

# Declarative Development of Distributed Applications

Alexander Böhm

alex@db.informatik.uni-mannheim.de

Department of Mathematics and Computer Science, University of Mannheim, Germany

## ABSTRACT

We present a novel approach for the convenient implementation and efficient execution of distributed XML message processing applications. Various application classes such as Web Services are based on asynchronously exchanging XML data. However, today's programming languages and execution systems fail to support their particular demands such as integrated XML type support and efficient, asynchronous messaging. As a result, the development of distributed applications is unnecessarily complex and their runtime performance deteriorates. To overcome these deficits, we propose a fully declarative XML processing language that allows for the convenient specification of an individual node participating in a distributed application. It is complemented by a corresponding runtime system for reliable and efficient application execution. Within this paper, we discuss the details of our approach, the results achieved so far and open research questions.

## 1. INTRODUCTION

Apart from traditional usage scenarios such as online shopping and browsing, the web continues to evolve to an active platform for distributed applications, e.g. implementing business processes. Standardized protocols and technologies including Web Services [2], RSS/Atom feeds, REST [12] and AJAX [14] provide the communication channels between the involved systems. They allow the integration of heterogeneous components and architectures, thus creating applications which exclusively communicate using XML messages and asynchronous transfer modes.

Today, the individual nodes participating in distributed applications are realized using imperative languages such as Java or C# and deployed on traditional, n-tier architectures [2]. They involve a multitude of hard- and software layers, including messaging solutions for remote communication, application servers running object-oriented programs, (relational) database backends providing persistent storage, and transaction monitors enforcing transactional semantics

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

wherever necessary.

To support distributed applications and offer communication mechanisms such as SOAP and AJAX, these systems usually incorporate additional XML adapters and converters. However, they fail to efficiently support the requirements of distributed, XML-messaging applications. These include convenient, high-performance XML processing facilities and reliable, asynchronous messaging operations. As a consequence, development gets unnecessarily complex:

- Users have to rely on complex interfaces to perform even the most basic message processing tasks.
- Significant marshalling effort is required to convert XML data to the type system of the programming language whenever exchanging data with remote peers.
- The immense number of hard- and software layers leads to a considerable communication overhead between the involved components.

Our goal is to investigate how database technology can be applied and extended to overcome those problems and restrictions. We want to

1. Find a *declarative* alternative to the existing solutions for creating distributed applications.
2. Develop a high-performance runtime system that avoids the protocol overhead and impedance mismatch that plague today's architectures.

Our approach is motivated by the observation that the success of database systems is a consequence of using declarative (query) languages. In contrast to imperative languages that expect developers to provide a detailed specification of the application execution plan (including optimizations), declarative languages allow to specify the desired outcome instead of the required algorithmic steps. This increases developer productivity by means of powerful constructs. At the same time, declarative languages provide the corresponding execution system with a higher degree of freedom, as they allow choosing from different evaluation strategies, i.e. using indexes and cost-based selection of algorithms.

We follow a practical methodology: We design an entirely declarative, XML-aware application language and a corresponding, high-performance execution engine, and implement them. To evaluate the benefits of our language, we plan to compare the development effort for creating applications using existing, imperative languages with that of using our novel, declarative approach.

Given the assumption that we can find a design for an efficient runtime system, we want to perform performance

comparisons with existing systems to evaluate the benefits of our approach. For this purpose, we are implementing a runtime system prototype. It's our goal to exploit the optimization potential induced by the declarative application specification, and - by using database technology - thus to allow for a more efficient application execution and optimization.

**Structure of the Paper.** In Sect. 2, we give an overview of the key characteristics and concepts of our approach consisting of a declarative application language and a corresponding runtime system. In the next section, we discuss the current status of the language (Sect. 3.1) and the execution engine (Sect. 3.2) in greater detail. Sect. 4 briefly reviews the available solutions for asynchronous XML message processing and gives an overview of related work. Section 5 concludes the paper and gives an outlook on future work and open issues.

