Keywords: Cloud Computing, Cloud services, *Cloud Integrator*, Semantic workflows, Autonomous adaptation.

1 Introduction

The growing interest in the Cloud Computing paradigm is grounded on its utility model in which *computing services* are delivered through a pay-per-use model. By exploiting this model, applications can be composed of services provided by distinct third-party cloud providers. The selection of the proper cloud services that fit the application needs is based on non-functional information, i.e. properties of the services such as price, availability, response time, etc., and applications can rely on a middleware that abstracts away the burden of directly dealing with underlying mechanisms for service selection and communication with the cloud providers.

In this context, our previous work introduced *Cloud Integrator* [1–3], a service-oriented middleware platform for composing, executing, and managing services provided by different Cloud Computing platforms, so that *Cloud Integrator* works as a *mediator* between the service providers and the applications (clients). Moreover, it provides an environment that facilitates the development and execution of fault-tolerant applications that use such services by composing *semantic Web services* [4] in a *semantic workflow* [5] composed of a sequence of abstract *activities* that must be performed by concrete cloud services in order to achieve the application's business goal. In order to execute such semantic workflow, it is necessary to create at least one *execution plan* containing a set of concrete Web services that perform each one of the activities specified in the workflow.

By using *Cloud Integrator*, an application can specify the set of cloud services that it needs as well as the properties of each service, and the selection mechanism provided by the middleware platform is able to choose the cloud services provided by the integrated platforms that fulfill the application requirements. However, most cloud-related information (service properties) can change at any time during application execution, so that it is essential to support the *adaptation* of applications in such dynamic conditions to ensure that the cloud services currently provided to the applications adhere to the established requirements.

In this paper we present an adaptation process to coordinate the autonomous adaptation of cloud applications based on the replacement of services by alternative ones that fulfill the application requirements, in case of service failure or when any change in the properties of a cloud service (e.g. quality parameters) can potentially affect the running application(s). Section 2 details our adaptation process, which is evaluated in Section 3. Finally, Section 4 presents final remarks.

2 Adaptation of Cloud Applications

The service composition model adopted by *Cloud Integrator* is based both on the *functionality* of each service and on its *metadata* (such as QoS parameters and prices), thus enabling a better choice of the available services. Moreover, this composition mechanism enables *Cloud Integrator* to deal with situations in which a service that is available at composition time becomes unavailable at runtime or in case of quality degradation of any service that is included in the composition. If two or more services with similar functionality are available, then *Cloud Integrator* builds different execution plans for the current workflow, each of them using one of these alternative services. Thus, in case of service failure or quality degradation, another execution plan that contains the service with similar functionality can replace the current one in order to ensure the quality and availability of the running application. In this section we describe the *adaptation process* that supports this capability. Herein, an *adaptation* of an application means to replace a running execution plan by another one that performs the same activities. The adaptation process performed by *Cloud Integrator* currently addresses the replacement of *application services*, which may be SaaS and/or PaaS cloud services or even other traditional (non-cloud) services.

2.1 Factors That Trigger an Adaptation

There are three classes of changes in cloud environments that may trigger the adaptation process. The first one is the *failure of one or more services* due to loss of connection between *Cloud Integrator* and a service provider or internal service errors. In both cases, the service provider becomes unable to respond to requests and then the current execution plan that contains such service must be immediately replaced by an alternative execution plan that contains a service with similar functionality, when possible. For instance, suppose that the execution plan $A \rightarrow B \rightarrow C_1 \rightarrow D$ is being executed and the service C_1 becomes unavailable. In this case, the adaptation mechanism must select an alternative execution plan with the equivalent service C_2 (for instance, the execution plan $A \rightarrow B \rightarrow C_2 \rightarrow D$), thus avoiding the failure of the whole application.

The second class of events that may trigger an adaptation is the *quality degradation of one or more services*. Cloud environments can present highly dynamic execution conditions in which fluctuations in the network and high service usage may affect the quality of executing services. In this case, an alternative execution plan containing a service that provides similar functionality to the degraded service can be adopted if its utility[1] [3] is significantly higher than the utility of the current execution plan. Finally, the third case is the *arising of new services*, which can be dynamically discovered. Since these new services may provide more advantageous alternative execution plans for running applications, it is necessary to analyze the need and convenience of replacing a current execution plan with an alternative one that contains one or more new services. Similarly to quality degradation, it is also important to consider the utility gains and the impact regarding the replacement of the current execution plan. In the current development state of *Cloud Integrator*, the adaptation process is only triggered in case of service failures. Adaptation triggered by quality degradation or discovery of new services will be addressed in future works.

2.2 Factors That Affect the Adaptation Process

When the adaptation process replaces an execution plan regarding an application, its major concern is to ensure that the application requirements continue to be satisfied. Although the quality of alternative execution plans is an essential factor to be considered, it is not sufficient to ensure an efficient adaptation. Moreover, since replacing an execution plan may lead to the re-execution of services or other costly actions, the decision about such possible replacement must consider the *adaptation cost*, which is the *overhead* imposed by actions that must be performed in order to resume the application after replacing the current execution plan with an alternative one. As an example, when some of the services included in the current execution plan have already been executed, it may be advantageous to select the alternative execution plan that has more

[1] As presented in our previous work [3], the computation of the utility of an execution plan takes into account both QoS parameters and the monetary costs of the services that compose it.

services in common with the current one. Therefore, such similar plan can offer a lower adaptation cost by reusing the outputs produced by the executed services, thus avoiding the need of actions that may have considerable impact.

The computation of the adaptation cost regarding alternative execution plans must consider the following factors: (i) *reuse of executed services*; (ii) the *rollbacks* required to return services that have already been executed to a previous execution state, and; (iii) *compensatory actions* taken in order to restore an previous execution state when a service needs to return to a previous state and it does not support rollbacking. In this perspective, the computation of the *adaptation cost* of an execution plan p starts with the computation of its *absolute adaptation cost* $c_{abs}(p)$ (Eq. 1), which is defined as the sum of the number of services to be executed after the replacement of the current execution plan (e), the number of services that require rollbacks (r), and the number of services that require compensatory actions (a):

$$c_{abs}(p) = e + r + a \tag{1}$$

In turn, the adaptation cost $c(p)$ regarding an execution plan p is calculated through a vector normalization of its absolute adaptation cost $c_{abs}(p)$ (Eq. 2):

$$c(p) = 1 - \frac{\frac{1}{c_{abs(p)}}}{\sqrt{\sum_{r \in EP}\left(\frac{1}{c_{abs}(r)^2}\right)}} \tag{2}$$

in which $c_{abs}(r)$ is the absolute adaptation cost of each execution plan r in the set of available execution plans EP.

2.3 The Adaptation Process

The adaptation process performed by *Cloud Integrator* was inspired in *Adapt-UbiFlow* [6], an adaptation mechanism proposed in the context of ubiquitous applications. The selection of an alternative execution plan considers two essential parameters: (i) the *initial utility* (u) of the candidate execution plans, which expresses their properties in terms of price and quality parameters, and; (ii) the *adaptation cost* (c), which weights the actions that must be performed when replacing the application's execution plan. When the adaptation process is triggered, the initial utility of the alternative plans is computed by using the same criteria (QoS parameters and prices) that were originally used for selecting the application's current execution plan [3]. Next, the *adaptation cost* is computed for each alternative plan, as shown in Equation 2. In the adaptation process, the selection of an execution plan is based on the computation of the *adaptation utility* $\mu(p)$ of each alternative execution plan p defined as a weighted sum of the initial utility $u(p)$ and the adaptation cost $c(p)$. Equation 3 shows how the adaptation process computes the adaptation utility $\mu(p)$ for each alternative execution plan p, so that the alternative plan with maximum adaptation utility is selected to replace the current execution plan. If two or more execution plans have maximum utility, the adaptation process selects one of them at random.

$$\mu(p) = [u(p) * w_{EP}] + [c(p) * w_{AC}] \quad (3)$$

The weights w_{EP} and w_{AC} in Equation 3 are respectively assigned by the user to the initial utility and the adaptation cost, with w_{EP}, $w_{AC} \in [0, 1]$ and $w_{EP} + w_{AC} = 1$. The values assigned to these weights are defined in terms of five different configurations called *adaptation profiles*, as shown in Table 1.

Table 1. Adaptation profiles for the definition of the weights assigned to initial utility and adaptation cost

Adaptation profile	Description	Assigned weights	
		w_{EP}	w_{AC}
maximum initial utility	exclusive priority to the initial utility of the execution plan	1.00	0.00
high initial utility	prioritizes the initial utility of the execution plan while considering the adaptation cost	0.75	0.25
balanced	default configuration, with equal weights	0.50	0.50
low adaptation cost	prioritizes the adaptation cost, while also considering the initial utility of the execution plan	0.25	0.75
minimum adaptation cost	exclusive priority to the adaptation cost	0.00	1.00

After selecting an alternative execution plan, the required rollbacks and compensatory actions are transparently performed in order to replace the current execution plan and resume the execution of the application.

