

Declarative Development of Distributed Applications

Alexander Böhm

alex@db.informatik.uni-mannheim.de

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

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, RSS/Atom feeds and REST, provide the communication infrastructure for 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 those distributed applications are realized using imperative languages, such as Java or C#, and deployed on traditional, n-tier architectures. They involve a multitude of hard- and software layers, including messaging solutions for remote communication, application servers running object-oriented programs, and (relational) database backends providing persistent storage. 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

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.

Proceedings of SIGMOD2007 Ph.D. Workshop on Innovative Database Research 2007(IDAR2007), June 10, 2007, Beijing, China.

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 languages. In contrast to imperative languages that expect developers to provide a detailed specification of the application execution plan, 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.

2. APPROACH

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 and messaging systems, 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. Our goal is to combine these successful concepts, thus yielding a scalable, easy-to-use XML messaging system architecture.

2.1 Declarative Application Language

Our idea is to specify the application logic as a set of declarative rules, describing the message flow between message queues. Queues are not only used as asynchronous communication gateways, but also as persistent and durable storage containers, keeping state information for both local application modules and external processes. Whenever a new message gets inserted into a queue, rule execution is performed. 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.

We want to allow for the convenient specification of XML messaging applications by using a declarative, XML-aware programming language. By directly operating on XML messages (and queues as their storage containers), this approach

avoids the impedance mismatch that plagues most of today's systems. However, existing, declarative XML query languages, such as XPath and XQuery, lack essential features that are required for application development using our message- and queue-based paradigm. These include, among others, the possibility to interact with queue data structures, creating side effects and XML-messaging facilities. As a result, the question arises whether these languages can be used in a rule language and whether they can be extended while keeping the resulting language fully compositional.

By building on the XQuery Update Facility [3], our approach provides 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 allows them to perform update operations. In order to achieve a seamless integration of XQuery expressions with our queue-based processing model, we extend the XQuery Update Facility with additional, system-provided functions and update operations. These can e.g. be used to retrieve all messages stored in a particular queue or to insert new messages. 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.

2.2 Keeping the 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*.

Today's application servers involve considerable effort for performing complex context management operations (e.g. hydration/dehydration in [1]). Context management becomes necessary in order to deal with multiple, concurrent application instances, each of them having specific state information. To avoid this performance-consuming management overhead, our idea is to refrain from explicitly materializing instance context and using local variables. Instead, we retain all messages being sent and received and dynamically determine the application state by querying this message history. Thus, a large number of concurrent application instances can be supported without an expensive context management.

As (almost) any system will eventually run out of message storage capacity, 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 specified *declaratively*.

2.3 Efficient Application Execution

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 pro-

vide the runtime system with a greater degree of freedom. This principle is one of the main reasons why declarative 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, where queries mainly target incoming messages, and traditional databases, where query execution is performed against a persistent store. Particularly, we want to investigate whether it is possible to find 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. Even more challenging is the question whether it is possible to develop a cost model for optimizations.

2.4 Queue-Based, Reliable XML Storage

The realization of the message processing primitives of our language requires efficient, reliable XML message queue storage facilities that are 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). However, these systems are not designed to provide queue-based data structures and do not incorporate messaging operations. As a consequence, it has to be investigated how XML queue data structures can be integrated into the database kernel of an XDS while preserving both high-performance queue and database operations. If this is possible, we could benefit from both the advantages of message queues and native XML data stores. Particularly, it would allow us to avoid having redundant functionality (such as transaction control) in two different system components and save us from crossing system borders when forwarding data between the database and the messaging infrastructure. Given that queue data structures can be successfully integrated, the next interesting issue is whether we can find corresponding index structures. This would allow us to speed up processing by retrieving individual messages from our queue-based storage without searching the whole message history.

We refer the reader to our technical report [2] for a more comprehensive description of our declarative application language and the corresponding runtime system. It also includes additional literature references, as well as a discussion of related work and the current status of our research.

3. REFERENCES

- [1] S. Blanvalet. Managing a BPEL production environment. Technical report, Oracle Corporation, 2006.
- [2] A. Böhm. Declarative development of distributed applications. Technical report, University of Mannheim, 2007.
- [3] D. Chamberlin, D. Florescu, and J. Robie. XQuery update facility. Technical report, World Wide Web Consortium, July 2006. W3C Working Draft.