## 2. MODEL

We believe that a very elegant model for distributed, message-based applications is based on sets of XML message queues and declarative rules that govern inter-queue message flow.

Most of today's message-oriented communication solutions, including message-oriented middleware [15] and messaging systems [2], successfully employ queues to provide highly efficient, asynchronous processing facilities. Hence, the fundamental concept for message storage in our model is the message queue. Further, scalable data processing in database systems relies on declarative query languages and supporting runtime systems. In the following, we will explain how, in our model, these successful concepts are combined to yield a scalable, easy-to-use XML messaging system architecture.

### 2.1 Logical Model

On the logical layer, our approach relies on a declarative, rule-based application language (Sect. 2.1.1) and a message retention policy (Sect. 2.1.2) that allows us to determine application state from the message history.

#### 2.1.1 Declarative Application Language

Our approach incorporates a declarative, XML-aware programming language that allows for the convenient specification of distributed applications. It directly operates on XML messages (and queues as their storage containers), thus avoiding the impedance mismatch that plagues most of today's systems.

Using this language, the application logic is specified as a set of declarative rules, describing the message flow between both local and remote message queues. Rule execution is performed whenever a new message gets inserted into a queue. Thus, it might either be caused by a remote system sending a request, or as a consequence of forwarding data locally from one queue to another.

Our goal is to control all aspects of applications using declarative facilities. Apart from specifying the application logic using declarative rules (as described above), this includes remote communication, the definition of data retention criteria and error handling facilities. This leads to a number of challenges.

**Research Questions.** How can existing, declarative XML query languages such as XPath and XQuery be incorporated in a rule language? How to allow for side effects (e.g. creat-

ing and sending messages) while keeping our language fully compositional? How do we designate transactional borders in the language? How much data independence can we guarantee, hiding details about message parsing, storage, persistence, and transactions?

#### 2.1.2 Message History

The XML messages exchanged over the network record both the observable state of external entities and the individual steps taken by an application. The history of messages is a useful resource for application debugging, reasoning about system behavior, keeping audit trails, and conflict handling. In fact, an important observation is that the state of any instance of a distributed, XML-messaging based application can be determined from the XML messages sent to and received from remote communication partners. Particularly, determining the next steps to be executed can be viewed as a *query against the message history*.

Our approach is to retain all messages sent and received. Message queues are not only used in their traditional way, providing asynchronous communication gateways, but also as storage containers keeping state information for both local and external processes. As application state can be determined by querying the message history, we can refrain from explicitly materializing application context and using local variables. Thus, our approach allows for large numbers of concurrent application instances without requiring complex and expensive context management operations (e.g. hydration/dehydration in [6]).

**Research Questions.** (Almost) any system will eventually run out of message storage capacity. As a result, there have to be some retention criteria, indicating whether a particular message is still required, or whether it can be safely deleted to reclaim disk space. An interesting question in this context is whether these retention criteria can be derived from application programs or, alternatively, can be *declaratively* specified. How can we aid developers by providing language primitives (such as history views, or virtual queues) that allow to directly access relevant messages instead of filtering the whole message history by hand?

### 2.2 Physical Model

We implement the logical concepts described above by a queue-based, reliable XML message store and a runtime system. These two components are discussed in the following sections.

#### 2.2.1 Efficient Application Execution

The declarative language introduced above is complemented by a corresponding runtime system. Our goal is to exploit database technology to allow the efficient execution of application rules. Despite imperative programs that usually provide a detailed algorithmic specification of the steps to be executed, our declarative rules provide the runtime system with a greater degree of freedom. This principle is one of the main reasons why declarative query languages are so successful. We want to apply the existing knowledge from database systems to message processing.

Our idea is to compile application rules into query execution plans that are evaluated against the message store. Thus, the problem of efficiently executing distributed applications is transformed into one of database query processing, where every application rule is a query against both

an incoming message and the system state reflected by the messages in the queue store.