3 Evaluation

A preliminary evaluation was performed aiming to assess the time spent by the adaptation process triggered in case of failure of a service involved in the execution plan that is being executed, so that such failure is detected when a timeout in which the service does not respond to the request is reached. In this perspective, services involved in the selected execution plan for executing the application were forced to fail in order to trigger the adaptation process. As shown in Table 2, the time spent in milliseconds to perform the adaptation process (and that covers the selection of an alternative execution plan to replace the current one) is significantly small, thus not significantly impacting on the application execution. More details about the performed evaluation can be found at `http://consiste.dimap.ufrn.br/projects/cloudintegrator/dais2013`.

Table 2. Minimum and maximum values, average and standard deviation regarding the time (in milliseconds) spent by the adaptation process

Execution plans	Minimum	Maximum	Average	Standard deviation
2	0.2754	0.5950	0.4694	0.0708
4	0.4598	0.9522	1.2352	0.1626
8	0.7716	1.9416	1.7846	0.2574
12	1.4680	3.7006	3.2596	0.5168
18	4.0158	11.7134	7.2724	1.7644

4 Final Remarks

In this work we discussed the events that can trigger the need for adaptation in a cloud-based environment, as well as the factors that should be taken into consideration when choosing the best way to react to changes in the runtime environment in order to ensure the quality and availability of the application. We described the adaptation support designed for *Cloud Integrator* in which service failures and quality degradation are addressed with an algorithm that takes the adaptation cost (incurred in the replacement of services) into account. A preliminary evaluation of such adaptation process in case of service failures has shown that the adaptation process does not significantly impact in the application execution. In addition, the proposed adaptation process works with minimal user awareness, thus promoting the autonomy of the application in case of failures or other conditions that may trigger an adaptation.

References

1. Cloud Integrator, `http://consiste.dimap.ufrn.br/projects/cloudintegrator/`
2. Cavalcante, E., et al.: Cloud Integrator: Building value-added services on the cloud. In: First International Symposium on Cloud Computing and Applications, pp. 135–142. IEEE Computer Society, USA (2011)
3. Cavalcante, E., et al.: Optimizing services selection in a cloud multiplatform scenario. In: 2012 IEEE Latin America Conference on Cloud Computing and Communications, pp. 31–36. IEEE Computer Society, USA (2012)
4. Martin, D., et al.: Bringing Semantics to Web Services: The OWL-S Approach. In: Cardoso, J., Sheth, A.P. (eds.) SWSWPC 2004. LNCS, vol. 3387, pp. 26–42. Springer, Heidelberg (2005)
5. Abbasi, A., Shaikh, Z.: A conceptual framework for smart workflow management. In: 2009 International Conference on Information Management and Engineering, pp. 574–578. IEEE Computer Society, USA (2009)
6. Lopes, F., et al.: AdaptUbiFlow: Selection and adaptation in workflows for Ubiquitous Computing. In: 9th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, pp. 63–71. IEEE Computer Society, USA (2011)

Toward Unified and Flexible Security Policies Enforceable within the Cloud

David Eyers[1] and Giovanni Russello[2]

[1] University of Otago, NZ
[2] University of Auckland, NZ

Abstract. Security engineering for any given application can usually be done in many different ways. There is often a tradeoff between usability (including efficiency) and the level of protection offered. Typically the risks are assessed by developers, and a particular approach is chosen, with the assumption that the design can stay fixed for some time.

Adoption of Cloud computing will challenge the viability of this approach. Beyond the extra difficulties faced when doing security engineering within distributed systems, Cloud providers require a different threat model from self-hosted resources. They are best considered "trusted but curious" even if the curiosity is accidental on the Cloud provider's part. Some threats from such Cloud providers can be confounded through the use of cryptography, but doing so is overkill in terms of the performance penalty for many applications.

To acquire the benefits of Cloud computing while minimising security risks, we believe that application developers should be provided with the ability to dynamically change the security enforcement technology in use by their software, balancing performance and security as they see fit. Recent cryptography research will significantly increase our ability to offer a runtime choice of contrasting security enforcement approaches *without needing to modify the security policy*. We present our initial research into this area, and outline our vision for the future.

1 Introduction

Application developers need to carefully consider the security engineering of their software. This is particularly true today, as so many devices are network accessible—and indeed need network functionality in order to operate at all—their exposed security surface area is larger than in the past days of isolated microcomputers.

Even software on single machines needs to interact with a number of local software components (broadly defined), simply due to the need to build on top of existing operating system and language runtime codebases.

Two broad types of data security model are Discretionary Access Control (DAC) and Mandatory Access Control (MAC)—see [1] for details. In DAC systems, the owners of protected resources are empowered to change the permissions of those objects, e.g., in Access control lists, capabilities and role-based access control (RBAC).

In MAC models, access control policy is enforced regardless of the desires of the owners of a protected resource. A common illustration is multi-level security models used in the military and other government organisations: data and principals are given security labels, and policy determines for any data access whether that access will be permitted as a function of the type of access and the labels of the principal and the data, e.g., policy may prevent a principal writing ‘top secret’ data into a ‘classified’ resource.

DAC schemes are often enforced by ensuring that code execution paths that access protected resources do permission checking before allowing such access. A risk of this approach is that it relates more to software code than the data it is trying to protect: in some cases access control checks may be accidentally omitted. In other cases this is due to control flow paths allowing implicit access to protected data.

MAC schemes are frequently implemented in a more directly data-centric manner. Since no code path should be able to circumvent the label-based protection, it is appropriate to use techniques to protect the data that may have higher run-time overheads, such as hardware-assisted memory protection. It is impractical to avoid having some trusted computing base (TCB) even in the strictest MAC schemes, but the TCB is ideally isolated entirely from the applications and data it is protecting (often it would be part of the operating system).

One potential approach to ensuring mandatory, data-linked protection against information leakage is to maintain the data in encrypted form, if the TCB is the only part of the code that possesses the decryption keys. Implementing MAC this way may allow for less effort being required to protect the target data: some application-level functionality, such as writing data into a database, might be able to be permitted, despite the data storage functionality not actually being contained within the TCB. This may lead to a more practical system than approaches that require expensive runtime interception of all protected data access.

Regardless of the mechanism used to enforce policy, generally accepted best practice is to abstract security policy away from application-specific implementation code. Widely available implementations of access control technologies such as XACML have helped make it easier to achieve this practice. Maintaining this sort of policy abstraction simplifies making access control policy modifications after software has been deployed.

Security engineering in distributed systems is significantly more complex than for software running on single hosts. However, widespread adoption of Cloud computing will cause distributed security engineering to be required within a growing proportion of applications. The terminology of IETF RFC 2904 [2] includes separate notions of a policy decision point (PDP) and a policy enforcement point (PEP), which highlights the possibility to perform expensive policy checking in a different part of the distributed system than the software component that mediates access to a protected resource.

1.1 Enter Cloud Computing

Cloud computing provides a challenging type of distributed system in which to deploy security-sensitive software. The challenge stems from the Cloud provider being a different organisation from the Cloud tenant, combined with the requirement that Cloud resources need to be accessed across a network. Beyond the need to run above a hypervisor in the first place, Infrastructure as a Service (IaaS) Cloud offerings need to manage the data travelling in a Cloud provider’s network between instances of virtual machines, e.g., that host parts of a typical three-tier web application. Then, as Cloud deployment increasingly uses Platform as a Service (PaaS) offerings, a growing number of application components will be distributed in a way that exposes previously internal security considerations.

Cloud computing providers are most safely treated as “trusted but curious”. The curiosity might well be accidental—a provider may leak data from the processes that they use to achieve data backups, for example. However the Cloud provider is clearly “trusted” in that outsourcing to an overtly malicious organisation makes no sense at all.

Put another way, if the primary objectives of computer security are to effect confidentiality, integrity, availability, then we are targeting systems that can provide integrity and availability, but through software or working practice errors, may fail to maintain complete confidentiality.

In this respect, Fully Homomorphic Encryption (FHE) represents the holy grail in Cloud security: FHE allows computers to perform arbitrary computation on encrypted data. FHE protects users’ privacy from curious Cloud providers since they are unable to view the users’ data. However, the first solution to FHE presented by Gentry [3] is very inefficient. Partially Homomorphic Encryption (PHE) schemes exist supporting limited computations on the encrypted data but in a more efficient way. PHE schemes have been used extensively for supporting search operations over encrypted databases in outsourced settings [4,5,6,7,8,9,10,11,12,13,14,15]. The main issue of these approaches is that they enforce a very basic access control policy: if a user has a decryption key then she is allowed to access the whole database. We argue that more flexible security policies can be enforced if the mechanism for protecting the data from the Cloud provider is separated from the mechanism for controlling access to the data.

1.2 Contribution

This position paper describes our work to increase the flexibility of security in outsourced systems, by proposing a distributed architecture where data protection mechanisms are orthogonal to the security policy enforcement. We aim to allow developers to flexibly tradeoff between speed, simplicity and security for different parts of their applications without needing to rewrite their access control policy.

