Processes Are Data: A Programming Model for Distributed Applications

Alexander Böhm and Carl-Christian Kanne

University of Mannheim
Mannheim, Germany
{alex,cc}@db.informatik.uni-mannheim.de

Abstract. Many modern distributed applications employ protocols based on XML messages. Typical architectures for these applications follow an approach where messages are organized in queues, state is stored in DBMS, and application code is written in imperative languages. As a result, much developer productivity and system performance is wasted on handling conversions between the various data models (XML messages, objects, relations), and reliably managing persistent state for application instances. This overhead turns application servers into data management servers instead of process servers. We show how this model can be greatly improved by changing two aspects. Firstly, by using a declarative rule language to describe the processing logic. Secondly, by providing a single, unified data model based on XML messages that covers all kinds of data encountered, including process state. We discuss the resulting design choices for compile-time and run-time systems, and show and experimentally evaluate the performance improvements made possible.

1 Introduction

Many modern distributed applications employ protocols based on XML messages (e.g. Web Services [2]). Typical architectures for these applications follow an approach where messages are organized in queues, state is stored in DBMS, and application code is written in imperative languages. As a result, much developer productivity and system performance is wasted on handling conversions between the various data models (XML messages, objects, relations), and reliably managing persistent state for application instances. This overhead turns application servers into data management servers instead of process servers.

The goal of the Demaq project is to investigate a novel way of developing distributed applications. We believe that the state of the art can be greatly improved by changing two aspects. Firstly, by using a declarative rule language to describe the processing logic. Secondly, by providing a single, unified data model based on XML messages that covers all kinds of data encountered, including process state. This novel programming model leverages database technology for managing persistent application state and message processing. Our approach is motivated by the observation that the behavior of a node in a message-driven application is determined by all the messages it has seen so far. The externally visible behavior of the node is represented by messages it sends to other nodes. Hence, the node's processing logic can be specified as a declarative query against the message history, and the result is a set of new messages to send. This way, we turn application processing into a declarative query processing problem. In an initial sketch of our vision of Demaq [8], we focused on the general concepts and language syntax, motivated by simple and elegant application specification and developer productivity.

In this paper, we turn towards the performance improvements made possible by our simple message-focused model. We look under the hood of our execution system, reviewing the involved design choices, and introduce techniques that allow to tune the language semantics and execution model to achieve highly concurrent execution and scalability.

Our main contributions include:

– We present the design of a solid transactional execution model for declarative messaging applications.
– We introduce a method to decouple garbage collection of irrelevant messages by allowing for the declarative specification of the relevant suffix of the message history in application programs.
– We introduce a variant of snapshot isolation optimized for message queues to improve concurrency of our run-time system.
– We present experimental results, comparing our implementation to a commercial application server.

The remainder of the paper is organized as follows. We present the elements of our programming model in Sec. 2. We elaborate on our declarative message processing language in Sec. 3, and discuss the corresponding execution model in Sec. 4. Sec. 5 gives a brief overview of our system implementation. Sec. 6 presents experimental results that show significant improvements in performance and scalability compared to a commercial application server. After briefly reviewing related work in Sec. 7, we conclude in Sec. 8.

2 Programming Model

Our programming model describes the application logic of a node in a distributed XML messaging application using two fundamental components.

XML message queues provide asynchronous communication facilities and allow for reliable and persistent message storage. *Declarative rules* operate on these message queues and are used to implement the application logic. Every rule specifies how to react to a message that arrives at a particular queue by creating another message (Figure 1).

XML Message Queues. Distributed messaging applications are based on *asynchronous* data exchange. Queue data structures are ideally suited for message management as they offer efficient message storage and retrieval operations while preserving the order of incoming data.