This approach leads to a new scenario for query optimization, different from both stream processing systems [4] where queries mainly target incoming messages, and traditional databases where query execution is performed against a persistent store.

**Research Questions.** Are there any heuristics for rewriting application rules or entire rule sets into more efficient ones, e.g. by combining several rules into a conjoint execution plan? Can we come up with a cost model for optimizations? Where do existing query optimization techniques fail in our scenario and can they potentially be adapted?

### 2.2.2 Queue-Based, Reliable XML Storage

Realization of the message processing primitives of our language requires efficient, reliable XML message queue storage that is also suitable for query processing against the message history.

Reliable and efficient storage of large amounts of XML data is the domain of XML data stores (XDS). To benefit both from the advantages of message queues and native XML data stores, we rely on a queue-enabled XDS to handle all XML data fragments. This approach avoids having redundant functionality in two different system components [17] and saves us from crossing system borders when forwarding data between the database and the message broker. However, some questions arise:

**Research Questions.** How can XML queue data structures be integrated into the database kernel while preserving both high-performance queue and database operations? Can we find index structures that allow retrieving messages from our queue-based storage without searching the whole message history? How can these indexes be efficiently maintained in the presence of queue-based access patterns?

## 3. CURRENT STATUS

Within this section, we discuss the current status of our research and the results achieved so far. We start with a description of our declarative application language in Sect. 3.1 before giving a brief overview of our runtime system in Sect. 3.2.

### 3.1 Application Language

Our approach relies on a declarative application language with integrated support for XML data and asynchronous messaging operations. Using this language, applications can directly interact with XML messages and their queue storage containers - without any impedance mismatch or auxiliary transformation layers. In the following sections, we briefly discuss its key characteristics, a more comprehensive discussion, including several additional examples, can be found in [7].

When developing a new application, remote transport endpoints and local data structures for message storage have to be defined in a first step. This can be done using the simple definition language that will be introduced in the following section. Based on the foundation of these queue definitions, we rely on a declarative, rule-based language to describe the application logic (Sect. 3.1.2). The execution of these application rules is governed by the processing model described in Sect. 3.1.3. To facilitate application development, virtual

queues, called *slices*, can be used to declaratively specify groups of logically related messages (Sect. 3.1.4).

#### 3.1.1 Queue Definition Language (QDL)

The Queue Definition Language (QDL) is used to create the infrastructure of message queues mentioned above. QDL incorporates primitives to both set up local queue data structures and allows the specification of gateway queues acting as interfaces for network communication.

*Local queues* provide applications with queue-based storage containers. Various kinds of local queues are supported, including transactional, persistent queues as well as transient, main-memory queues. These transient queues allow for highly efficient message handling for those parts of an application that do not require transactional guarantees and persistence. Local queues can, for example, be used to capture (intermediate) information, e.g. to keep track of all incoming customer orders.

*Gateway queues* provide application programs with in- and outbound communication facilities, both allowing them to send messages to remote transport endpoints and to receive incoming data. They offer a convenient, queue-based interface to the services of the communication subsystem. By using this approach, sending a message to a remote peer (such as a Web Service) can be easily done by enqueueing it to the corresponding gateway queue.

#### 3.1.2 Queue Rule Language (QRL)

The Queue Definition Language (QDL) discussed above is complemented by the Queue Rule Language (QRL), which is used to declaratively specify the application logic. The idea of the QRL is to describe application programs as a set of rules governing the flow of messages between the QDL-defined message queues.

As an instance of event-condition-action (ECA) rules [19], each QRL rule consists of an event statement defining when it should be evaluated and a combined condition-action part specifying the application logic to be executed if the corresponding event occurs. The event statement associates each rule with exactly one queue of the system. Whenever a message is inserted into this particular queue, the application logic defined in the condition-action part is evaluated. Eventually, this results in the creation of new messages, being inserted into either local or gateway queues, thus either becoming the input for other rules or sending messages to remote transport endpoints.