In particular, application developers can choose between enforcing security using access control barriers, or through the use of encryption. Using barriers will process data more quickly, but the policy enforcement infrastructure must be trusted to see the data that it is releasing to the principals. In the latter case, we extend how cryptographic protection is implemented, by utilising PHE algorithms to allow the policy enforcement point to remain oblivious to the content of the data being released, avoiding an undesired, additional TCB point within the overall distributed system.

2 Encrypted Policy Enforcement

There are several PHE-based solutions that offer encrypted storage of data while allowing basic search capabilities to be performed on the server side without the server learning anything about the plaintext data [4,5,6,7,8,9,10,11,12,13,14,15]. However, these solutions assume all users have the same right to access data. Basically, these solutions lack access control mechanisms to enforce access policies for regulating the access of a user (or a group of users) to a particular subset of the stored data. To partially alleviate this problem, solutions such as the one described in [16] have been proposed where access policies are encoded as a set of encryption keys. Only users possessing a key are authorised to access the data. The main drawback of this approach is that security policies are tightly coupled with the security mechanism. Therefore, any changes in the security policies require to generate new keys and to redistribute them to the users.

In our research, we argue that the security model used for controlling access to the data should be made disjoint from the mechanism used for protecting the data from the Cloud provider. For instance, most companies use as a security model for protecting their IT infrastructure variations of Role-Based Access Control (RBAC) [17]. In the RBAC model, access decisions are based on the role of the user making a access request. However, typical implementations of this model cannot be deployed on an infrastructure that is not fully trusted, such as Cloud environments because the information they deal with may leak information on the data they are protecting and the internal structure of the organisation. Similar considerations can be made for other MAC and DAC access control models—but for lack of space we focus on the RBAC model here.

We consider an Cloud scenario with outsourced security management involving two service providers: the **Data Service Provider** is responsible for storing the data; the **AC Service Provider** is responsible for the enforcement of access control policies. For reliability, the two services might be provided by different Cloud providers (although this is not a strict requirement for our approach). It is assumed that the Cloud providers are honest-but-curious, that is, they allow the components to follow the protocol to perform the required actions but curious to deduce information about the exchanged and stored data. Figure 1 provides an overview of our distributed architecture for separating policy enforcement from data protection mechanisms.

The responsibility of specifying policies and updating them is that of the **Admin User**. The **Requester** requests access to the data residing in the outsourced environment. The **Company RBAC Manager** is responsible for assigning a role to the requested user (that is, an Admin User or a Requester). The **Server RBAC Manager** is responsible for managing the encrypted role hierarchy graph. The **Trusted Key Management Authority (KMA)** generates and securely transmits the secret keys to the key store. An Admin User (i) gets a role from the Company RBAC Manager and then (ii) deploys the access policies to the Administration Point that (iii) stores the policies in the Policy Store. Meanwhile, the Company RBAC Manager sends the role hierarchy graph that is stored by the Server RBAC Manager (iv).

To request data, a query needs to be encrypted first (0). The Requester then (1) gets a role assigned by the Company RBAC Manager. Next, the Requester (2) sends the request to the Policy Enforcement Point (PEP) which (3) forwards the request to the Policy Decision Point (PDP). The PDP (4) fetches matching policies against the request, (5) collects the contextual information from the Policy Information Point (PIP), and (6) the role information. The PDP evaluates the access request and the contextual information against policies and sends the policy decision to the PEP (7). If the decision is a *deny*, then PEP usually drops communication or replies to the Requester with an error.If the decision is *permit*, the PEP (8) forwards the encrypted query to the Data Store. Finally, the PEP (9) forwards the query results to the Requester.

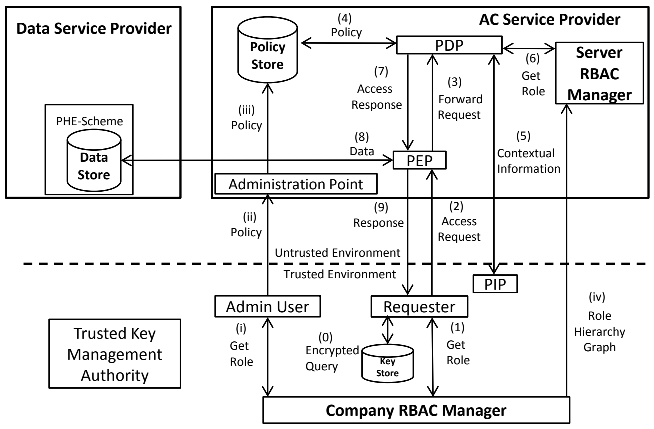


Fig. 1. Our distributed architecture for enforcing RBAC in outsourced environments

The enforcement of encrypted RBAC policies is based on the ESPOON$_{ERBAC}$ system [18]. However, the idea presented in this position paper is to extend the functionality of the PEP to support communication with external service providers where the data is stored. The storage service can use several different types of data protection, independently of the access control model implemented in the AC Service Provider.

The degree of decoupling can be selected by the system designer, depending on their view of the security risk posed by the various different hosting organisations in use within the distributed system. If the trusted/untrusted environment division shown in figure 1 is too conservative for some parts of the application, 'untrusted' components can be moved to the 'trusted' side. Crucially, for the RBAC policies discussed here, this change in trust *does not require the access control policy to be rewritten.*

3 Future Work and Conclusion

In this paper we have discussed how Cloud computing environments have increased the number of participants in distributed access control applications, and how this can negatively impact security. Nonetheless, highest-grade encryption may be overkill for some parts of an application, where the security risks do not justify its use.

Our goal is to develop security technologies for Cloud computing that can flexibly change between software-based protection (i.e., access control monitors) and stronger protection encoded using encryption, without requiring the policy to be rewritten. We have presented an implementation of RBAC that uses the ESPOON$_{ERBAC}$ system, and illustrated how PHE solutions can provide a cryptography-based distributed RBAC enforcement environment. The same RBAC can be checked more cheaply using conventional access control monitors, working from an equivalent policy implementation. Our future work will increase the coverage of policy features that can be implemented using encryption, and to improve the performance of our techniques. We believe our notion of decoupling access control enforcement from the policy specification will be crucial to effect comprehensive security within Cloud computing.

References

1. Department of Defense: Department of Defense Trusted Computer System Evaluation Criteria. (December 1985) DOD 5200.28-STD (supersedes CSC-STD-001-83).
2. Vollbrecht, J., Calhoun, P., Farrell, S., Gommans, L., Gross, G., de Bruijn, B., de Laat, C., Holdrege, M., Spence, D.: IETF RFC 2904: AAA authorization framework (2000)
3. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, pp. 169–178. ACM, New York (2009)
4. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceedings of the 2000 IEEE Symposium on Security and Privacy, SP 2000, pp. 44–55. IEEE Computer Society, Washington, DC (2000)
5. Hacigümüş, H., Iyer, B., Li, C., Mehrotra, S.: Executing sql over encrypted data in the database-service-provider model. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pp. 216–227. ACM (2002)
6. Golle, P., Staddon, J., Waters, B.: Secure conjunctive keyword search over encrypted data. In: Jakobsson, M., Yung, M., Zhou, J. (eds.) ACNS 2004. LNCS, vol. 3089, pp. 31–45. Springer, Heidelberg (2004)
7. Chang, Y.-C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: Ioannidis, J., Keromytis, A.D., Yung, M. (eds.) ACNS 2005. LNCS, vol. 3531, pp. 442–455. Springer, Heidelberg (2005)
8. Wang, H., Lakshmanan, L.: Efficient secure query evaluation over encrypted xml databases. In: Proceedings of the 32nd International Conference on Very large Data Bases, pp. 127–138. VLDB Endowment (2006)
9. Bösch, C., Brinkman, R., Hartel, P., Jonker, W.: Conjunctive wildcard search over encrypted data. In: 8th VLDB Workshop on Secure Data Management, Seattle, WA, USA (2011)
10. Popa, R., Redfield, C., Zeldovich, N., Balakrishnan, H.: Cryptdb: protecting confidentiality with encrypted query processing. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, pp. 85–100. ACM (2011)
11. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: CCS 2006: Proceedings of the 13th ACM Conference on Computer and Communications Security, pp. 79–88. ACM, NY (2006)
12. Boneh, D., Waters, B.: Conjunctive, subset, and range queries on encrypted data. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 535–554. Springer, Heidelberg (2007)
13. Zhu, B., Zhu, B., Ren, K.: Peksrand: Providing predicate privacy in public-key encryption with keyword search. In: 2011 IEEE International Conference on Communications (ICC), pp. 1–6. IEEE (2011)
14. Dong, C., Russello, G., Dulay, N.: Shared and searchable encrypted data for untrusted servers. Journal of Computer Security 19(3), 367–397 (2011)
15. Shao, J., Cao, Z., Liang, X., Lin, H.: Proxy re-encryption with keyword search. Inf. Sci. 180(13), 2576–2587 (2010)
16. Di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Over-encryption: management of access control evolution on outsourced data. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 123–134. VLDB endowment (2007)
17. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. Computer 29, 38–47 (1996)
18. Asghar, M.R., Ion, M., Russello, G., Crispo, B.: Espoon: Enforcing encrypted security policies in outsourced environments. In: ARES, pp. 99–108. IEEE (2011)

Towards Decentralised Workflow Scheduling via a Rule-Driven Shared Space

Héctor Fernández[1], Marko Obrovac[2], and Cédric Tedeschi[2]

[1] VU University Amsterdam, The Netherlands
[2] IRISA. Université Rennes 1 / INRIA, France
hector.fernandez@vu.nl, {marko.obrovac,cedric.tedeschi}@inria.fr

Abstract. This paper addresses decentralised workflow scheduling, which calls for fulfilling two seemingly contradictory requirements: decentralisation and efficiency. We describe a two-layer architecture that allows decentralisation while making it possible for each scheduling decision to be taken based on a global perspective of the current state of resources. The first layer expresses the scheduling strategy on a global perspective, relying on a coordination space where workflows are first decomposed in tasks, and then tasks mapped onto resources. The second layer allows this global policy to be enacted in a fully-decentralised manner, based on a distributed hash table indexing resources, enhanced with advanced discovery mechanisms. Thus, in spite of decentralisation, the system is able to select the momentarily most appropriate resource for a given task, independently of the location of the provider of the resource. The framework's expectations in terms of scalability and network overhead are studied through simulation experiments.