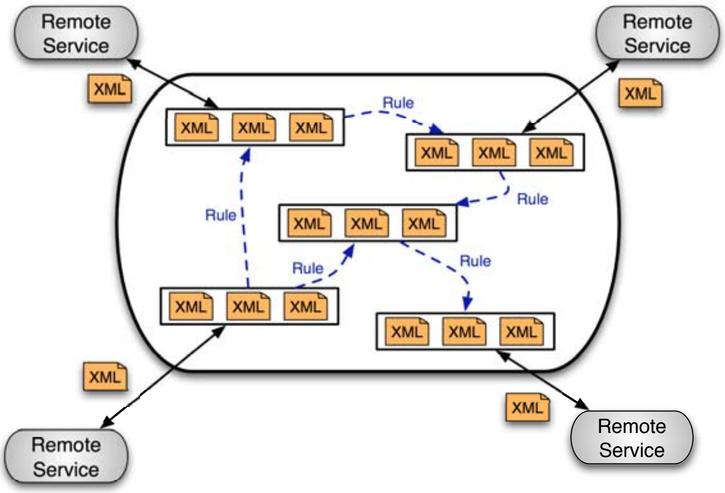

Fig. 1. Programming model

Apart from their typical functionality as intermediate message buffers, our model also uses persistent queues as internal representation of the application state. This approach is based on the observation that the state of every application instance in an individual node results from the messages sent to and received from its communication partners. Instead of *materializing* this state using a separate data model, it can alternatively be retrieved by *querying* the message flow. Hence, instead of a traditional queue semantics where messages are deleted after processing, messages have to be retained as long as they are necessary to compute the state of an instance. We discuss garbage collection in Sec. 3.2.

As logical data model, our message queues use the XQuery Data Model (XDM) [13]. Building on XDM allows us to incorporate existing XML processing systems such as stores and query processors without type system mismatches. For our purposes, XDM is particularly suited, as its fundamental type is the ordered sequence, which nicely captures message queue structure.

Declarative Rules. In our model, the processing logic of an application is specified as a set of declarative rules that operate on messages and queues. Each rule describes how to react to a single kind of event - the insertion of a new message into a queue. Depending on the structure and content of this message, rule execution results in the creation of new messages. These result messages can either become the input for other rules, or be sent to a remote system using queue-based communication facilities.

Our rule language is built on the foundation of XQuery [7]. It allows developers to directly access and interact with the XML messages stored in the queues. Thus, there is no mismatch between the type system of the application programs and the underlying communication format. Additionally, the content of messages and queues can be directly accessed within application programs without crossing system boundaries or requiring complex, intermediate APIs.

3 Language

Every Demaq application consists of three key components which we discuss in the following sections. These are message queues, application rules and user-defined message groups, called *slicings*.

3.1 Queues

Our programming model incorporates two different kinds of queues. *Gateway queues* provide communication facilities for the interaction with remote systems. There are two different kinds of gateway queues, *incoming* and *outgoing* ones. Messages that are placed into outgoing gateway queues are sent, while incoming gateway queues contain messages that have been received from remote nodes.

Queues are also used as persistent storage containers. These *basic queues* allow applications to store messages (e.g. reflecting intermediate results) without sending them to external systems. As a result, messages received from remote communication endpoints and internal state representation are handled in a uniform manner, simplifying application development.

3.2 Slicing the Message History

In Demaq, all application state is encoded in the message history. Rules access the state by posing queries against the history. Of course, processing queries against all existing messages for every processing step is inefficient, and the need to filter the relevant messages for every rule may lead to repetitive code. In addition, keeping the complete message history forever requires unbounded storage capacity. For these reasons, the Demaq language provides mechanisms to declare portions of the message history that are relevant in particular contexts, called *slicings*.

A slicing defines a family of *slices*, where each slice consists of all the messages with the same value for a particular part of the message (*slice key*). To identify the part of a message that should be used as the slice key - and thus as the basis for the partitioning - we rely on a subset of our XQuery-based rule language. Particularly, XPath [6] expressions are used to identify the part of a message that should be used as the slice key.

A slicing is created by specifying a unique name and the *slicing property*. The property definition lists a number of queues on which the slicing is defined. Messages from these queues are partitioned into *slices* according to their property value. All messages that share the same value of the slicing property become part of the same slice. Apart from providing convenient access to the relevant parts of the message history, slicings are also used to specify message retention policies.

Relevant Slice Suffix. Slicings declare which parts of the message history are relevant to the application and thus allow to identify obsolete messages which could be safely removed from the system to reclaim storage space. A simple, value-based partitioning is not enough, however, as it would still require to retain the complete, unbounded message history. Instead, we need a mechanism to specify which messages reflect the relevant application state, and which messages have become irrelevant and may thus be dropped.

To avoid unbounded buffering of message streams, *windows* have been proposed to specify relevant sub-streams [1] based on their position in a stream. The boundaries of such windows are based on the window size or relative to some landmark object in the stream, and the application developer must translate the message retention needs into window specifications.

