Routed Message Driven Beans:
A new Abstraction for using EJBs

Erik Wilde and Manfred Meyer
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology, Zürich

Abstract

Asynchronous messaging between cooperating software components proves to be useful in many scenarios. One framework supporting this functionality is Sun's J2EE platform with its *Message-Driven Beans (MDB)* model. We present a novel way to use MDBs by providing a way to add routing information to the messages, which is then used to send a message through a given path of processing components. We call this model *Routed Message-Driven Beans (RMDB)*, and the two main topics that are important for RMDBs are (1) the message format that is used for the routing information, and (2) the API which can be used by programmers to take advantage of the abstraction provided by RMDBs. Performance measurements show that the overhead caused by our RMDB framework is acceptable if messages are routed through several EJBs.

Contents

1 Motivation

Programming frameworks are used to make program development easier, faster, and more reliable, and the overall goal is to produce better software with less expenses. In the last 10 years, the *World Wide Web (WWW)* [15] — and in particular applications built on top of the services provided by it — has increasingly been the focus of software developments, as indicated by frequently used keywords such as *Intranet*, *Business-to-Business (B2B)* and *Business-to-Consumer (B2C)*, or other keywords referring to Web- or Internet-specific technologies. Thus, programming environments for Web-enabled applications have become quite popular.

In the context of a Web-focused software system, we have developed the *Routed Message-Driven Bean (RMDB)* abstraction for asynchronously routed messages between *Enterprise Java Beans (EJB)*, based on a pre-computed path through a any number of RMDBs. These RMDBs all contribute to the processing of the message as cooperating software components. In the following sections, we briefly introduce the topics relevant to our work.

1.1 Web Servers

Web server architectures have gone a long way from the first Web server (implemented in 1989/90 [1]) to the complex integrated server architectures of today. The first Web server simply mapped the request's URI path to a file system path, and then returned the file's content in the response. While the communications mechanism between Web clients and servers, the *Hypertext Transfer Protocol (HTTP)* [5], basically still works the same way than it did 10 years ago (clients send a request with a URI path and get back a response containing a result), the architectures on the server side have evolved significantly. Three basic evolutionary steps can be identified:

- *Server directives*

  Simple directives for invoking pre-defined server operations, such as inclusion of a file's content or inclusion of the modification date of a file. These mechanisms are very easy to use, but also very limited in their functionality. Extending the server's functionality means extending the server's set of supported directives, which normally involves modifying the server software.

- *Scripting languages*

  Scripting languages are still very popular today, the most widely used are probably PHP[1], Sun's *Java Server Pages (JSP)*, and Microsoft's *Active Server Pages (ASP)*. Scripting languages are embedded into Web pages, and the Web server scans the Web page for scripting before delivering it. Scripting languages are a very powerful way to embed dynamically generated information into static content.

  Scripting languages can be used in conjunction with other technologies, a very popular example being the *LAMP (Linux, Apache, MySQL, PHP)* application suite [14], which makes it possible to easily access relational databases from within PHP's scripting environment. However, the applications used by the scripting languages are outside of the scope of the scripting language itself.

---

[1]See http://www.php.net/

- *Programming environments*

  Complete programming environments for Web services not only include a scripting language for embedding dynamically generated information into static content, but also an entire programming framework for building Web-based applications. These environments are undoubtedly more complex than the rather simple scripting languages, but they are becoming increasingly popular due to the growing number of complex Web-based applications.

One important class of programming environments are application servers based on Sun's *Java 2 Enterprise Edition (J2EE)* [11], which is a platform based on a Java run-time environment and a number of additional services. J2EE is not the only programming environment for Web-centric applications (Microsoft's .NET platform is a well-known competitor), but with dozens of implementations (including several free software initiatives) it is the most widely implemented one.