To perform these XML processing and messaging tasks in a declarative way and without any impedance mismatch, QDL builds on the XQuery Update Facility [10] to express the application logic in the condition-action parts. Relying on the XQuery Update Facility has several advantages. It allows us to provide application developers with the powerful, declarative XML processing capabilities of XQuery (e.g. to extract data from incoming messages and construct new XML fragments) and to perform update operations, such as adding newly constructed fragments to messages queues. In order to achieve a seamless integration and convenient interaction of XQuery expressions with our queue-based processing model, we extend the XQuery Update Facility with additional, system-provided functions. These functions can e.g. be used to access individual messages in the system or to retrieve all messages stored in a particular message queue. By incorporating an additional `enqueue` update primitive,

XML fragments can be easily enqueued to a particular destination queue. As the overall number of additions is comprehensible, we expect developers who are already familiar with XQuery to easily understand and adopt our programming language.

```

1 create rule findPremiumRequests for customerRequests
2 let $orders := qs:queue("orders")
3 let $lastOrders := $orders[//custID eq qs:message()//custID]
4 let $orderCount := fn:count($lastOrders)
5 return
6   if ($orderCount ge 10)
7   then
8     let $newMessage :=
9       <premiumCustomer>
10        <orderCount>{$orderCount}</orderCount>
11        {/order}
12        </premiumCustome>
13     return do enqueue $newMessage into premiumRequests
14   else
15     do enqueue . into regularRequests;

```

Figure 1: Example application rule

Figure 1 gives a brief example of a QRL expression. This particular rule used in a shop application checks whether an incoming order originates from a premium customer (someone with more than ten previous orders). The event statement (line 1) creates a new rule named `findPremiumRequests` for a queue called `customerRequests`. The `qs:queue` function in line 2 allows the rule to retrieve all messages currently stored in the `orders` queue. The following `let` statement retrieves all orders that have the same customerID as the current message (retrieved using the `qs:message` function). Finally, if the amount of orders exceeds ten, a new message is constructed and forwarded to a message queue handling the requests of premium customers. Otherwise, the current message (which is also the context item for the rule) is simply forwarded to the `regularRequests` queue for further processing.

Unfortunately, due to space constraints, we cannot give additional examples of QRL rules. We refer the reader to [7] for additional examples and a more comprehensive description of the QRL application language.

### 3.1.3 Processing Model

The execution of our declarative application rules is governed by a processing model specifying when and how the rule-based application logic should be evaluated. Following the terminology proposed by Paton and Diaz [19], we rely on a detached coupling mode that separates processing a message from its insertion into the queue store. As a result, there may be a number of yet unprocessed messages in every queue of the system. For each message, the execution engine evaluates all QRL application rules with the root of the current message as context item. Like the "snapshot semantics" approach used by the XQuery Update Facility [10], a list of resulting update operations is collected. These update operations are not performed until all rules have been evaluated. Thus, the results of one particular rule cannot be seen by another rule being evaluated on the same message. These snapshot semantics allow us to provide an iterative cycle policy that avoid recursive rule invocation and thus facilitates application debugging.

As discussed above, our approach relies on determining the state of an application instance by querying its message history. Therefore, all messages sent to and received from remote systems need to be preserved. The rule execution

```

create slicing property shippedOrders
queue orders value /request/customerID
queue shippings value /confirmation/custID;

```

Figure 2: Example slicing definition statement

engine guarantees this by performing a *non-consuming* dequeue when retrieving messages from the queue-based data store. While the message is marked as processed to avoid considering it multiple times, it is not removed from the message store.

