How-To Java by Todd Sundsted

Messaging makes its move, Part 1

Learn how to build your own Java-based messaging service

Summary
Messaging products have found their niche within enterprise applications as tools for coordinating distributed application components. What they do is remove the responsibility for ensuring messages are delivered reliably and correctly from the application components. This month, Todd demonstrates how to build a Java-based messaging service. *(2,000 words)*

Building enterprise applications around distributed architectures may solve some problems, but it introduces others. Consider for a moment a few of the ways in which communication can fail between components in a distributed application: a component can voluntarily, but unexpectedly, go offline; a component can unexpectedly terminate; a component can fail to respond to a request in a reasonable amount of time; or communication can be severed in the midst of a request. Given enough time, you could make this into a very long list!

I'm not advocating we turn away from distributed architectures, of course. None of the problems I mentioned above are insurmountable. Rather, I'm highlighting the need for a class of products that addresses the issues I raised above, preferably without burdening the application components with too many of the supporting details.

Messaging provides one possible solution to these problems. Messaging products allow distributed application components to communicate and coordinate their activity (via messages) by providing critical services such as message queuing, guaranteed once-and-only-once delivery, priority delivery, and transaction support.

Mail this article to a friend

In this column and the next, I will take you inside a messaging service and show you how it's put together. In the process, I'll highlight the Java Message Service (JMS) API, Sun's standard Java interface to messaging services, and teach you how to use it.

Roadmap

This column is the first in a series of two articles on the subject of messaging. In this column, I'll introduce the subject, discuss the JMS API, and show you how to build the underlying functionality. In next month's column, I'll show you how to add a JMS API specification-compliant interface. I don't plan on implementing the entire JMS API (writing a robust messaging service is far from trivial), but I do intend to provide you with a complete and fully functional subset of the API.

If you've never looked closely at messaging services before, you might want to begin by reading a brief introduction to them before plunging headlong into the code. I recommend my coverage of the JMS API specification (*JavaWorld,* January 1999).

Building a Java-based implementation of the JMS API

The JMS API specification defines the Java language interface to a messaging service. All current implementations of the JMS API wrap existing messaging products. There are at least two reasons for this:

1. It's easier to write a new interface to an existing product than to create a product from scratch.

2. Most potential customers are looking for products that will facilitate Java's integration into an existing enterprise application using an existing messaging service.

The result is a heavyweight implementation of the JMS API -- heavyweight because the implementation requires the support of an external, native messaging product.

It turns out that there are some practical reasons why one might want to provide a Java-based implementation of the JMS API. First, the implementation would be lightweight -- It wouldn't depend on external, non-Java resources. Second, it would provide true platform independence thus more fully realizing Java's write-once, run-anywhere promise.

The foundation

In order to ease the assimilation of the material that follows, I'll provide definitions for three key messaging concepts. Then I'll present simple examples of the two messaging paradigms the JMS API supports.

> **Message** -- A *message* is a lightweight object consisting of a header and a body. The header contains identification and routing information. The body contains application data.
>
> **Destination** -- A *destination* is an object that encapsulates the addressing information used by a messaging product to locate the end point for message delivery. It is not, in itself, an end point.
>
> **Client** -- A *client* is any object, process, or application that makes use of a messaging service. No distinction is made between traditional "clients" and "servers" -- they are both messaging clients.

The JMS API specification defines two different messaging paradigms: *point-to-point* and *publish/subscribe*. These two paradigms, or domains, represent the two leading models of messaging provided by existing messaging products. Both domains can be characterized by the type of destination for message delivery and the pattern of interaction between destination and client.

Point-to-point
The destination in the point-to-point domain is called a *queue.* The simple point-to-point example in Figure 1 contains a single queue and two clients. One client sends messages to the queue, and the other receives messages from the queue. Assume the queue and both of the clients already exist and are connected as illustrated.

Both clients already hold active connections to the queue: both the `QueueReceiver r` and the `QueueSender s` have been initialized. The receiver `r` and the sender `s` are used to communicate with the queue.