In Demaq, we allow application developers to directly specify a condition that must be met by the messages that are sufficient to represent the current application state. Access to the slice then yields the smallest suffix which contains such a set of relevant messages. This very powerful semantics allows for a very intuitive, direct expression of relevance conditions on the message history.

To express message retention conditions, our language incorporates an additional construct. This `require` expression is an arbitrary XQuery expression of type boolean. Among all the contiguous sets of candidate messages in the slice that fulfill this condition, the most recent set is considered the currently relevant state of the slice. This set, and any messages more recent than that, are visible to the application. The `require` expression may refer to the candidate set of relevant messages using a special function (`qs:history()`). However, it may not refer to any other messages in the system. The reasons for this latter constraint are simple, efficient evaluation and garbage collection.

Since the `require` expression may not refer to other parts of the system state, and it always includes a complete suffix of the message history, and the fact that we chose the most recent qualifying set, we can guarantee a monotonous behavior of our relevant slice state: We can divide the slice into two parts, a relevant suffix (marked gray in Figure 2) and an irrelevant prefix. Our semantics guarantees that once a message belongs to the irrelevant prefix of a slice, it will never become relevant again. In other words, the boundary between relevant and irrelevant messages only moves toward more recent messages. Thus, it can be represented and tested using a simple message identifier or timestamp comparisons. This allows a simple, decoupled garbage collection strategy: If a message is no longer visible to any application rule because it is not part of any relevant suffix, it can be safely pruned from the message history. Consequentially, storage capacity can be reclaimed in a separate garbage collection process that never conflicts with rule evaluation.

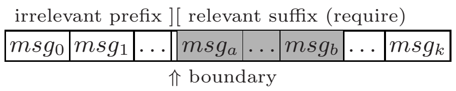

Fig. 2. Message history in a slice of size k+1

3.3 Application Rules

For our application rules, there is a significant overlap with the capabilities of existing, declarative XML query languages, in particular with XQuery [7]. XQuery allows for querying (sequences of) XML documents and document construction, supports XML data types, schema validation, etc.

Building on an existing language, such as XQuery, provides significant advantages. Developers can benefit from previous experience and reuse programming tools and development infrastructure. Additionally, existing query processing techniques can be potentially adapted to our application language. In the following sections, we discuss how the features required for building message-driven applications can be integrated into XQuery.

Assigning Application Rules. Every application rule is assigned to a single queue or slicing. Whenever a message gets inserted into this queue or slicing, the rule is evaluated with this message as the context item. In order to allow application developers to perform this association of rules to queues and slicings, we extend XQuery by incorporating an additional rule definition expression. Rule definition expressions can be used to give a unique name to an XQuery expression and assign it to a particular queue or slicing.

Optionally, an *error queue* can be defined for an application rule. Whenever a runtime error is encountered during the execution of this rule, a corresponding notification message is sent to this queue. Thus, this error handling mechanism allows other rules to handle this error. If no error queue is defined, error notifications are inserted into a system-provided, default error queue.

Enqueuing Messages. Every application rule describes how to react to a message by creating new messages and enqueuing them into local or gateway queues. While XQuery allows for the creation of arbitrary XML fragments, it does not incorporate any primitives for performing side effects. In our model, this is a severe restriction, as there is no possibility to modify the content of the queues underlying our application rules.

We adopt the extensions proposed by the XQuery Update Facility [10] to perform side effects on the message store. Every application rule is an updating expression that produces a (possibly empty) list of messages that have to be incorporated into the message store by enqueuing them to corresponding queues. In order to allow application programs to both specify the XML fragment to be enqueued as well as their target queue, we extend the XQuery Update Facility with an additional `enqueue message` update primitive.

Message Access. While XQuery incorporates powerful features to query (sequences of) XML documents, it does not provide operations for accessing the content of structures such as queues and slices. These read-only access operations can be easily provided by the runtime system in the form of external functions, particularly without requiring changes to the syntax or semantics of XQuery. In application rules, they are used to access the sequence of XML messages stored in a queue or slice.

Example. The example below illustrates the key building blocks of our declarative application language. First, we define two slicings that can be used to retrieve the master data and shopping cart items for a particular customer or transaction. These slicings are defined on the `customers` and `shoppingCart` queues and use XPath expressions to locate the customer/transaction ID in the respective messages. The `require` expressions are used to restrict the part of the message history that is returned by these slicings. For the customer master data, preserving the most recent item of the message history is sufficient, while the `require` expression `fn:false` for the cart items indicates that these items should never be removed from the message history.

