Java theory and practice: Should you use JMS in your next enterprise application?

Learn how message queueing can improve the flexiblity and scalability of your enterprise applications

Brian Goetz (brian@quiotix.com)
Principal Consultant
Quiotix Corp

01 February 2002

Welcome to *Java theory and practice*, a new monthly column by veteran Java developer Brian Goetz. This column aims to explore that elusive juncture where design principles meet the requirements to solve real-world problems. Each month, we'll explore design patterns, principles of reliable software design, and why "best practices" are best, with an eye to how they are applied to real problems. This month, Brian takes a look at enterprise message queuing technology.
In recent years, enterprise message queueing (MQ) products have become more widely available to developers. When used properly, MQ technology can often improve the organization, performance, and scalability of your application. The Java Message Service (JMS), an integral part of J2EE, makes MQ services available to any J2EE application. In this first installment, Brian outlines some of the benefits of using message queuing in Java applications, and explores the types of problems that can benefit the most from MQ technology.

View more content in this series

Message-queueing (MQ) tools are not as well known or understood as database tools, such as relational (SQL) databases, which are a key component of nearly all enterprise applications and a great number of simpler applications. Developers have always had access to a wide range of database products, ranging from cheap, desktop-only databases (such as dBase or Microsoft Access), to workgroup database servers (such as Sybase SQL/Anywhere), to enterprise database servers (such as DB2 or Oracle.) Whatever your project, there is a database product with the mix of price, performance, and features you need.

Like databases, MQ products, sometimes referred to as message-oriented middleware (MOM), have been around for quite some time. However, until recently, MQ servers have been expensive, high-end products, available only to the most well-funded enterprise developers. As a result, fewer developers have been exposed to the benefits of using messaging in their applications.

Message queuing for the masses

Fortunately, this situation is starting to change; there are several lower-cost MQ servers on the market today. In 1997, Microsoft released MSMQ, a transactional message-queuing product, as an integrated part of Windows NT Server -- with no additional licensing fee. Sun gave messaging a big boost when it included the JMS API in the initial J2EE specification. And in version 1.3 of the J2EE specification, all J2EE containers are required to include a JMS provider.

JMS, the Java Message Service, is an API that allows Java applications to access a wide range of MQ servers (or, in JMS parlance, providers) through a standardized interface, just as JDBC allows programs to access many different database servers through a common interface. Most J2EE containers include a JMS provider; in the future, all J2EE containers will. JMS can also be used without a J2EE container; several stand-alone JMS provider implementations are available on the market. In addition, the EJB 2.0 specification introduces a new type of EJB -- the message-driven bean -- which makes it very easy to create message-driven components that make use of entity and session beans.

Now that we all have access to JMS services, we should learn to leverage the strengths of messaging in our applications. Willy Farrell, an e-business architect at IBM, wrote an excellent introductory tutorial on using JMS (see Resources). It covers the basics of creating messages and queues, and all the options for prioritizing, retrieving, and encoding messages.

Messaging and databases offer complementary strengths, and in many cases, using both messaging and relational databases together can yield a solution that is much better than either one by itself.

Historically, MQ servers have been used in event-oriented applications, such as financial service applications, or as a means of interfacing disparate systems (such as connecting disparate database applications or connecting one enterprise to another in a supply chain network.) The term "message-oriented middleware" is often applied to MQ servers and underscores the perception that the primary use for MQ technology is to connect disparate systems. However, with the cost of MQ products decreasing, many other applications can now benefit from message queuing.

What do MQ servers do?

In MQ parlance, a message is simply a stream of bytes (which could be an XML document, a serialized Java object, a text string, or even an empty message). The interpretation of the message is left to the application domain; the MQ infrastructure imposes no semantics or structure onto the message. Messages are stored in queues, and MQ servers allow you to enqueue messages onto queues, and dequeue messages from them.