### 3.1.4 Virtual Queues

Queues provide an efficient way of physically organizing related messages, e.g. by keeping all messages of a particular type, such as customer requests, in a conjoint queue. However, in complex applications, there are usually several, orthogonal access patterns for the underlying messages. For example, an application might need to operate on all high volume orders, access the messages sent from an individual customer, or need to retrieve all orders for a particular product. To provide a convenient access to these logical groups of messages, our approach allows the definition of virtual queues, called *slicings*.

Similar to the concept of parameterized views [20], these slicings can be used to create logical groups of related messages. In Figure 2, a slicing is used to combine the messages from the `orders` and the `shippings` queue. For each distinct customer identifier (contained in order messages as `customerID` element and as `custID` in shipping confirmations), a new virtual queue (called *slice*) is created, containing all messages which share the same result for the `value` expression in the slicing definition.

Slicings can be used in the same ways as physical queues. Particularly, rules can be defined on slicings, which causes their application logic to be evaluated whenever new messages are added to a particular slice. Thus, apart from defining operations based on the *physical* placement of messages in queues, developers can specify application rules on the basis of *logical* groups of related messages, independent of their type and storage location.

In addition to a more convenient specification of the application logic, we believe that slicings are a powerful concept and a promising foundation to declaratively control other application aspects. For example, we plan to investigate whether they can be used for the specification and integration of message retention criteria into applications.

## 3.2 Runtime System

To evaluate the practical feasibility of our approach and to allow performance comparisons with existing solutions, we are implementing the proposed runtime system in C++.

The foundation of our system is the native XML data store Natix [11], which provides a high-performance, transactional XML storage engine. To allow for efficient message storage and retrieval, we have integrated queues as first class data structures into the database kernel.

A rule execution engine is responsible to process the application rules. Whenever a new message is inserted into a queue (either from a remote peer or as a consequence of local rule execution), it is announced to a scheduler, which contains references to all pending messages. These pending messages are consumed by multiple, concurrent processing threads. Each thread evaluates the application rules with

System/Language	Declarative	XML	Messaging	Reliable	Scalability	Realtime
Our approach	+	+	+	+	+	-
XL [13]	0	+	+	-	-	-
BPEL [3]	0	+	+	-	-	-
Active Rules [8]	+	+	0	-	-	-
ActiveViews [1]	+	+	-	-	-	-
XChange [5]	+	+	+	-	-	-
XQueryP	0	+	-	+	-	-
Queue-enabled databases (e.g. [16, 22])	-	0	0	+	+	+
Message brokers [2]	-	0	+	+	+	+
Continuous query systems	+	0	+	0	+	+

+: Considered/supported, 0: partially considered/supported, -: Not considered/supported

**Figure 3: Classification of existing approaches**

respect to a particular message. As a result, new messages are created and either enqueued into local queues (e.g. for capturing intermediate state) or forwarded to the communication subsystem in order to send them to remote transport endpoints.

The communication subsystem connects the execution engine with remote peers such as Web Services or other distributed applications. It implements the necessary data exchange mechanisms such as SOAP, REST, RSS, or Atom. By providing application programs and the rule execution engine with queue-like interfaces, most of the details of network communication can be hidden from the developer. However, as remote communication is a potential source for various kinds of errors [21], the communication subsystem needs to propagate these errors to application programs. By representing these errors with XML messages that are automatically sent to associated "error queues", applications can easily react to errors by implementing corresponding rules.

#### 4. RELATED WORK

Before discussing the approaches proposed in the literature and related systems in the following section, we briefly summarize their main characteristics in Figure 3. The matrix indicates whether they provide a declarative application language, offer XML and messaging support, include reliable and transactional operations, consider scalability aspects and allow for realtime processing.

Today's application servers mainly rely on additional modules to support asynchronous XML data processing. These solutions either incorporate XML support into high-performance **message broker** solutions [2] or - in the spirit of the thesis that queues are databases [17] - directly integrate queuing facilities into XML-enabled **database servers** [16, 22]. Thus, they offer highly efficient and reliable communication facilities for interacting with remote services. By providing standardized interfaces such as the Java Message Service (JMS), they allow for the convenient integration of XML messaging functionality into existing architectures. However, the message consuming application programs are usually written in imperative languages such as Java or C#. These languages lack integrated support for XML data types. As a result, data has to be transformed into the type system of the host language. This impedance mismatch causes significant marshalling overhead for every single message which is sent and received.