J2EE is a specification from Sun Microsystems [12], including a large number of technologies, most notably a Java runtime environment as the most basic component, *Java Database Connectivity (JDBC)* for database connectivity, *Java Server Pages (JSP)* as the scripting approach for Web pages, and EJB as its software component model. In the latest release of J2EE (version 1.3), the *Java Message Service (JMS)* [7] and the *Java Transaction API (JTA)* [3] have been added to better support EJB development. Furthermore, the following enhancements have been made to the architecture:

- A new kind of enterprise bean, the *Message-Driven Bean (MDB)*, enables the asynchronous consumption of messages.

- Message sends and receives can participate in JTA transactions, thus making it possible to combine JMS-based asynchronous messaging with the transaction services provided by JTA.

In many application scenarios involving several autonomous services, asynchronous messaging can be a very powerful abstraction. In our application scenario, we have a large database of URI references and semantic relationships between them, and we are implementing a server that can make use of this data in a variety of ways, effectively implementing a specialized database for linking information (often also referred to as a linkbase [4]).

1.2 XLinkbase

The XLinkbase system is based on the observation that it is not the amount of information available on the Web which often limits our ability to use it in a meaningful way, but the lack of relationships between individual information resources. XLinkbase is based on the idea of an *Open Hypermedia System (OHS)* [9], and its data model is similar to Topic Maps [8,10]. In the context of this paper, the interesting aspect of XLinkbase is that its current implementation is built on top of the J2EE platform and JMS. For a more detailed discussion of the ideas and concepts behind XLinkbase, please refer to a paper by Lowe and Wilde [16].

Basically, requests to the XLinkbase server may be issued via HTTP to a Web server, or from other applications via an EJB API. The request is then handled by a controller EJB, which computes routing information for the request and then sends it as a message via the RMDB mechanism. The routing information specifies which EJB(s) should process the

message, and the RMDB framework makes the routing transparent for the XLinkbase-specific EJB code.

Seen from an architectural perspective, the RMDB framework is used by the XLinkbase server to implement message routing among the EJBs within the XLinkbase server. These EJBs may implement functionalities such as XSLT transformations, logging, security checks, access to data storage for XLinkbase data (such as access via JDBC), or any other functionality that might be useful within the XLinkbase server. XLinkbase is designed as a generic platform for implementing linkbases, and it can easily be extended by implementing new functionality in RMDB components, which can then be used to process requests.

A typical example of an XLinkbase request is the following scenario: The XLinkbase Web server receives a request and forwards it to the controller EJB. The controller computes a processing graph, consisting of any number of RMDBs, for example a logging component, a security check, a database access component retrieving data, an XSLT transformation component converting the XLinkbase data to HTML, and another logging component, before the controller finally gets back the result of the request processing and sends it to the client via the Web server. All the components are RMDBs, and they use the RMDB framework to transparently route the message between the components. If at any time the processing graph changes (for example because the security components rejects the request and needs to re-route it), the RMDB routing information can be changed via the API to reflect the new processing graph. In general, all interaction between the application functionality and the RMDB framework is handled via the RMDB API.

2 Related Work

Web-based application frameworks are very popular today, and they range from rather small software suites to rather heavy programming environments such as J2EE. One of the early approaches to use XML technologies for building a publishing framework is Apache's *Cocoon*[2] project, which is built on standard Web technologies such as DOM, XML, and XSL(T). However, Cocoon does not support distributed processing and load balancing, and because we wanted to be able to support this, we decided to not use it.

One step further, the *Java 2 Platform Enterprise Edition (J2EE)* supports distributed processing and load balancing, and (starting with version 1.3) also supports message passing between components. Furthermore, J2EE is a specification (and not a product), and therefore software built on top of it is not restricted to one vendor's platform. Even though most J2EE products are commercial products (with BEA System's *WebLogic* and IBM's *WebSphere* currently being the market leaders), there are also open source J2EE implementations such as JBoss[3].