1 Introduction

With the rise of Service Computing, a growing number of scientific applications are defined as *workflows of services*, *i.e.* temporal compositions thereof, which in turns intensifies the usage of distributed computing infrastructures, which consequently suffer from their centralisation, leading to reliability, privacy, and sustainability [1,2]. These situations led to the advocation of decentralised way of building these infrastructures, based on the federation of distributed computing resources [2]. Unfortunately, decentralisation and efficiency may be contradictory objectives. As described in [2], some *coordination* between participants/nodes of the federation is required to provide some efficiency in spite of decentralisation. In this position paper, we focus on **workflow scheduling**, *i.e.*, the process deciding which resource each task of some workflow has to be run on, and how to provide this *coordination* specifically for scheduling. Coordination is needed in order to ensure the consistency of the scheduling decisions taken independently by distinct nodes scheduling different tasks.

Let us briefly review few recent approaches for fully decentralised schedulers. Works such as [3] motivate the need for interlinking computing platforms through peering arrangements enabling resource sharing. Local schedulers are connected through *gateways* used to serve locally-unsatisfied requests. However, preferring locality might lead to an inefficient scheduling. Ranjan *et al.* [4] proposed a decentralised scheduler using a distributed hash table (DHT) split in regions through a P2P coordination space. The DHT acts as a distributed *blackboard* containing requests. Each region is managed by one peer, responsible for finding suitable resources in its part of the platform. Finally, works such as [5] propose gossip-based schemes to schedule computation-intensive jobs, where there are no predefined schedulers — any entity can schedule a job. However, the unstructured nature of gossiping protocols leads to only weak guarantees when searching a suitable resource for a task. The presented framework intends not only to decentralise the scheduling process, but also targets the possibility to support efficient scheduling algorithms through a coordination layer, built on top of the network, enabling a global knowledge of available resources.

Decentralisation and coordination cannot be tackled at once. We separate two concerns: *(i)* the specification of scheduling's logic; and *(ii)* its decentralised implementation. For the first problem, we need some high-level abstractions able to express these rules naturally. Rule-based programming, and in particular, the chemical programming model, offers adequate abstractions to specify coordination in distributed systems [6]. This model envisions a computation as a set of concurrent reactions between molecules of data. Formally speaking, the data is a multiset rewritten by a set of rules to be applied concurrently by distributed processes. Then, the second problem – the distributed implementation – can be refined as, how to distribute this multiset while being able to read and write it concurrently, so that the coordination is decentralised in its entirety. In this paper, we rely on HOCL (*Higher-Order Chemical Language*) [7], a rule-based language, enhanced with a chemistry-inspired execution model. According to the metaphor, molecules of data float in a solution, and, on collision, react according to reaction rules (the program) producing new molecules (the resulting data). In HOCL, the solution is a multiset containing molecules, and rewriting rules define reactions. The reactions take place in an implicitly parallel and non-deterministic way until no more reactions are possible — a stable state referred to as *inertia*. Let us consider the following chemical program which extracts the maximum value from a set of integers: $\textbf{replace}\ x, y\ \textbf{by}\ x\ \textbf{if}\ x \geq y\ \textbf{in}\ \langle 2, 4, 5, 7, 9 \rangle$. The rule specifies that any pair of integers inside the solution can react, consuming these two molecules and creating a new one with the highest value of the two. The HOCL execution model simply assumes reactants are captured atomically, so each molecule reacts only once. The exact degree of parallelism and the order in which the rule is applied is left to the implementor of a runtime supporting the language. Thus, one of the possible execution is the following: $\langle 2, 4, 5, 7, 9 \rangle \rightarrow^* \langle 4, 5, 9 \rangle \rightarrow \langle 5, 9 \rangle \rightarrow \langle 9 \rangle$. In case new molecules (here, integers) are dynamically inserted into the multiset, an *imbalance* may arise, triggering new reactions. Enabling persistent coordination amongst scheduling entities requests for such a concept. We here use HOCL to express the scheduling process.

In the following, we present a two-layered, fully-decentralised workflow scheduling framework. The top layer is a *chemically*-coordinated shared space where workflows are decomposed into tasks, which are mapped to resources. The bottom layer implements this shared space in a fully decentralised way, based on a peer-to-peer overlay network allowing the efficient storage and retrieval of molecules. The system proposed allows for a dynamic multiple-workflow scheduling. The conducted simulations allow to describe more precisely the scalability and overhead of such a platform. Section 2 describes our decentralised workflow scheduling system and its coordination model. Section 3 evaluates the performance and network overhead of the framework. Section 4 concludes.

2 A Distributed Shared Space for Workflow Scheduling

We now present a fully-decentralised, just-in-time, multiple-workflow scheduler, where the scheduling process is shared by a set of *chemical engines* running on every resource machine. As illustrated by Figure 1, the proposed system comprises a *communication* layer and a *coordination* layers.

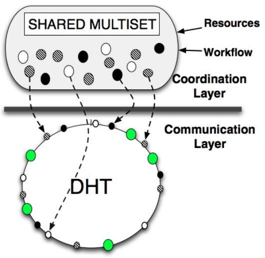


Fig. 1. Two-layer architecture

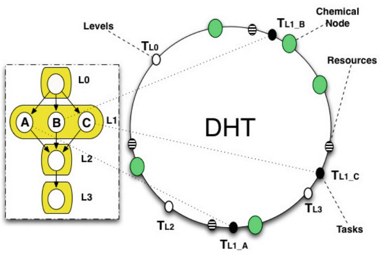


Fig. 2. Workflow decomposition

Abstracting out the underlying network topology and dealing with the large number of resources, chemical engines, referred to as *nodes* in the remainder, are connected through a DHT [8], constituting the **communication layer**, and illustrated in the lower part of Figure 1. The DHT handles the dynamic nature of the platform while preserving a uniform and efficient communication pattern.

The DHT allowing chemical engines to share data (molecules) in a scalable fashion, a *distributed shared multiset* will be created on top of it, to which nodes expose their molecules representing workflow tasks and resources, as shown in the upper side of Figure 1. Thus, nodes can use molecules they do not hold, making it possible to build a **coordination layer** based on a *distributed scheduling space*. Each chemical engine is provided with rules to decompose the workflow and schedule the tasks, acting by consuming and producing molecules within the shared multiset.

2.1 Molecule Types

There are three types of molecules in our system: molecules representing workflow levels, task molecules and resource molecules, respectively. Each molecule is assigned a unique identifier using the DHT's hash function. The molecule is then placed in the shared multiset (as depicted in Figure 2), by routing it to the appropriate node based on its identifier. Upon its entry in the system, a workflow is decomposed into levels by the entry node, producing **level molecules** (small white circles in Figure 2). Level molecules take the form $\text{LEVEL}{:}idLevel{:}\langle task_1, \ldots, task_n \rangle$, where *idLevel* identifies this level in the workflow, and $task_1, \ldots, task_n$ are the tasks the level comprises. Once it is the turn of a level to be processed, the node storing its molecule splits it into a set of **task molecules** (black circles in Figure 2), one per task. A task molecule takes the form $\text{TASK}{:}idTask{:}\langle cmd{:}res\_desc \rangle{:}\langle \text{DEST}{:}destTaskId, \ldots \rangle$, where *idTask* is the task's identifier, *cmd* denotes the actual service to invoke, *res_desc* is the description of the resource requirements, and the $\langle \text{DEST}{:}destTaskId, \ldots \rangle$ sub-solution specifies to which tasks the output of this task has to be sent. Physical resources are represented by **resource molecules** of the form $\text{RES}{:}idRes{:}\langle feature_1, \ldots, feature_n \rangle$, where *idRes* is the identifier of the resource and $feature_1, \ldots, feature_n$ are its current characteristics, such as the number of processors, the CPU load, or the memory usage. Unlike level and task molecules, resource molecules are not uniformly hashed. Instead, they are kept on the originating node (as suggested in Figure 2). Doing so keeps the network cost of updating a resource molecule at zero.