In line 7, a new rule `handleCheckout` is defined for the `incomingMessages` queue. The corresponding rule body (lines 8 to 21) is evaluated every time a message is inserted into this queue. It mainly consists of XQuery expressions that are used to analyze the content of the incoming message and derive a result message. In lines 13 and 15, the slice access function `qs:slice` is used to access all relevant state information from the message history. Finally, after the result message has been created, the `enqueue message` expression is used to enqueue it to the `outgoingMessages` queue (line 20).

In this example, we omit the queue definitions expressions for the queues involved (e.g. `incomingMessages`) for reasons of brevity.

```
 1 create slicing property masterDataForCustomer
 2   queue customers value //customer/customerID require fn:count(qs:history()) eq 1;
 3
 4 create slicing property cartItemsForTransaction
 5   queue shoppingCart value /cartItem/transaction/ID require fn:false();
 6
 7 create rule handleCheckout for incomingMessages let $request :=
 8 //checkout return
 9   if($request) then
10     let $transactionID := $request/transactionID/text()
11     let $customerID := $request/customerID/text()
12     let $customerData := qs:slice($customerID, ”masterDataForCustomer”)[last()]
13     let $customerOrders := qs:slice($transactionID, ”cartItemsForTransaction”)
14     let $result :=
15       <result>
16         <orderedItems>{$customerOrders//item}</orderedItems>
17         <delivery>{$customerData//address}</delivery>
18       </result>
19     return enqueue message $result into outgoingMessages
20   else();
```

4 Execution Model

The Demaq language provides simple, yet expressive primitives to describe desired reactions to messages in terms of the message history. The use of a declarative language for rule bodies allows data independence and efficient execution using a query optimizer.

Our objective is to create an elegant way to completely specify stateful messaging applications, and not only to monitor or analyze message streams - we not only want to read state, but to modify it. Hence, a crucial aspect of the Demaq design is to define how state can be managed in a reliable way and - at the same time - allow for an efficient and scalable application execution.

The design issues in this context revolve around the transactional coupling of rule execution to the message store. It turns out that modeling both requests and state information as messages yields novel opportunities to improve execution performance. A major reason for this is the append-only strategy for the message history: We never perform in-place updates. As a consequence, there is much less need to synchronize concurrent execution threads, and there are fewer ways how a concurrent modification of the system state can cause conflicts.

The Demaq execution model captures the typical behavior of message-driven applications in a few simple rules and guarantees which are observed by the runtime system. It precisely determines Demaq rule semantics, and at the same time leaves enough freedom for an actual implementation to optimize runtime performance, as we will see in Sec. 5.

Core Processing Loop. The fundamental behavior of messaging applications can be described as a simple loop that (1) decides which message(s) to process next, (2) determines the reaction to that message based on the message contents and application state, and (3) effects the reaction by creating new messages. Actual implementations of the Demaq model may use any form of processing loop(s) that obeys the following constraints:

1. Each message is processed exactly once. This means that the evaluation of all rules defined for the message's queue and slicings are triggered once for every message.
2. Rules are evaluated by determining the result of the rule body as defined by XQuery (update) semantics, extended by the built-in function definitions described in Sec. 3.3. The result is a sequence of pending update operations in the form of messages to enqueue.
3. The overall result of rule evaluation for a message is the concatenation of the pending actions of the individual rules in some non-deterministic order.
4. Processing the pending actions for a message is atomic, i.e. after a successful rule evaluation all result messages are added to the message history in one atomic transaction, which also marks the trigger message as processed.
5. All rule evaluations for the same trigger message see the same snapshot of the message history, which contains all messages enqueued prior to the trigger message, but none of the messages enqueued later. This is motivated by the interpretation of each rule as an isolated statement of fact about the system behavior – if a certain situation arises, a certain action will eventually happen, no matter what other rules are defined for the same situation.

The above list includes strong transactional guarantees necessary to implement reliable state-dependent applications, but still allows many alternative strategies to couple message processing to a transactional message store.

Note that the above model does not allow for message store transactions that span rules. However, application developers do have some control over the amount of decoupled, asynchronous execution: The expressive power of XQuery allows the bundling of complex processing steps into single rules, which are executed in a single transaction and hence allow to constrain the visibility of intermediate results to concurrent transactions. Further, application developers can isolate intermediate messages in local queues that are not accessed by conflicting control paths.