J2EE provides a powerful programming environment, but from an application programmer's point of view, the level of support for the typical J2EE application is not as good as it could be. Sun Microsystems, author of the J2EE specification, has recognized this and now offers the so-called *J2EE Blueprints*[4], which basically are design patterns [6] for using EJBs. At the time of writing, the set of blueprints available is quite small, and we see our framework as one contribution to the overall goal of producing generic solutions for J2EE, which may be

---

[2]See http://xml.apache.org/cocoon/
[3]See http://www.jboss.org/
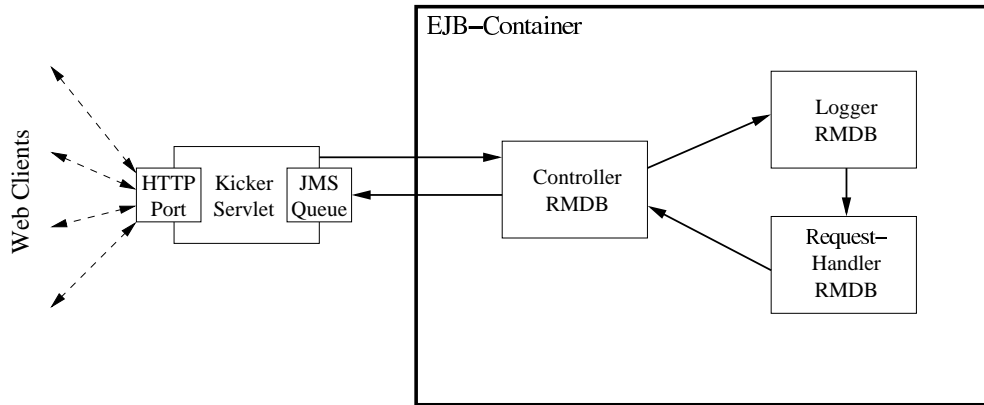[4]See http://java.sun.com/j2ee/blueprints/

Figure 1: XLinkbase server

re-used for similar application scenarios.

Asynchronous messaging is provided in many environments[5], but the ability to compute (and modify) paths through any number of components, and to pass the path information as well as any results through these components, to our knowledge is not available as a generic component in any J2EE platform.

3 The Concept

In a very simple setup, an XLinkbase server could be constructed as shown in Figure 1. A servlet is set up to receive client requests from browsers or other Internet applications. This servlet then kicks off the RMDB mechanism by sending a JMS message to the Controller in the EJB container. The Kicker servlet must have its own JMS Queue to be able to receive the response in order to send it back to the client.
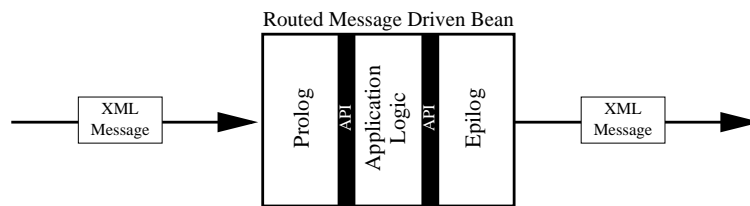

Figure 2: RMDB framework

RMDBs are based on the *Message-Driven Beans (MDB)* model of the new *Enterprise Java Beans (EJB)* 2.0 specification. MDBs are a very simple type of EJBs. They are exclusively addressed through the JMS queue or topic they are deployed to listen to. There is no `home` and no `remote` interface and they can be coded in one single class file by simply implementing two interfaces.

---

[5]IBM's *MQSeries* and Microsoft's *MSMQ* currently are probably the most widely deployed messaging platforms.