2.2 Workflow Scheduling Process

The framework proposed aims at providing a decentralised *just-in-time* task-to-resource mapping, on top of which more complex workflow scheduling heuristics can be implemented. It follows a rule-based (event-driven) execution model, in which a rule is triggered when a node receives a molecule or a new workflow. Three entities can trigger a rule: a workflow, a level molecule and a task molecule. The chemical rules used for the scheduling process are given in Algorithm 1(down). A workflow initially enters the platform by being sent to a given node, its *entry point*, which receives the workflow and decomposes it in levels, producing level molecules. Workflows are described using the *chemical workflow definition* illustrated in Algorithm 1(up), where the main solution is composed of as many *task molecules* as there are tasks participating in the workflow.

Upon the receipt of a workflow, a node triggers the *workflowDecomp* rule which reorganises the tasks represented as sub-solutions of the workflow representation into levels. To distinguish between levels which can be scheduled and those which have to wait, we use two types of molecules: $\text{LEVEL}{:}num{:}READY$ and $\text{LEVEL}{:}num$. The initial workflow decomposition, through the activation of the rule *workflowDecomp*, produces only $\text{LEVEL}{:}num$ molecules to indicate that none of the levels can be scheduled. However, as the scheduling goes on, these molecules will, one by one, turn into $\text{LEVEL}{:}num{:}READY$ molecules, indicating that the tasks of the previous levels have been completed and that the tasks of the next level can be scheduled for execution. To extract tasks from level molecules, a node uses the *levelDecomp* rule. This rule consumes a level molecule with its state set to $READY$, and produces as many task molecules

Algorithm 1. Chemical workflow representation (up); and Generic rules (down).

```
1.01 $\langle$ TASK : 1 : $\langle cmd_1 : res\_desc_1 \rangle$ : $\langle$DEST : 2, DEST : 3, ...$\rangle$,
1.02   TASK : 2 : $\langle cmd_2 : res\_desc_2 \rangle$ : $\langle$DEST : 4, ...$\rangle$,
1.03   TASK : 3 : $\langle cmd_3 : res\_desc_3 \rangle$ : $\langle$DEST : 4, ...$\rangle$,
1.04   TASK : 4 : $\langle cmd_4 : res\_desc_4 \rangle$ : $\langle\rangle$  $\rangle$
2.01 let workflowDecomp = replace $\langle$ TASK$_1$, ..., TASK$_n$ $\rangle$
2.02           by LEVEL:1:$\langle$ TASK$_1$ $\rangle$, ..., LEVEL:L:$\langle$ TASK$_n$ $\rangle$
2.03 let levelDecomp = replace-one LEVEL:num:READY:$\langle$ TASK$_1$, ... ,TASK$_n$ $\rangle$
2.04           by TASK$_1$, ... ,TASK$_n$
2.05 let mapTaskRes = replace TASK$_i$, RES$_j$  by system.deploy(TASK$_i$, RES$_j$ )
2.06           if (TASK$_i$.isCompatibleWith(RES$_j$))
```

as there are tasks in the given level (Algorithm 1, line 2.03). Upon the receipt of a task molecule, a node uses the *mapTaskRes* rule to map the task to the best resource it can find (Algorithm 1, line 2.05). For this purpose, we use a second, order-preserving DHT layer, physically matching the first one, to store *meta-molecules* — *pointers* to resource molecules. A meta-molecule is placed in this layer according to its value, *i.e.* cpu usage. When a node looks for a resource to execute its task on, it sends a request in this second layer. Using the resources' requirements indicated in the task molecule, the second DHT layer is scanned by a range query (whose complexity is typically in $O(log^2(n))$) [9], such as $cpu < 80\%$, reflecting the **if**-clause of the *mapTaskRes* rule. If a matching resource molecule is found, the rule produces a molecule that deploys the given task onto the resource thus found (denoted by the *system.deploy()* molecule in Algorithm 1). When all of its tasks have been completed, the node holding the (currently active) level molecule retrieves the inactive molecule of the next level, to allow the next level to be processed. Once the tasks of the last level have completed, its responsible node collects all of the results and transfers them to the entry node, which delivers them to the client that submitted the workflow for execution. Note that, due to decentralisation, multiple workflows can be scheduled at the same time. They can be processed in parallel, each being managed by a different set of nodes, while the final scheduling decisions are still taken on a global perspective. They are independently decomposed, each on a different entry-point node, but their tasks compete collectively for resources.

3 Preliminary Evaluation

We built a discrete-time simulator in Python to simulate the framework when multiple randomly-generated workflows are executed. As shown by Figure 3, increasing the number of nodes has an impact on the time taken to schedule workflows which is limited, as the routing's cost grows logarithmically with the number of nodes. Also, thanks to decentralisation, increasing the number of workflows does not augment much the time to solve them, as it enables a high degree of parallelism. Figure 4 shows that the logarithmic nature of the routing limits the impact of increasing the number of nodes on the network congestion. The number of messages naturally increases proportionally with the number of workflows, as the scheduling of a task relies on resource retrieval, which needs $O(log^2(n))$ messages to complete. Finally, Figure 5 suggests that the number of messages sent per node drastically reduces when more nodes take part in the scheduling process, even when an elevated number of workflows is present.

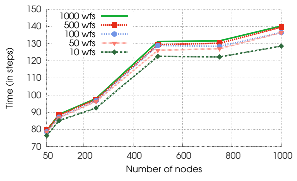


Fig. 3. Execution time

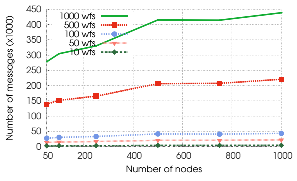


Fig. 4. Network traffic

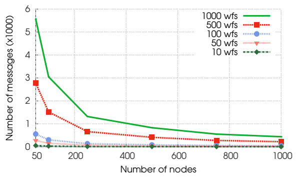


Fig. 5. Traffic per node

4 Conclusion

To address the contradictory objectives of decentralisation and efficiency of scheduling, we have proposed a high-level coordination mechanism relying on a chemistry-inspired, rule-based programming model, and supported by a DHT-based decentralised architecture to retrieve and match tasks and resources efficiently. This *DHT-driven* coordination enables *just-in-time* scheduling, allowing to match each task to the momentarily best resource available. Simulations were conducted, showing further the feasibility and scalability of the approach.

References

1. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: A berkeley view of cloud computing. tech. rep., UC Berkeley (2009)
2. Marinos, A., Briscoe, G.: Community cloud computing. In: CloudCom (2009)
3. Leal, K., Huedo, E., Llorente, I.M.: A decentralized model for scheduling independent tasks in federated grids. In: FGCS, vol. 25 (2009)
4. Ranjan, R., Rahman, M., Buyya, R.: A decentralized and cooperative workflow scheduling algorithm. In: CCGRID. IEEE (2008)
5. Vasilakos, X., Sacha, J., Pierre, G.: Decentralized As-Soon-As-Possible grid scheduling: A feasibility study. In: ICCCN (2010)
6. Fernández, H., Tedeschi, C., Priol, T.: A Chemistry-Inspired Workflow Management System for Scientific Applications in Clouds. In: e-Science (2011)
7. Banâtre, J., Fradet, P., Radenac, Y.: Generalised multisets for chemical programming. Mathematical Structures in Computer Science 16(4) (2006)
8. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
9. Chen, L., Candan, K., Tatemura, J., Agrawal, D., Cavendish, D.: On overlay schemes to support point-in-range queries for scalable grid resource discovery. In: International Conference on Peer-to-Peer Coomputing (2005)

Strengthening Consistency in the Cassandra Distributed Key-Value Store

Panagiotis Garefalakis, Panagiotis Papadopoulos, Ioannis Manousakis, and Kostas Magoutis

Institute of Computer Science
Foundation for Research and Technology-Hellas
Heraklion GR-70013, Greece
{pgaref,panpap,jmanous,magoutis}@ics.forth.gr

Abstract. Distributed highly-available key-value stores have emerged as important building blocks for applications handling large amounts of data. The Apache Cassandra system is one such popular store combining a key distribution mechanism based on consistent hashing with eventually-consistent data replication and membership mechanisms. Cassandra fits well applications that share its semantics but is a poor choice for traditional applications that require strong data consistency. In this work we strengthen the consistency of Cassandra through the use of appropriate components: the Oracle Berkeley DB Java Edition High Availability storage engine for data replication and a replicated directory for maintaining membership information. The first component ensures that data replicas remain consistent despite failures. The second component simplifies Cassandra's membership, improving its consistency and availability. In this short note we argue that the resulting system fits a wider range of applications, and is more robust and easier to reason about.

1 Introduction

The ability to perform large-scale data analytics over huge data sets has in the past decade proved to be a competitive advantage in a wide range of industries (retail, telecom, defence, etc.). In response to this trend, the research community and the IT industry have proposed a number of platforms to facilitate large-scale data analytics. Such platforms include a new class of databases, often referred to as NoSQL data stores, which trade the expressive power and strong semantics of long established SQL databases for the specialization, scalability, high availability, and often relaxed consistency of their simpler designs.