The performance and convenience deficits of today's architectures resulted in novel approaches and languages to tackle the specific requirements of efficient XML data processing. One of the first designs of a native Web Service programming language is **XL** [13]. It extends XQuery with ad-

ditional constructs, e.g. for synchronous and asynchronous service invocations, update operations and statement combinators. Some of these extensions can also be found in the **XQueryP** [9] language, aiming at the simplification of application development with XQuery. XL programs are executed by a sophisticated runtime system, which employs various optimization techniques. A virtual machine governs the code execution and manages application data and instance metadata using an XML database.

Instead of providing a full-featured programming language for XML data, the main focus of the Business Process Execution Language for Web Services (**BPEL**) [3] is on the composition of existing Web Services. Built on standards such as SOAP and WSDL, BPEL's main application is the orchestration of Web Service invocations. The choice of XPath 1.0 as the underlying expression language leads to a restricted expressiveness of BPEL programs. This limitation is tackled by auxiliary proposals such as BPEL for Java technology (BPELJ), which allows Java fragments to be directly embedded into BPEL code.

Bonifati et al. [8] propose **Active Rules**, an event-condition-action (ECA) rule language for incorporating reactive functionality into an XML repository. The event part of their application rules tracks modifications of repository documents. In the conditional statement, XQuery is used to define both the predicate for action execution and variable bindings that can be used by the action part. The action part of the rules is used to send SOAP messages to external applications.

The **ActiveViews** system [1] is another example of using an ECA rule language for reactive applications. Here, events are not restricted to document changes, but could also be operations on instance variables, objects and remote calls. Again, an XML query language is used for the conditional part of the rules. The resulting actions comprise service invocations, operations on variables as well as notifications and logging.

The **XChange** programming language [5] uses ECA rules to create a reactive language for the semantic web. It treats the web as a distributed document repository and provides reactivity for both local and remote events, which are represented by explicit event notification messages. To support complex event processing, the event part of an XChange rule consists of an event query that operates on local events and event messages. The conditional part is expressed in an XML query language and accesses local and remote documents using a peer-to-peer communication model. The resulting action of a rule is either a single update operation or a message sent to a remote service.

*Discussion.* While these proposals contain interesting approaches to improve the development of distributed applica-

tions, they exhibit several deficits. First and foremost, none of the proposals provides an entirely declarative solution for XML message processing. For example, as BPEL's XPath 1.0 based expression language is too limited to specify complex applications, developers have to resort to embedding imperative Java code into BPEL applications. Another example for this approach is XL, which borrows constructs and concepts (e.g. for exception handling) from Java. These imperative constructs restrict the optimization potential of the development language. Other deficits of existing approaches include the lack of closure, as they e.g. allow users to react to local database changes by sending XML messages, but do not provide them with the possibility to handle incoming messages [8]). Reliability aspects such as transactional message storage and recovery - which are crucial in various application areas such financial services [18, 23] - are often not considered at all. Additionally, most designs proposed in the literature do not consider application scalability. For example, both XL and the Oracle BPEL Process Manager [6], one of the most advanced BPEL engines, rely on (potentially large) runtime contexts for each application instance. These contexts require sophisticated memory replacement strategies, especially when implementing business processes involving long running transactions [6].

## 5. CONCLUSION

Within this paper, we discussed a novel, declarative approach for the development of distributed, asynchronous XML messaging applications. We gave a brief overview of the problems today's application servers and programming languages exhibit with respect to this new class of applications. While there are several interesting approaches to the convenient and efficient development and execution of distributed applications in the literature (Sect. 4), there is no solution that allows both for a declarative specification of the application logic and its efficient execution. Additionally, critical aspects such as reliable processing, transaction mapping and system scalability are often not considered at all.