Like any MDB, a RMDB starts to work when a message arrives in the JMS queue. In the MDB model, next to the *JMS Queues* there also exist *JMS topics* which make it possible to send a message to many recipients. However, for RMDBs these JMS topics are not supported, because the message path is not allowed to split anywhere in the processing path, as there can only be one message per request in the server. All communication is done using JMS text messages, because they can carry a payload of type `string`, which enables the RMDBs to send and receive XML strings. The message is an XML document containing all the information about what the RMDB is supposed to do:

- The command and its parameters

- The data to process

- The route to the next RMDBs

The message format and its semantics are described in detail in Section 4.1. Before the application logic of the RMDB has access to the data in the message, the RMDB message (which is an XML document) must be parsed. After processing it, a new message must be compiled and serialized to a `string`, in order to send it to the next RMDB. Figure 2 shows the schematic message flow through one RMDB.

The parsing is done in the RMDB's prolog, while the generation, routing, and sending of the new message is done in the epilog phase. This behavior is identical for every RMDB, and consequently the code for doing this can be encapsulated in a framework to be transparently processed at the beginning and the end of an RMDB's execution. Creating a new RMDB using this framework is then reduced to implementing the application logic. The access to the XML data is provided by an API and similarly data can be added to the message as well as adding new commands to the path for delegating work to other RMDBs.

4   Implementation

Two main aspects discussed in this section are the format of the XML message, and the class structure used to implement the routing and data handling mechanisms. The actual XML message is never visible for the RMDB programmer, since access to it is exclusively provided through an API. Because of the sophisticated design of the message format, RMDBs can be kept stateless and the processing of messages can be easily parallelized and distributed.

The class structure described in Section 4.2 has been invented to hide framework code from the developer who wants to create new RMDBs and to provide a hook to insert the RMDB's application logic into the framework. The transparency achieved by this is a relief for programmers, reduces the opportunities to make errors in the own code, and enables rapid software development.

4.1   RMDB Message Format

The XML structure of messages exchanged by RMDBs is shown in Figure 3. It contains the document element `message` with four child elements. The message DTD is listed in Figure 4, and the document element's child element types have the following semantics:

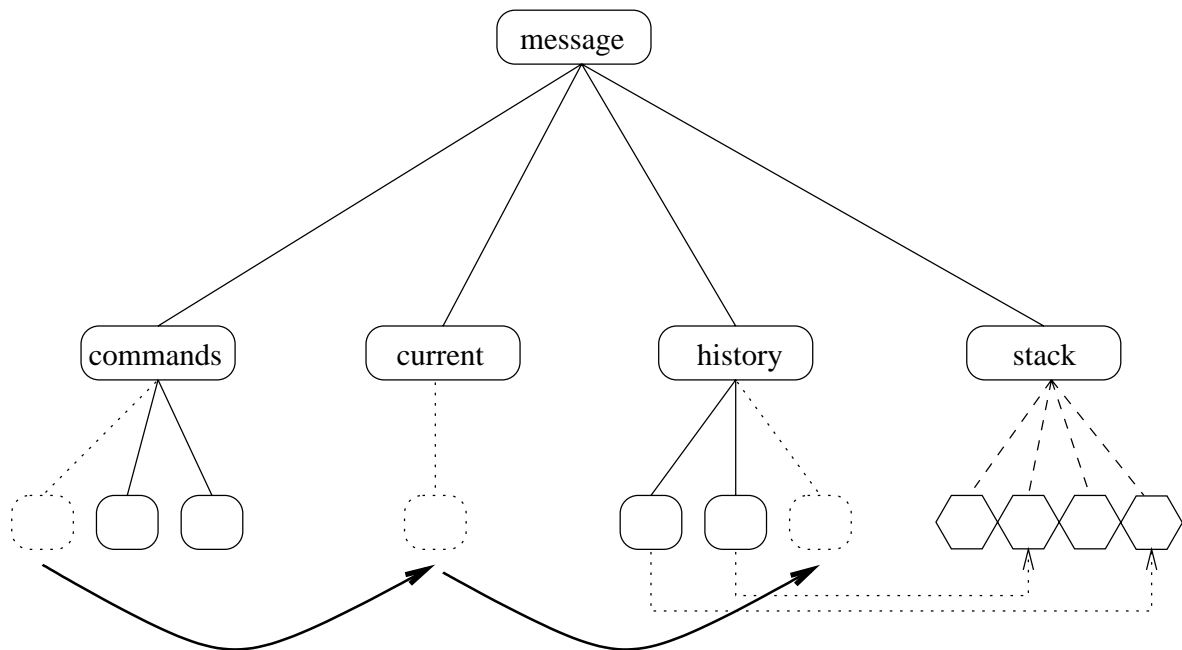Figure 3: RMDB message structure

```
<!ELEMENT  message   (commands,current,history,stack) >
<!ATTLIST  message
  history  ( yes | no )  "yes" >
<!ELEMENT  commands  ( rmdb* ) >
<!ELEMENT  current   ( rmdb ) >
<!ELEMENT  history   ( rmdb* ) >
<!ELEMENT  stack     ( data* ) >
<!ELEMENT  rmdb      ( arg* ) >
<!ATTLIST  rmdb
  name     CDATA         #REQUIRED
  data     IDREFS        #IMPLIED >
<!ELEMENT  arg       ANY >
<!ATTLIST  arg
  name     CDATA         #REQUIRED >
<!ELEMENT  data      ANY >
<!ATTLIST  data
  id       ID            #IMPLIED >
```