So far, a message queue sounds a lot like a simple linked list. In its simplest form, that's exactly what it is, but enterprise MQ servers enhance functionality by wrapping this linked-list management with a number of features:

- Security to control who can write to or read from a queue

- Network interfaces, which allow message producers and consumers to be located in different places
- Transactional support, so enqueue and dequeue operations exhibit the transactional properties of atomicity, consistency, isolation, and durability
- Distributed transactional support, so queue operations can participate in distributed transactions with other resource managers, such as SQL databases
- Persistence
- Load balancing
- Failover
- Management

The strengths of MQ

The strengths of MQ derive from the inherent *loose coupling* provided by the asynchronous nature of message processing. The processes of posting a message to a queue by one entity and removing the message and processing it by another entity are completely separate. The two entities don't have to be running at the same time, be located on the same system, or even know the other's identity; they each interact only with the queue, on their own schedule. They only have to agree on the forms of messages they will accept from each other; otherwise, they don't have to know anything about each other.

Loose coupling has many advantages. It provides a natural mechanism for dividing a unit of work into smaller, independent components, which implicitly creates abstractions between the stages of processing a request. This subdivisioning allows us to easily abstract the implementation of each component from one another, better measure and manage the resource utilization by each component, or replace one component with another that provides similar functionality without having to change the other components.

The JMS specification requires that JMS providers also implement *publish and subscribe* features, which allow you to create distinct, application-defined channels called *topics*, and allow individual entities to subscribe to topics. A message queued to a topic is automatically placed on the private queue of any entity who is subscribed to that topic. Topics perform a valuable sorting function in financial services or news delivery applications. For example, while more than 5000 stocks are traded on the major US exchanges, each investor may be interested in only 30. So you could create a topic for each ticker symbol, let users subscribe to the symbols they are interested in, and then let the MQ engine do the work of displaying only the quotes each investor wants, avoiding the delivery of duplicate quotes. This process is much harder to implement with an SQL database.

Classic MQ usage patterns

There are a number of common usage patterns that are well-suited for using message queueing. When you see one of these in your application, you should consider using messaging.

Event-oriented applications

Highly event-oriented applications are very well suited to using MQ technology. These include financial services applications (think about a trading station that displays stock price updates, initiates trades based on changes in prices or the execution of other orders, reports on the status of orders, and so on), news-wire service applications, and supply-chain applications. In financial markets, events must be acted on quickly; you want to be notified when something important happens, as soon as it happens.

Polling a database

Databases are excellent for storing data persistently, but storing transient data and notifying us when the data has changed is not their strength.

Even though it is inefficient, polling databases is extremely common. The display monitors in every airport are constantly polling the database to update the information they are displaying. A bank with many teller windows often uses electronic signs to indicate the location of the next free teller. An order processing system inside an e-commerce site may poll the database to see if any new orders need to be processed. Or an insurance claims workflow system might poll to see if there are any new claims that can be assigned to available claims adjusters.

All of this polling makes for a lot of extra work. If there are many entities polling frequently (and they would have to, if they want updates in the data to be reflected quickly), this could create a substantial load on the database server and the network. Most of the time, polling returns no data, or worse, data we've already seen and have to process again or identify as having already been processed.

Databases are simply not designed for polling or for events. If you must take action relatively quickly after an event occurs or data changes, asynchronous messaging is by far the easier and more efficient way to accomplish this. Whenever you find yourself polling a database for an update, consider using JMS instead.

Workflow

According to the workflow portal (a cooperative effort by the Workflow Management Coalition and the Workflow And Reengineering International Association), workflow is "...the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules." Workflow applications (such as document routing and approval, insurance claims processing, and so on) are particularly well-suited to MQ solutions, as MQ technologies closely model how a workflow problem would be addressed in a paper-bound office, where each participant has an inbox and an outbox.

Workflow applications are characterized by having many agents (agents can be humans, automated processing steps, or even physical equipment such as machinery or printers) who each perform a small portion of the task and pass it on to the next agent as determined by business rules. Consider the process of approving and paying an electronic expense report. The employee creates and submits the report, and then the report needs to be approved by the employee's manager (and maybe by another level of management if the dollar amount exceeds some threshold). Then it goes to HR, where it will be checked for accuracy and scrutinized to ensure that the expenses are valid business expenses and comply with the company's policies. If approved, a payment request will be created and the check will be scheduled for printing. After that, it may go to accounting, where the individual charges will be applied to the appropriate accounts and cost centers. At each of these stages, the expense report may be bounced back to the employee or the employee's manager.