Companies such as Amazon [1] and Google [2] and open-source communities such as Apache [3] have adopted and advanced this trend. Many of these systems achieve availability and fault-tolerance through data replication. Google's BigTable [2] is an early approach that helped define the space of NoSQL *key-value* data stores. Amazon's Dynamo [1] is another approach that offered an eventually consistent replication mechanism with tunable consistency levels. Dynamo's open-source variant Cassandra [3] combined Dynamo's consistency mechanisms

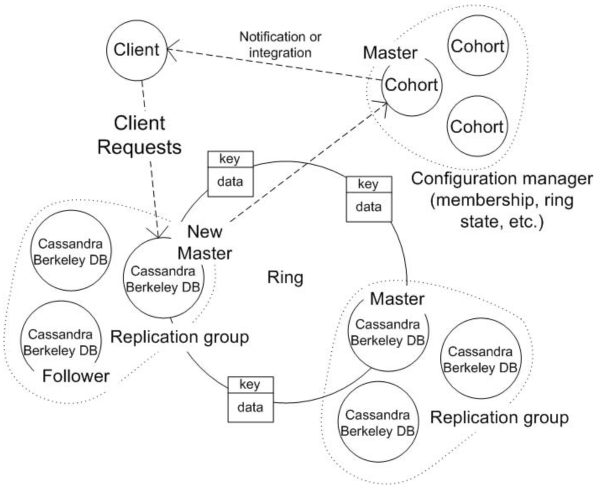


Fig. 1. System architecture

with a BigTable-like data schema. Cassandra uses consistent hashing to ensure a good distribution of key ranges (data partitions, or *shards*) to storage nodes.

Cassandra works well with applications that share its relaxed semantics (such as maintaining customer carts in online stores [1]) but is not a good fit for more traditional applications requiring strong consistency. We recently decided to embark on a re-design of Cassandra that preserves some of its features (such as its data partitioning based on consistent hashing) but replaces others with the aim of strengthening consistency. Our design does not utilize multiple masters on concurrent updates to a shard or techniques such as hinted handoff [1]. Instead, service availability requires that a single master per shard (part of a self-organized replication group) be available and its identity known to I/O coordinators. We reduce intervals of unavailability by aggressively publishing configuration updates. Furthermore we improve performance by using client-coordinated I/O, avoiding a forwarding step in Cassandra's original I/O path. In summary, our re-design centers on:

– Replacing Cassandra's data replication mechanism with the highly available Oracle Berkeley DB Java Edition (JE) High Availability (HA) key-value storage engine (hereafter abbreviated as BDB). Our design simplifies Cassandra while at the same time it strengthens its data consistency guarantees.
– Enhancing Cassandra's membership protocol with a highly available Paxos-based directory accessible to clients. In this way, replica group reconfigurations are rapidly propagated to clients, reducing periods of unavailability.

The resulting system is simpler to reason about and backwards-compatible with original Cassandra applications. While we expect that dropping the eventual consistency model may result in reduced availability in certain cases, we try to make up by focusing on reducing recovery time of the I/O path after a failure.

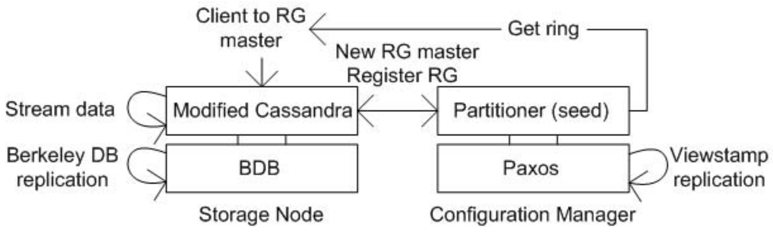


Fig. 2. System components and their interactions

The rest of the paper is organized as follows: In Section 2 we describe the overall design and in Section 3 we provide details of our implementation and preliminary results. In Section 4 we describe related work and in Section 5 directions of ongoing and future work. Finally in section 6 we conclude.

2 Design

Our system architecture is depicted in Figure 1. We preserve the Thrift-based client API for compatibility with existing Cassandra applications. We also maintain Cassandra's ring-based consistent hashing mechanism (where keys and storage nodes both map onto a circular ring [1]) but modify it to map each key to a BDB replication group (RG) instead of a single node. BDB implements a B+-tree indexed key-value store via master-based replication of a transaction log, using Paxos for reconfiguration. In our setup, all accesses go through the master (ensuring order) while writes are considered durable when in memory and acknowledged by all replicas. Periodically, replicas flush their memory buffers to disk. These settings offer a strong level of consistency with a slightly weaker (but sufficient for practical purposes) notion of durability [4].

Each node in an RG runs a software stack comprising a modified Cassandra with an embedded BDB (left of Figure 2). On a master, Cassandra is active and serves read/write requests; on a follower, Cassandra is inactive until elected master (election is performed by BDB and its result communicated to Cassandra via an upcall). The ring state is stored on a Configuration Manager (or CM, right of Figure 2). The CM complements Cassandra's original metadata service which uses a gossip-based protocol [3]. It combines a *partitioner* (a module that chooses tokens for new RGs on the ring) with a primary-backup viewstamp replication [5] scheme where a group of nodes (termed *cohorts*) exchange state updates over the network. The CM can be thought of as a highly-available alternative to Cassandra's *seed* nodes. It contains information about all RGs, such as addresses and status (master or follower), and corresponding tokens. Any change in the status of RGs (new RG inserted in the ring or existing RG changes master) is reported to the CM via RPC. The CM is queried by clients to identify the current master of an RG (by token).

We improve data consistency over original Cassandra by prohibiting multi-master updates. For a client to successfully issue an I/O operation, it must have access to the master node of the corresponding RG. Causes of unavailability include RG reconfiguration actions after failures and delays in the new ring state propagating to clients. Our implementation supports faster client updates by either eager notifications by the CM [6] or by integrating with the CM. Additionally, clients can explicitly request RG reconfiguration actions if they suspect partial failure (i.e., a master visible to the RG but not to the client).

Our partitioner subdivides the ring to a fixed number of key ranges and assigns node tokens to key-range boundaries. This method has previously been shown to exhibit advantages over alternative approaches [1]. Each key range in our system corresponds to a different BDB database, the total number of key ranges on the ring being a configuration parameter. Finally, data movement (streaming) between storage nodes takes place when bootstrapping a new RG.

3 Implementation and Preliminary Results

Our implementation replaces the original Cassandra storage backend with Oracle Berkeley DB JE HA. One of the challenges we faced was bridging Cassandra's rich data model (involving column families, column qualifiers, and versions [3]) with BDB's simple key-value get/put interface where both key and value are opaque one-dimensional entities. Our first approach mapped each Cassandra cell (row key, column family, column qualifier) to a separate BDB entry by concatenating all row attributes into a larger unique key. The problem we faced with this model was the explosion in the number of BDB entries and the associated (indexing, lookup, etc.) overhead. Our second approach maps the Cassandra row-key to a BDB key (one-to-one) and stores in the BDB value a serialized HashMap of the column structure. Accessing a row requires a lookup for the row and subsequent lookup in the HashMap structure to locate the appropriate data cell. Our current implementation following this approach performs well in the general case, with the exception of frequent updates/appends to large rows (the entire row has to be retrieved, modified, then written back to BDB). This is a case where Cassandra's native no-overwrite storage backend is more efficient by writing the update directly to storage, avoiding the read-modify-write cycle.

Our Configuration Manager (CM) uses a specially developed Cassandra partitioner to maintain RG identities, master and follower IPs, RG tokens, and the key ranges on the ring. We decided to use actual rather than elastic IP addresses due to the long reassignment delays we observed with the latter on certain Cloud environments. Each RG stores its identifier and token in a special BDB table so that a newly elected RG master can retrieve it and identify itself to the CM. The CM exports two RPC APIs to storage nodes: *register/deregister RG*, *new master for RG*; and one to both storage nodes and clients: *get ring info*. The CM achieves high availability of the ring state via viewstamp replication [5, 7].

Preliminary results with the Yahoo Cloud Serving Benchmark (YCSB) over a cluster of six Cassandra nodes (single-replica RGs) on Flexiant VMs with 2 CPUs, 2GB memory, and a 20GB remotely-mounted disk indicate improvement by 26% and 30% in average response time and throughput respectively, compared

Table 1. YCSB read-only workload

	Throughput (ops/sec)	Read latency (average, ms)	Read latency (99 percentile, ms)
Original Cassandra	317	3.1	4
Client-coordinated I/O	412	2.3	3

to original Cassandra (Table 1 summarizes our results). This benefit is primarily due to client-coordination of requests. Our ongoing evaluation will further focus on system availability under failures and scalability with larger configurations.

4 Related Work

Our system is related to several existing distributed NoSQL key-vale stores [1–3] implementing a wide range of semantics, some of them using the Paxos algorithm [8] as a building block [6, 9, 10]. Most NoSQL systems rely on some form of relaxed consistency to maintain data replicas and reserve Paxos to the implementation of a global state module [9, 10] for storing infrequently updated configuration metadata or to provide a distributed lock service [6]. Exposing storage metadata information to clients has been proposed in the past [1, 9, 11], although the scalability of updates to that state has been a challenge.