Figure 4: RMDB message DTD

commands: A list of the following RMDBs which are scheduled to further process the request. They are registered by their JMS queue name, and may contain several arguments (wrapped in `arg` elements) to customize their behavior or tell them on which data stack entries to operate. The `command` element can be regarded as a stack, since the topmost command is always next to receive the message. Newly added commands are pushed on the top of the stack exclusively. This ensures that the message finally returns to the client and that the order of the RMDBs on the stack remains unchanged, as they might rely on that by passing information on the data stack. For the case of an unrecoverable error occurring in an RMDB so that normal processing is not possible anymore, the API function `sendException` is available, which interrupts the routing and returns an error message to the client.

current: The current command, representing the RMDB which is about to process the message. We use an own branch of the XML tree to enable the framework to easily access it. This means to read the arguments from it, or add references to freshly generated data stack entries. During the RMDB's epilog phase, just before the message is sent to the next RMDB, the current command is exchanged. The first element in the `commands` stack is moved here, while the replaced one is appended to the `history`.

history: This branch contains a list of all successfully processed commands. A developer could use this for logging or debugging purposes as well as for discovery of data on the data stack, learning by which RMDB it has been produced. If an application developer does not intend to access the `history`, the XML attribute `history="no"` can be set in the document element of the message to disable the RMDB history mechanism, in which case executed commands will be discarded instead of being added to the history.

stack: This is a data stack where all the information needed while processing a request can be stored. Its entries can be of type `string` or of arbitrarily complex XML structures. For retrieval, they can be addressed implicitly through the position on the stack, or explicitly by their ID. The ID is automatically generated by the framework as a unique number[6], and returned to the application logic for later reference.

The complete XML DTD of RMDB messages is shown in Figure 4. Since we assume that messages are only produced and consumed by the RMDB framework itself, we currently do not validate the messages (ie, upon arrival at an RMDB, the message is only checked for wellformedness), but this behavior could be changed easily with switching on validation of the XML parser.

4.2 The Class Structure

Regular J2EE MDBs do not inherit any code from superclasses and are easily constructable by implementing two interfaces. However, the EJB specification 2.0 explicitly allows to use subclassing for MDBs. This gives us the possibility to write our own RMDB class, including the prolog and epilog code, the API to access the message, and a hook for later insertion of the application logic. The `RMDB` class implements all necessary methods of the interfaces `MessageDrivenBean` and `MessageListener` to fulfill the requirements set by the EJB container (see Figure 6).