In building a workflow application, the primary design goals are to make sure work gets quickly from agent to agent, and to make sure that tasks don't fall through the cracks. MQ servers, working hand-in-hand with databases, make it easy to build flexible, scalable, and extensible workflow processing into your application.

Using MQ to get risky operations off the critical path

The process of accepting, approving, and filling an order in an e-commerce or supply chain application has a lot in common with workflow applications, although most of the steps involve electronic participants rather than human ones. The acceptance and fulfillment of an order might include some or all of the following steps:

- Accept the order and store it in the database.
- For consumer customers, validate the credit card.
- For commercial customers, check the customer's credit, either with a credit reporting agency or with your company's credit department.
- Perform some fraud-checking analysis.
- Check inventory.
- In the case of multiple fulfillment centers, decide which centers will fill the order.
- Send a confirmation e-mail to the customer.
- Send a notification to the customer's sales rep.
- Generate picking lists and route them to the fulfillment centers.
- Ship the order.
- Charge or bill the customer.

You don't want the customer to have to wait very long after clicking "Buy" to get their confirmation number and receipt, so whatever code path gets executed as a result of submitting the order should be short and have predictable execution time. But many of these steps involve accessing resources that may or may not be available at the time the customer places the order, or whose response time may be unpredictable. In this case, they should be taken off the critical path; let the initiation of the order set the chain of events in motion, but get the user out of the loop as quickly as possible by having the submission step do the absolute minimum to initiate the order. In addition to giving the customer a better user experience, separating the order processing into multiple discrete (and shorter) steps improves resource utilization and reduces contention -- it means transactions are shorter (so locks will be released earlier), and resources that are used in one step (such as network or database connections) are released earlier.

One of the least predictable steps in the order-processing is sending the confirmation e-mail. Mail servers are often congested and may take a long time to accept a message, or may reject the connection entirely. If the customer's mail server rejects the confirmation message, you want to retry sending it later. In this way, sending the confirmation e-mail is "risky" because it may not succeed the first time or may take a long time to process, and you don't want to make the customer (or anything else, for that matter) wait until the confirmation is sent. Similarly, the inventory may be stored in a separate database from the order processing (the inventory database may even be owned by an outsourced fulfillment service) and may not be available at the time the order is placed. Or the credit department may be off on Fridays.

By using message queues to store pending confirmation e-mails, inventory checks, or credit checks, you can separate the "risky" operation from the rest of the process, and thereby insulate the rest of the process from the risk that the operation will fail or take a long time. This also tends to simplify error-handling of each task considerably, as each processing step is only doing one simple task.

Conclusion

When applied correctly, message queuing (MQ) technology can greatly simplify the processing of complex tasks by facilitating decomposing the task into simple subtasks, and can increase the flexibility and scalability of our applications. As of J2EE version 1.3, every J2EE container will include a JMS provider, which means that we all will be able to take advantage of the power of asynchronous message queueing in our applications.

Next month, we'll explore the workings of the Java Transaction Service (JTS). Although it gets much less attention than EJB or other elements of J2EE, JTS is perhaps the most critical part of J2EE -- transactions are what enable us to build reliable distributed applications. We will revisit MQ in a future column and build a workflow application to demonstrate how message queuing and relational databases offer complementary strengths.

Resources

- Get specifications, references, and examples on JMS from Sun.
- *Java Message Service* by Richard Monson-Haefel and David Chappell (O'Reilly and Associates, 2000) offers a throrough look at the JMS API and provides many practical examples for putting it to work.
- *Enterprise JavaBeans* by Richard Monson-Haefel (O'Reilly and Associates, 2000) is a fully updated edition featuring a detailed look at the changes in the EJB 2.0 specification.
- In his tutorial Introducing the Java Message Service (developerWorks, August 2001), Willy Farrell gives an excellent overview of JMS and offers the basics for developing programs that use it.
- The e-workflow workflow portal, a joint effort by Workflow Management Coalition and the Workflow And Reengineering International Association, provides comprehensive resources for engineers developing workflow applications.
- Learn to use MQSeries with JSP technology in this article from the WebSphere Developer Domain.
- Find other Java related resources on the developerWorks Java technology zone.

About the author

Brian Goetz

Brian Goetz is a software consultant and has been a professional software developer for the past 15 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, CA. See Brian's published and upcoming articles in popular industry publications.