Perhaps the closest approaches to ours are Scatter [12], ID-Replication [13], and Oracle's NoSQL database [11]. All these systems use consistent hashing and self-managing replication groups. Scatter and ID-Replication target planetary-scale rather than enterprise data services and thus focus more on system behavior under high churn than speed at which clients are notified of configuration changes. Just as we do, Oracle NoSQL leverages the Oracle Berkeley DB (BDB) JE HA storage engine and maintains information about data partitions and replica groups across all clients. A key difference with our system is that whereas Oracle NoSQL piggybacks state updates in response to data operations, our clients have direct access to ring state in the CM, receive immediate notification after failures, and can request reconfiguration actions if they suspect a partial failure. We are aware of an HA monitor component that helps Oracle NoSQL clients locate RG masters after a failure, but were unable to find detailed information on how it operates.

5 Future Work

Integrating the CM service into Cassandra clients (making each client a participant in the viewstamp replication protocol) raises scalability issues. We plan to investigate the scalability of our approach as well as the availability of the resulting system under a variety of scenarios. Another research challenge is in provisioning storage nodes for replication groups to be added to a growing cluster. Assuming that storage nodes come in the form of virtual machines (VMs) with local or remote storage on Cloud infrastructure, we need to ensure that nodes in an RG fail independently (easier to reason about in a private rather than a public Cloud setting). Elasticity is another area we plan to focus on. A brute force approach of streaming a number of key ranges (databases) to a newly joining RG is a starting point but our focus will be on alternatives that exploit replication mechanisms [14].

6 Conclusions

In this short note we described a re-design of the Apache Cassandra NoSQL system aiming to strengthen its consistency while preserving its key distribution mechanism. Replacing its eventually-consistent replication protocol by the Oracle Berkeley DB JE HA component simplifies the system while making it applicable to a wider range of applications. A new membership protocol further increases system robustness. A first prototype of the system is ready for evaluation while the development of more advanced functionality is currently underway. This work was supported by the CumuloNimbo (FP7-257993) and PaaSage (FP7-317715) EU projects.

References

1. DeCandia, G., et al.: Dynamo: Amazon's Highly Available Key-value Store. ACM SIGOPS Operating Systems Review 41, 205–220 (2007)
2. Chang, F., et al.: Bigtable: A Distributed Storage System for Structured Data. ACM Transactions on Computer Systems (TOCS) 26, 4 (2008)
3. Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. ACM SIGOPS Operating Systems Review 44, 35 (2010)
4. Birman, K., et al.: Overcoming CAP with Consistent Soft-State Replication. IEEE Computer (45) 50–58
5. Mazieres, D.: Paxos Made Practical. Technical report (2007)
6. Burrows, M.: The Chubby Lock Service for Loosely-coupled Distributed Systems. In: Proc. of the 7th Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA (2006)
7. Oki, B.: Liskov. B: Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In: Proc. of the 7th ACM Symposium on Principles of Distributed Domputing (PODC), Toronto, Canada (1988)
8. Lamport, L.: Paxos made simple. ACM SIGACT News 32, 18–25 (2001)
9. Lee, E., Thekkath, C.: Petal: Distributed Virtual Disks. ACM SIGOPS Operating Systems Review 30, 84–92 (1996)
10. MacCormick, J., et al.: Niobe: A Practical Replication Protocol. ACM Transactions on Storage (TOS) 3 (2008)
11. Oracle, Inc.: Oracle NoSQL Database: An Oracle White Paper (2011)
12. Glendenning, L.: et al.: Scalable Consistency in Scatter. In: Proc. of 23rd ACM Symposium on Operating Systems Principles (SOSP), Cascais, Portugal (2011)
13. Shafaat, T.M., Ahmad, B., Haridi, S.: ID-Replication for Structured Peer-to-Peer Systems. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 364–376. Springer, Heidelberg (2012)
14. Lorch, J., et al.: The SMART Way to Migrate Replicated Stateful Services. In: Proc. of EuroSys 2006, Leuven, Belgium (2006)

Distributed Architecture for a Peer-to-Peer-Based Virtual Microscope

Andreas Jaegermann[1], Timm J. Filler[1], and Michael Schoettner[2]

[1] Department of Anatomy, University of Duesseldorf
[2] Department of Computer Science, University of Duesseldorf

Abstract. Virtual microscopes are commonly used in medical education. They provide a platform for distributing whole slide images (WSI) with several GB size to exploring students. Even in courses with a few hundred students and dozens of WSI the network traffic may be high, but it will vastly increase, when the system is opened to access from the Internet. The same applies to user-generated content like interactive annotations (each student generates approx. 200 labels per term). In a collection that consists of several thousand WSI, which need to be annotated for training or quiz-based purposes, there will be millions of user contributions. In an abstract view users navigate through a universe of WSI and annotations and may meet other users watching the same or related WSI. This paper presents a distributed architecture build on PathFinder for Internet-based virtual microscopy addressing the challenges of distributing tightly connected data chunks on an overlay network consisting of random graphs.

1 Introduction

A virtual microscope is a system, which provides digitalized slides for a large number of simultaneously accessing clients, similar to geographic applications like Google Maps. Slide scanners are used to digitize glass slides and create proprietary slide file formats depending on the scanners' manufacturer [6]. The file size varies with the dimensions of the scanned specimen and ranges from a few MB to several GB. Virtual microscopes have been developed more than a decade ago to fit the needs of pathologists and have been adopted for educational purposes in microscopic anatomy in the last few years [5]. As the typical setup consists of a server accessing the file resources and delivering them on demand to any connected client via either proprietary communication protocols or http/ftp, the system's transmission capacity is obviously limited by the server's network connection. Usually, all systems are designed to work with classroom-sized courses and some are capable of handling up to a few hundred clients connecting simultaneously. The number of virtual microscopes increased steadily, but nearly every solution suffers from one main disadvantage: if the number of histological slides in a collection exceeds a few hundred, the labeling process to provide students with detailed, qualified information about different structures is extremely time consuming for lecturers. However, the annotations created in a slide are essential to virtual microscopes because they extend the virtual microscope far beyond the capabilities of a real light microscope. Single cells, strata or complex tissue formations are examples of structures to be labeled. Not only are these annotations some explanations in an otherwise silent image, but rather do they provide a feedback on the personal learning progress for students. On the other hand, an increasing number of already validated annotations will help to evaluate new and varying annotations. Additionally, annotations can be used in online exams for automated evaluation.

In a common client/server setup the number of connecting clients is limited to a few hundred at maximum due to the hardware and connection requirements. Meeting more substantial requirements often leads to disproportional costs. To address these challenges we develop a system that supports a distributed architecture for transmitting digitized slides as well as a context-based user platform for real-time interaction.

To improve the scalability of virtual microscopes accompanied by an elevation of interactivity between users among themselves and the network, we transformed the typical client/server setup into a (managed) peer-to-peer architecture. Our solution is based on PathFinder, a peer-to-peer overlay network that relies on random graphs and provides an efficient combination of long range queries and single key-value lookups [4].

This paper is organized as follows. In Sect. 2 we present an overview of our distributed virtual microscope (DVM). Section 3 catches up with related work in the field of peer-to-peer systems. In Sect. 4 the advantages of PathFinder for our approach and the necessary adaptions are explained. Conclusions are presented in Sect. 5.

2 Architecture of the Distributed Virtual Microscope

The WSI are stored in tiles and a fully digitalized slide with a decent Z-stack in the highest magnification can easily produce up to 0.5 million tiles. A whole collection of a few thousand slides will lead to billions of images that need to be transmitted across the DVM along with corresponding annotations, messages and status information.

New WSI can only be stored in the network by selected nodes to ensure proper quality and compliant preparation.

Each node participating in the DVM has a standardized architecture and provides different services to the user and/or the network itself. It has to contribute a variable amount of storage capacity to be used by the DVM for the distribution of WSI. Additionally the user can store needed (or predictably needed) data. To assure data integrity the storage container acts like a black box and no standard user is allowed to modify any stored objects or store new objects.

The bottom most layer of a node (see Fig. 1) contains the overlay network responsible for handling joins and deletions of nodes. Additionally it detects and handles crashes as well as it adapts the network size. It manages the queries to physically locate data, nodes and users.