---

[6]Including a leading character to satisfy the XML naming constraints for ID attributes.

```
<message history="yes">
  <commands>
    <rmdb name="Controller"/>
    <rmdb name="Kicker"/>
  </commands>
  <current>
    <rmdb name="Monitor"/>
  </current>
  <history>
    <rmdb name="Controller" data="i0"/>
    <rmdb name="Logger"/>
    <rmdb name="RequestHandler" data="i1">
      <arg name="cmd">request=0</arg>
    </rmdb>
  </history>
  <stack>
    <data id="i1">
      <!-- XML data omitted for brevity -->
    </data>
    <data id="i0">request=linux</data>
  </stack>
</message>
```
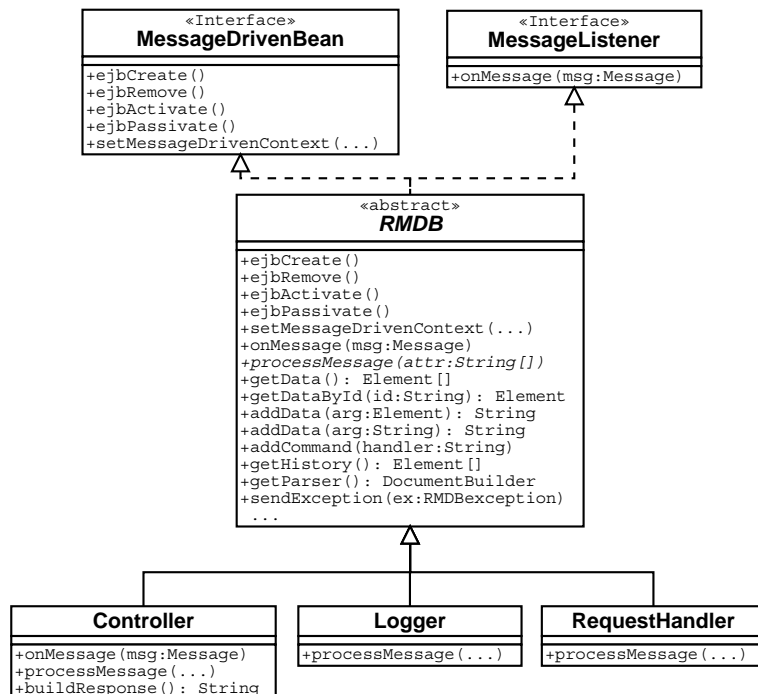
Figure 5: An example message

Figure 6: RMDB class structure

The `RMDB` class also contains the abstract method `processMessage`, which has to be overridden by a subclass, because it will be called to execute the actual application logic of the RMDB. This is the only programming work a developer has to do to construct a new RMDB. The example classes `Logger` and `RequestHandler` shown in Figure 6 demonstrate the simplicity of RMDBs.

The `Controller` class is a special case. At least one instance of this bean must be present in every RMDB based server and is located at the beginning and the end of every message path. It is designed to handle the direct communication with the client. Upon reception of a new request, the controller builds a new XML message frame, adds a default route to the command stack, and the actual request as a first data stack element. The default route always leads to the Controller a second time at the end of the path. This is the moment when the Controller finally removes the message frame and passes the response to the Client. It is the responsibility of the extra method `buildResponse` to return the correct response string. By default, it simply returns the top of stack element as a plain text string. This is a simple default strategy, and it is possible to override this method if a different behavior is required (for example for passing complex data structures as responses).

5 Performance

Programming RMDBs means programming on a very high level of abstraction. This is very comfortable, but there are a lot of lower software layers (the Java Virtual Machine, the J2EE application server, and the RMDB class) which all consume CPU time and other system resources. To get an idea how RMDBs perform, we measured their minimal processing time compared to the simpler MDBs (which provide the foundation for our RMDBs). Figure 7 shows the chart of the benchmark measurement. The X-axis shows zero to five Beans. On the lower line there are just MDBs, the upper one has been measured using RMDBs containing the full routing and XML messaging mechanisms. Both types of beans do not contain any application logic.