Enqueue-Time Snapshot Isolation. To achieve a maximum of concurrency, our message store uses a variant of snapshot isolation [5] that is made possible by our unified message-based view of state and requests. In general, snapshot isolation freezes the system state visible to a transaction by creating a private version of the state. This avoids locking and improves concurrency, but requires the retention of old state versions and conflict resolution policies.

In our programming model, guaranteeing snapshot isolation is very cheap, because the management of old state versions is trivial. As there are no in-place updates, we can access old versions of the system state by just ignoring newer messages. We simplify this by using the enqueue-time of the trigger message as begin-of-transaction (snapshot) time for our rule evaluation . Hence, rule evaluation can see only all messages enqueued prior to the trigger message. Concurrent rule evaluation transactions do not need to lock parts of the history because updates by definition do not affect their visible part of the message history. We only need to synchronize the message writing transactions to guarantee atomic insertion of result messages. Note that deadlock handling is simple, as the complete set of updates is known before the first update needs to be performed. This strategy tremendously improves the concurrency and scalability of our application engine because very few short-term locks are required for synchronization.

5 System

In this section, we outline the architecture and implementation of the Demaq system that implements our programming model.

When deploying a Demaq application, a rule compiler is used to transform the application specification into execution plans for the runtime system. The runtime system consists of three major components. A transactional XML queue store provides an efficient and reliable message storage. Remote messaging and transport protocol aspects are handled by the communication system. The rule execution engine executes the plans generated by the rule compiler.

Rule Compiler. The purpose of the rule compiler is to transform applications into optimized execution plans for the runtime system. For this compiler, optimization opportunities exist on several levels of an application.

- **Rule-set rewriting.** We can change the overall structure of an application by modifying its set of rules. For example, the compiler merges all rules defined on the same queue into a single, combined rule. This simplifies factorizing common subexpressions across rules and saves the runtime system from invoking the rule execution component multiple times for a single message.
- **Rule-body rewriting.** A significant part of our programming language consists of the XQuery Update Facility, and many optimization techniques developed for XQuery can also be applied to our application rules. To profit from these techniques without reimplementing all of them, the compiler can split rule bodies into two parts, one processed by the Demaq rule execution engine, and one processed by the message store. In case of XQuery-enabled message stores (as is the case in our current implementation), the store-processed part is simply rewritten into an XQuery expression without Demaq-specific constructs, which can then be optimized by the store's XQuery compiler.
- **Physical optimization.** Platform-specific rewrites may be performed to speed up application processing. For example, our runtime system incorporates a special operation which works similar to a link in Unix file systems. This operation avoids a full copy of the message when messages are forwarded unchanged from one queue to another. Another example for a runtime-specific optimization is template folding [19].

Rule Execution Engine. The rule execution engine implements our execution model described in Sec. 4. It decides when and in which order messages from the queue store should be processed, how to incorporate updates, and when to send or receive messages from the communication system. The main challenge in the design of a rule execution engine is to exploit the degrees of freedom of the execution model to provide high processing performance and message throughput. For this purpose, our system relies on multiple concurrent execution threads. Another issue is to find suitable garbage collection strategies to achieve an optimal balance between reclaiming storage capacity and runtime overhead.

Transactional XML Queue Store. Our message store is built on the foundation of a native XML base management system. Our current implementation alternatively uses Natix [14], a research prototype of a native XML data store, or IBM DB/2 Version 9. While XML-enabled database systems provide efficient and reliable XML document processing facilities, they usually manage XML repositories as collections, which are unordered sets of XML documents. As our programming model requires queue-based message storage, a main challenge was to replace the existing, collection-based data handling and recovery facilities to support a queue-based management model. We incorporated these changes into the Natix system. They allow us to use the efficient XML storage facilities as well as the sophisticated recovery and schema management features of Natix for our message queues. For DB/2, we simulate queue-based storage using auxiliary tables. Another challenge is to provide efficient access to the message history.

For this purpose, we rely on the persistent B-Tree indexes provided by the Natix system which we use to implement slicings. Additionally, Natix incorporates our specialized version of Snapshot Isolation (see Sec. 4).

Communication System. The communication system provides all remote communication facilities. It implements both asynchronous and synchronous transfer protocols (such as HTTP and SMTP), and thus allows applications to interact with various types of external communication endpoints.