Driven by the motivation to create a declarative and efficient solution for asynchronous, XML messaging applications, we introduced the concepts underlying our approach in Sect. 2. We proposed a fully declarative application language based on the XQuery Update Facility [10] that allows for the convenient specification of application programs without any impedance mismatch. Our application language is complemented by a runtime system with efficient and reliable processing facilities for asynchronous XML messaging based on a queue-enabled, native XML data store.

We discussed several open issues and research questions we plan to address in future work. Among others, these include providing transactional guarantees for groups of logically related messages, language extensions (e.g. for declaratively controlling message retention and handling time-based events), and rule set optimizations to speed up rule execution without changing the externally observable behavior of the implemented application.

*Acknowledgments.* We thank Simone Seeger for her helpful comments on the manuscript.

## 6. REFERENCES

- [1] S. Abiteboul et al. Active Views for electronic commerce. In *VLDB*, pages 138–149, 1999.
- [2] G. Alonso et al. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2004.
- [3] T. Andrews et al. Business process execution language for web services version 1.1. Technical report, 2003. <http://www.ibm.com/developerworks/library/ws-bpel>.
- [4] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.
- [5] J. Bailey et al. Flavours of XChange, a rule-based reactive language for the (semantic) web. In *RuleML*, pages 187–192, 2005.
- [6] S. Blanvalet. Managing a BPEL production environment. Technical report, Oracle Corporation, 2006. [http://www.oracle.com/technology/pub/articles/bpel\\_cookbook/blanvalet.html](http://www.oracle.com/technology/pub/articles/bpel_cookbook/blanvalet.html).
- [7] A. Böhm, C.-C. Kanne, and G. Moerkotte. Demaq: A foundation for declarative XML message processing. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007*, pages 33–43, 2007.
- [8] A. Bonifati, S. Ceri, and S. Paraboschi. Pushing reactive services to XML repositories using active rules. *Computer Networks*, 39(5):645–660, 2002.
- [9] D. Chamberlin et al. Programming with XQuery. In *XIME-P 2006*, 2006.
- [10] D. Chamberlin, D. Florescu, and J. Robie. XQuery update facility. Technical report, World Wide Web Consortium, July 2006. W3C Working Draft.
- [11] T. Fiebig et al. Anatomy of a Native XML base management system. *VLDB Journal*, 11(4):292–314, 2003.
- [12] R. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.
- [13] D. Florescu, A. Grünhagen, and D. Kossmann. XL: a platform for web services. In *CIDR 2003*, 2003.
- [14] J. James Garrett. Ajax: A new approach to web applications. Technical report, Adaptive Path, February 2005. <http://www.adaptivepath.com/publications/essays/archives/000385.php>.
- [15] D. Gawlick. Messaging/Queueing in Oracle 8. In *ICDE*, pages 66–68, 1998.
- [16] D. Gawlick and S. Mishra. Information sharing with the Oracle database. In *DEBS*, 2003.
- [17] J. Gray. Thesis: Queues are databases. In *HPTS*, 1995.
- [18] FIX Protocol Ltd. Financial information exchange (FIX) protocol. <http://www.fixprotocol.org/>, 2006.
- [19] N. W. Paton and O. Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999.
- [20] M. Toyama. Parameterized view definition and recursive relations. In *ICDE*, pages 707–712. IEEE Computer Society, 1986.
- [21] J. Waldo et al. A note on distributed computing. In *Mobile Object Systems*, pages 49–64, 1996.
- [22] R. Wolter. An introduction to SQL server service broker. Technical report, Microsoft, 2005.
- [23] O. Zimmermann et al. Second generation web services-oriented architecture in production in the finance industry. In *OOPSLA*, pages 283–289, 2004.