The Y-axis shows the time for a round trip from a client application through a variable number of beans and back to the client. For the RMDBs as well as for the MDBs, the total time depends linearly on the number of beans in the path. In the measurement of the RMDBs, the controller bean has not been counted as a working bean. Hence the value in the chart for zero RMDBs is 32ms and contains the time the controller needs to set up the initial message and communicate with the client. The client application also consumes about 4ms in both test cases.

The measurement has been made on Intel Pentium III processor machine with 500MHz and 384MByte RAM running Linux 2.2.14 and the BEA Weblogic 6.0sp1 server. The application server has been advised to create a sufficient amount of beans at deployment time, so the time for instantiating and initializing them did not affect the measured delay. Besides that, no further optimizations have been made for the application server, but there are plenty of parameters to tweak in the Weblogic server and in the operating system.

From the measured data, it can be concluded that the average time needed by a single RMDB is 17.4 milliseconds to fulfill all the required parsing, routing and communication work. This is to see as a minimal response time under very low work load. But for applications with very high work load the system is designed to achieve a good load balancing to keep the response time acceptable.
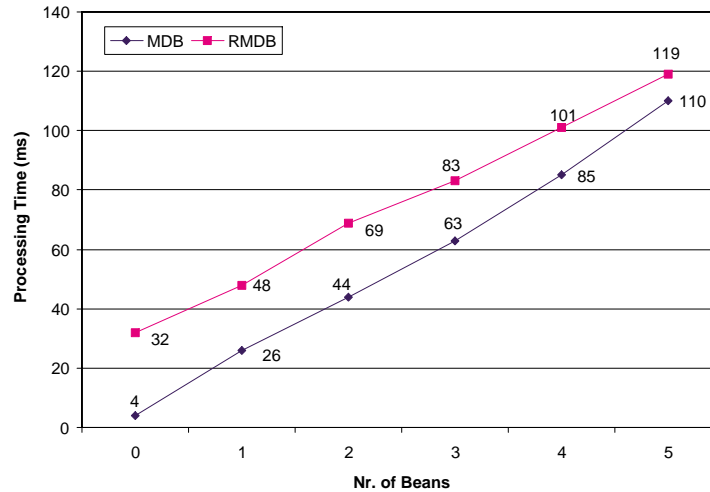
Figure 7: RMDB Performance

Naturally, performance for RMDB depends on many factors, the most important one being the size of the messages, ie the size of the data being sent through the RMDB message path. If there are big and complexly structured messages, parsing and serializing them will be more expensive than in our example, which used a very simple message structure. However, XML parsing and serialization may be neglected if the application logic is very complex and time-consuming and therefore predominates the overhead caused by the RMDB framework and the lower layers (JVM and J2EE).

6 Conclusions

We started thinking about an abstraction for routing messages through a number of software component as a result of our work on the XLinkbase system. When we started our work on the XLinkbase server, JMS was not part of J2EE, and we were very pleased with the inclusion of the messaging service. However, we discovered that asynchronous messaging as provided by MDBs was not as powerful an abstraction as we were looking for. In an effort to create a generic solution to the problem of a message flow through a complex set of software components, we invented the concept of the *Routed Message-Driven Bean (RMDB)*, which proved to be very useful in our application scenario.

It was our goal to separate the generic concept of RMDBs from our XLinkbase application, and we believe that we have succeded in creating an abstraction which might prove useful in a wide variety of application involving asynchronous messaging. It would be pretentious to think of RMDBs as something to be included in the next release of J2EE, but we believe that something similar to it, ie supporting a more abstract way of messaging as through the rather simple mechanisms of JMS queues and topics, would be an interesting idea. However, there are still some open issues, which we discuss shortly in the following section.