6 Experiments

In this chapter, we provide a brief experimental evaluation of our programming model. For this purpose, we use the Demaq runtime system introduced in the last section to execute an exemplary online shopping application. The complete source code of this application and additional application examples - such as a Demaq implementation of the TPC-App Application Server benchmark - are available at our project website http://www.demaq.net/.

We also implemented the application as a BPEL process and executed it on a commercial, enterprise-class application server with a relational database backend. Unfortunately, licensing restrictions do not allow to disclose details about the system, let alone vendor name and software version. However, we believe that the results help to evaluate the performance of our implementation in the light of one of the most advanced application servers available today.

Setup. All measurements were performed on a server equipped with an AMD Athlon 64 X2 Processor 4600, 2 GB of main memory, running Opensuse Linux 10.3. This system was used to run the BPEL application server and our native runtime system with Natix as the underlying message store. An additional client computer was used to send messages via HTTP.

In order to get an impression of the runtime to expect during the following measurements, we first performed an exemplary run of our online shopping application, consisting of 24 messaging operations: The client connects to the server, adds both 10 books and music items (each reflected by a message of 2.5 KB) to the shopping carts, requests the total value of both music and book items, and finally performs a checkout operation. This run was repeated 100 times to reduce the effects of statistical outliers.

The application server required an average of 6.25 seconds in order to run the scenario. Using our native implementation, the same run took 2.18 seconds, which confirms that native XML data handling and avoiding a multi-tiered architecture can help to improve application runtime.

Performance Impact of Context Size. In order to investigate the impact of context size on the runtime performance, we subsequently add 10000 books (each 2.5 KB in size) to the shopping cart of a single application instance. While a single customer buying thousands of books is rather unlikely for our online shopping example, handling thousands of messages in a single application context is no uncommon scenario in other application domains [4].

Figure 3 visualizes the round-trip-time (in seconds) for each request. With growing instance size, the response times of the application server deteriorate. This effect might be caused by performing an in-place update on the corresponding data structure of the runtime context and writing it back to the database back-end. In our runtime system, every additional book can be appended to a queue of the system, thus leaving the response time virtually unaffected.

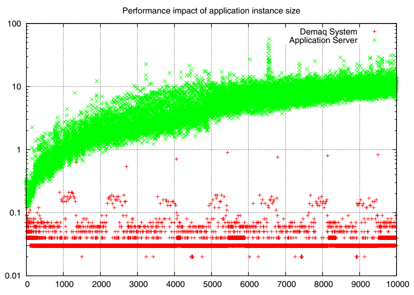

Fig. 3. Effect of context size

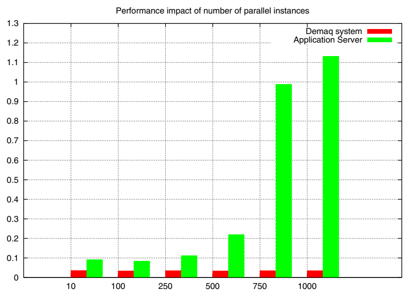

Fig. 4. Effect of active instances

Parallel Application Instances. In this experiment, we investigate the impact of multiple concurrent, active instances, each of them storing 100 book orders (250 KB) and 10 music orders (25 KB). We analyze how the response time of the systems change with an increasing number of parallel instances. For this purpose, our client sequentially requests the server to calculate the overall price of the music order items for each instance.

Figure 4 depicts the average response time (in seconds) for answering a client request. An increasing number of active instances has a considerable impact on the response times of the application server performing expensive instance management operations. For our runtime system, there are no instances that need to be managed. Instead, all messages that belong to a particular context are retrieved by querying the message store. Thus, an increasing number of parallel instances does not interfere with the response time.

7 Related Work

Application Servers Today, distributed applications are usually executed by multi-tier application servers [2]. For XML messaging applications, these tiers typically consist of queue-based communication facilities (e.g. [16,18]), a runtime component executing the application logic, and a database management system that provides persistent state storage. An additional transaction processing monitor ensures that transactional semantics are preserved across tiers.

Application servers allow for the convenient deployment of applications in distributed and heterogeneous environments. However, their use entails several problems which are discussed in literature. These problems include significant functional overlap and redundancy between the different tiers [17,20], and complex and brittle configuration and customization caused by multi-layer, multi-vendor environments [2]. Further, frequent representation changes between data formats (XML, format of the runtime component, relational database management system) decrease the overall performance [15].