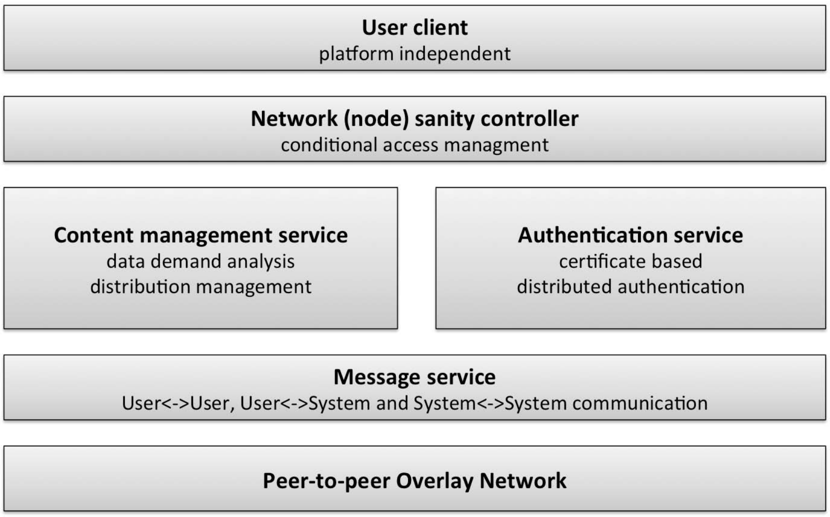

User client
platform independent
Network (node) sanity controller
conditional access managment
Content management service
data demand analysis
distribution management
Authentication service
certificate based
distributed authentication
Message service
User<->User, User<->System and System<->System communication
Peer-to-peer Overlay Network

Fig. 1. Modular layered architecture of node in the DVM

The message service layer uses the overlay network to transmit messages between different users, users and nodes and among nodes themselves. Messages are generated in upper layers and may contain different types of information or requests. This service uses fingerprints and checkpointing to recover from a partial or complete loss of messages.

As the heart of a node the content management controller decides upon locally observed and from the neighborhood acquired data, what objects should be moved to or from the local storage container. By calculating performance values from network and storage parameters it can decide to ask neighbors for help while processing complex requests. In addition it is responsible for generating requests to increase the size of the immediate neighborhood if too much load is generated on the node and its existing neighbors.

The authentication service is tightly controlling user access and responsible for distributed authentication in case the authentication server is not available. Therefore signed key pairs from a trusted resource are created for each node. For distributed authentication each participating peer can check the certificate chain to decide if single requests should be executed or declined. As this is a only fallback mechanism to keep the network operational in case of a server failure no new accounts can be created at that time.

The next higher authority for each node is the sanity controller that has to make sure all needed local services are running and that the node has a working connection to a sane neighborhood.

On top there is just a graphical user interface to allow users to interact with the DVM. Slides can be selected, annotations can be created and messages can be send to other users.

3 Searchable Peer-to-Peer Overlay Network

Multicast protocols to transmit a tiled WSI were discarded as the data in DVM is no longer located at a single source and the complete transmission of a single WSI is, due to regions of interest, commonly not needed.

As we designed DVM to be a peer-to-peer (P2P) system build on hundreds of thousands of heterogeneously distributed nodes of varying performance, we were in need of a search that can be used to locate specific tiles of a WSI (a specific hash can be assigned to each tile) as well as processing exhaustive searches e.g. to find WSIs matching given criteria.

The decision to create a structured or an unstructured network while designing DVM was strengthened by the benefits of distributed hash tables (DHT). DHTs offer some advantages over unstructured P2P system (like Gnutella or Napster) as they include load balancing, efficient routing and resilience against node failures.

Although the scientific community has seen many implementations and enhancements to DHTs [8,7] all designs rely on randomizing hash functions for inserting and searching keys in the hash table. This hashing is the main advantage for the efficient value retrievals of DHTs for a given key.

Other systems on unstructured networks like SkipGraphs [1], Mercury [3] and VoroNet [2] provide multiple-attribute range queries. Compared to DHTs they are less fault tolerant (or require more effort to equalize) and are not as scalable (concerning the search performance for known hashes).

BubbleStorm [9] is an overlay network capable of exhaustive searches in large-scale heterogeneous environments with adjustable probabilistic guarantees. The underlying structure of a well connected multigraph where each node has an even degree is extremely resilient to node crashes, as they do not destroy the search paths, and highly efficient concerning multiple-attribute queries. The only thing missing is the efficiency of a value lookup (like in a DHT) if the hash is known. PathFinder [4] fills this spot by augmenting BubbleStorms random graphs with a deterministic lookup mechanism.

4 Distributed Slides

After thoroughly comparing different approaches of overlay networks and DHT augmentations regarding query performance, routing and traffic overhead, we considered PathFinder to be a good starting point. The heterogeneous context of DVM regarding hardware, operating systems and network connections is incorporated into the topology of PathFinder. Designed as a connected multigraph PathFinder models this structure as a circular permutation only modified by *JOIN*, *LEAVE* and *CRASH* events of nodes and uses subgraphs of controlled size for data retrieval by solving the rendezvous problem. The proportionality of each nodes degree and its capacity leads to fixed size routing tables and locally controlled workload, which is highly appreciated. If a nodes workload increases, the DVM's content management service can simply request to add more nodes to the neighborhood.

The DHT-like hash lookups of PathFinder are realized by using two pseudo-random number generators (PRNG) that produce a deterministic sequence of numbers when initialized with a specific number. The first PRNG is used to calculate the number of neighbors each virtual node has by producing Poisson distributed numbers. The second one generates numbers that are treated as the node IDs of the neighbors for the current virtual node. This procedure enables each node to determine the neighbors for any node in the network without network communication. To find a path from the seeker u to node v containing the required information, the physical node responsible for u computes the neighbors of u (what is already accomplished, since they are contained in the routing table) and v, looking for matching neighbors while increasing the distance to u and v in each step by 1 if no match is found.

One key point in distributing slides is the connection between its tiles and the consequences this implies for the computed hash of the tiles. It is necessary that the hash for a tile can be calculated from the WSI it belongs to and its position therein (see Tab. 1).

Table 1. Definition for Naming Tiles in DVM

WSI Identifier	Magfication Layer	Z-Stack Layer	Tile Position X	Tile Position Y
32 Bit	8 Bit	8 Bit	16 Bit	16 Bit

Additionally, we necessarily want to store multiple copies of a tile in the DVM to reduce the load on single nodes containing some or many popular tiles. To achieve this, we change the definition of a virtual node. In PathFinder a virtual nodes is located on exactly one physical peer. We augment the definition in a way that a virtual node can be spared over one or more physical nodes.

What needs to be changed in the original concept is that a virtual node has to keep track of the physical nodes it relies on. Therefore each physical node maintains a list of physical nodes sharing a virtual node with it. Incoming requests for an object can be passed from one physical node to his equally responsible neighbor if allowed by performance information from the content management service.

Further, a virtual node must be able to acquire more capacities as new physical nodes join the system or at least move some of its content to adjacent virtual nodes. The former is possible only if a new node joins as a direct neighbor and the latter is already implemented in PathFinder as this is just the creating of a new virtual node resulting in a modification of the hash space. The new virtual node can be moved to other physical nodes if required by means of performance.

5 Conclusion

In this paper we have presented DVM, a distributed virtual microscope based on the PathFinder overlay network. We showed that the combined strengths of efficient hash value lookups as well as efficient long range queries in a single overlay network made PathFinder the superior starting point for DVM. The integrated possibility to locally manage any nodes workload with respect to asynchronous connection bandwidths was a major argument, too. We have also presented DVM-specific extensions to PathFinders structure definition by augmenting the distribution of virtual nodes across physical nodes and choosing a proper basis for the hash function to reflect the correlation between different tiles and their position in a WSI. The random graph based overlay network constructed by PathFinder will provide optimized search results in the application scope of a distributed virtual microscope. Another advantage of our structure enhancement is that many operations can be implemented using parallel algorithms what will most probably lead to at least slightly increased performance. The development process of the distributed architecture will be finished shortly and practical testing will allow further evaluation, adaption and optimization from the gained experiences.

References

1. Aspnes, J., Shah, G.: Skip graphs. In: SODA, pp. 384–393 (2003)
2. Beaumont, O., Kermarrec, A.-M., Marchal, L., Rivière, É.: Voronet: A scalable object network based on voronoi tessellations. In: Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS 2007). Society Press (2007)
3. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: Supporting scalable multi-attribute range queries. In: SIGCOMM 2004, pp. 353–366 (2004)
4. Bradler, D., Krumov, L., Mühlhäuser, M., Kangasharju, J.: PathFinder: Efficient lookups and efficient search in peer-to-peer networks. In: Aguilera, M.K., Yu, H., Vaidya, N.H., Srinivasan, V., Choudhury, R.R. (eds.) ICDCN 2011. LNCS, vol. 6522, pp. 77–82. Springer, Heidelberg (2011)
5. Drake, R.L., McBride, J.M., Lachman, N., Pawlina, W.: Medical education in the anatomical sciences: The winds of change continue to blow. Anat. Sci. Educ. 2, 253–259 (2009)
6. Rojo, M.G., García, G.B., Mateos, C.P., García, J.G., Vincente, M.C.: Critical comparison of 31 commercially available digital slide systems in pathology. International Journal of Surgical Pathology 14(4), 285–305 (2006)
7. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, pp. 329–350. Springer, Heidelberg (2001)
8. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM 2001, pp. 149–160 (2001)
9. Terpstra, W.W., Leng, C., Buchmann, A.P.: Bubblestorm: Analysis of probabilistic exhaustive search in a heterogeneous peer-to-peer system. Technical Report TUD-CS-2007-2, Technische Universität Darmstadt (2007)

Author Index