7 Challenges and Future Work

So far, we have concentrated on making the RMDB framework as generic and flexible as possible, so that it may be re-used in similar scenarios than our linkbase application. However, one important aspect we have not yet implemented is the issue of reliability. While JMS guarantees the delivery of messages, there is no transaction concept. We are currently investigating how to integrate J2EE's JTA service with our RMDB abstraction for implementing transactional semantics, ideally being able to make the whole RMDB processing of a request one transaction. Our plan is to extend the RMDB API and the RMDB message format with support for transactions, so that critical applications may choose between the more efficient and faster messaging service as provided by JMS only, and a more expensive, but more reliable transactional variant implemented by using JMS and JTA services.

Another issue we would like to investigate is the support for conditional paths within the messages, where the routing of RMDB messages is based on the result of computations earlier in the path. Again, this would require changes to the API as well as to the message format, but we believe that conditional routing could be useful in many scenarios, the most widely known being the handling of exceptions (ie, error conditions) in a more general way than only providing one exception handler.

We currently use an XML DTD to describe the RMDB message format. We are currently looking into using XML Schema [13, 2] for the schema definition, which would enable us to specify a better formal model of the message format. In particular, XML Schema's datatype concept could be used to make the RMDB code lighter by moving more data checking into the validation of the messages. However, this would require using a schema-validating XML parser, and we still have to look into the performance implications of such a step.

In our current RMDB framework, the calculation of the processing graph has to be done by the controller, and is completely handled by the application logic. One could easily think of making the RMDB model even more abstract by specifying certain constraints and conditions for each RMDB available to the controller, and then having the RMDB code compute the actual processing graph based on the incoming request and the RMDB specifications. This, however, would be a very hard task to solve generically, which is the main reason why we did not put any effort into it.

8 Acknowledgements

References

[1] TIM BERNERS-LEE. The World Wide Web. In *Proceedings of the 3rd Joint European Networking Conference*, Innsbruck, Austria, May 1992.

[2] PAUL V. BIRON and ASHOK MALHOTRA. XML Schema Part 2: Datatypes. World Wide Web Consortium, Recommendation REC-xmlschema-2-20010502, May 2001.

[3] SUSAN CHEUNG and VLADA MATENA. Java Transaction API (JTA) — Version 1.0.1. Technical report, Sun Microsystems, April 1999.

[4] Steven J. DeRose, Eve Maler, and David Orchard. XML Linking Language (XLink) Version 1.0. World Wide Web Consortium, Recommendation REC-xlink-20010627, June 2001.

[5] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. Internet proposed standard RFC 2616, June 1999.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, January 1995.

[7] Mark Hapner, Rich Burridge, Rahul Sharma, and Joseph Fialli. Java Message Service — Version 1.0.2b. Technical report, Sun Microsystems, August 2001.

[8] International Organization for Standardization. Information technology — SGML Applications — Topic Maps. ISO/IEC 13250, 2000.

[9] Peter J. Nürnberg and John J. Leggett. A Vision for Open Hypermedia Systems. *Journal of Digital Information*, 1(2), 1997.

[10] Steve Pepper and Graham Moore. XML Topic Maps (XTM) 1.0. TopicMaps.Org Specification xtm1-20010806, August 2001.

[11] Ed Roman. *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition.* John Wiley & Sons, New York, September 1999.

[12] Bill Shannon. Java 2 Platform Enterprise Edition Specification, v1.3. Technical report, Sun Microsystems, July 2001.

[13] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures. World Wide Web Consortium, Recommendation REC-xmlschema-1-20010502, May 2001.

[14] Luke Welling and Laura Thomson. *PHP and MySQL Web Development.* Sams, Indianapolis, Indiana, March 2001.

[15] Erik Wilde. *Wilde's WWW — Technical Foundations of the World Wide Web.* Springer-Verlag, Berlin, Germany, November 1998.

[16] Erik Wilde and David Lowe. From Content-Centered Publishing to a Link-Based View of Information Resources. In *Proceedings of the 33rd Hawaii International Conference on System Sciences*, Maui, Hawaii, January 2000. IEEE Computer Society Press.