Data Stream Management Systems Data stream management systems (DSMS) and languages (e.g. [1,12]) are targeted at analyzing, filtering and aggregating items from a stream of input events, again producing a stream of result items. Several stream management systems rely on declarative programming languages to describe patterns of interest in an event or message stream. In most cases, these languages extend SQL with primitives such as window specification, pattern matching, or stream-to-relation transformation [3].

In contrast to application servers that provide reliable and transactional data processing, stream management systems aim at low latency and high data throughput. To achieve these goals, data processing is mainly performed in main memory (e.g. based on automata [12] or operators [1]). Thus, in case of application failures or system crashes, no state recovery may be performed, and data can be lost.

XML Query and Programming Languages For an XML message processing system, choosing a native XML query language such as XPath [6] or XQuery [7] as a foundation for a programming language seems to be a natural choice. However, these query languages lack the capability to express application logic that is based on the process state - they are functional query languages with (nearly) no side effects. There are various approaches [9,11,15] to evolve XQuery into a general-purpose programming language that can be used without an additional host programming language.

8 Conclusion

We investigate a new programming model for distributed applications based on XML messaging. In our system, a declarative language that directly operates on messages and queues describes the processing logic in terms of message-driven rules. Application state is modeled exclusively using the message history. By treating application instance management as a data management problem best addressed by a data management server, we get a fresh perspective on how to optimize the architecture of application servers. A result is improved scalability of our execution engine to large numbers of concurrent application instances: In particular, we can avoid loading and saving the complete application state from a database for every processing step, which tends to take up a large fraction of conventional application servers' processing resources. A brief performance evaluation of our runtime system confirms the potential of the proposed approach. It also illustrates the practical benefits of treating process instances as data in terms of scalability and performance.

References

1. Abadi, D.J., Carney, D., Çetintemel, U., et al.: Aurora: a new model and architecture for data stream management. VLDB J. 12(2), 120–139 (2003)
2. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applictions. Springer, Heidelberg (2004)
3. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J. 15(2), 121–142 (2006)
4. Arasu, A., Cherniack, M., Galvez, E.F., et al.: Linear Road: A stream data management benchmark. In: VLDB, pp. 480–491 (2004)
5. Berenson, H., Bernstein, P.A., Gray, J., et al.: A critique of ANSI SQL isolation levels. In: SIGMOD Conference, pp. 1–10 (1995)
6. Berglund, A., Boag, S., Chamberlin, D., et al.: XML path language (XPath) 2.0. Technical report, W3C (January 2007)
7. Boag, S., Chamberlin, D., Fernández, M.F., et al.: XQuery 1.0: An XML query language. Technical report, W3C (January 2007)
8. Böhm, A., Kanne, C.-C., Moerkotte, G.: Demaq: A foundation for declarative XML message processing. In: CIDR, pp. 33–43 (2007)
9. Bonifati, A., Ceri, S., Paraboschi, S.: Pushing reactive services to XML repositories using active rules. Computer Networks 39(5), 645–660 (2002)
10. Chamberlin, D., Florescu, D., Melton, J., et al.: XQuery Update Facility 1.0. Technical report, W3C (August 2007)
11. Chamberlin, D.D., Carey, M.J., Florescu, D., et al.: Programming with XQuery. In: XIME-P (2006)
12. Demers, A.J., Gehrke, J., Panda, B., et al.: Cayuga: A general purpose event monitoring system. In: CIDR, pp. 412–422 (2007)
13. Fernández, M.F., Malhotra, A., Marsh, J., et al.: XQuery 1.0 and XPath 2.0 data model (XDM). Technical report, W3C (January 2007)
14. Fiebig, T., Helmer, S., Kanne, C.-C., et al.: Anatomy of a Native XML base management system. VLDB Journal 11(4), 292–314 (2003)
15. Florescu, D., Grünhagen, A., Kossmann, D.: XL: a platform for Web Services. In: CIDR (2003)
16. Foch, C.B.: Oracle streams advanced queuing user's guide and reference, 10g release 2 (10.2) (2005)
17. Gray, J.: Thesis: Queues are databases. In: Proceedings 7th High Performance Transaction Processing Workshop, Asilomar CA (1995)
18. IBM. WebSphere MQ (2009), `http://www.ibm.com/software/integration/wmq/`
19. Kanne, C.-C., Moerkotte, G.: Template folding for XPath. In: Third International Workshop on XQuery Implementation, Experience and Perspectives (2006)
20. Stonebraker, M.: Too much middleware. SIGMOD Record 31(1), 97–106 (2002)