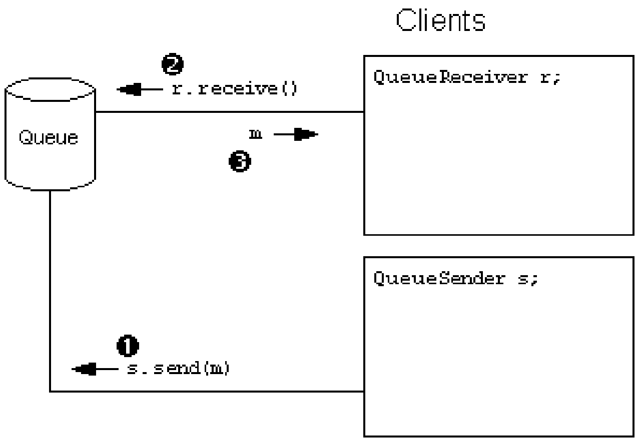

Figure 1. Diagram of point-to-point interaction

The characteristic pattern of interaction is as follows:

1. The sending client arrives at a point in its processing when it's necessary to send a message. It creates the message and sends it by invoking `s.send(m)`, where `s` is the sender and `m` is the message. The message is held in the queue.

2. The receiving client pauses and checks the queue for messages. It invokes `r.receive()`, where `r` is the receiver created earlier.

3. The message is removed from the queue and returned to the receiving client.

As you can see from the example, the point-to-point model provides synchronous delivery of messages.

Publish/subscribe

The destination in the publish/subscribe domain is called a *topic.* The simple publish/subscribe example in Figure 2 contains a single topic and two clients. One client subscribes to the topic and the other publishes messages to the topic. The topic takes care of distributing messages from the publisher to the subscriber. Assume the topic and both clients already exist and are connected as illustrated.

Both clients already hold active connections to the topic: both the `TopicPublisher p` and the `TopicSubscriber s` have been initialized. The publisher `p` and the subscriber `s` are used to communicate with the topic.

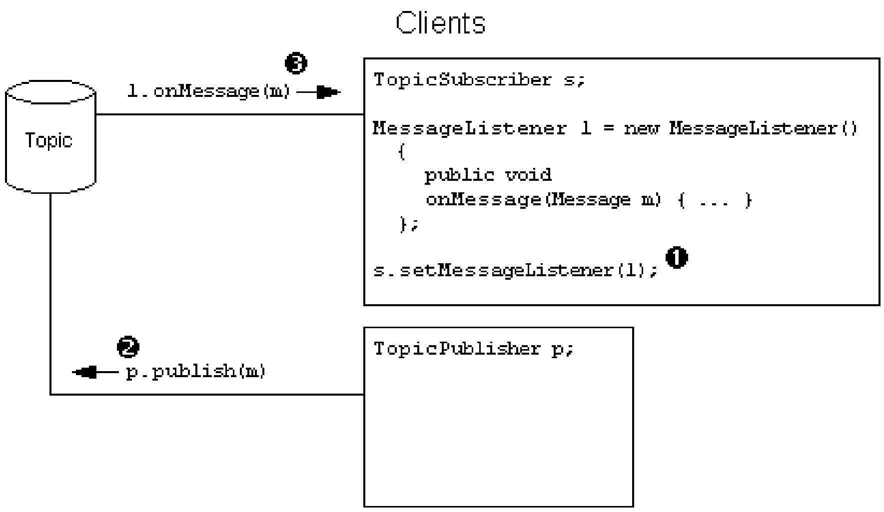

Figure 2. Diagram of publish/subscribe interaction

The characteristic pattern of interaction is as follows:

1. During the subscriber's initialization, the subscriber registers an instance of a class that implements the `MessageListener` interface by invoking `s.setMessageListener(l)` where `s` is the subscriber and `l` is the message listener. The topic keeps track of all of the subscribers subscribed to the topic.

2. The publishing client arrives at a point in its processing when it needs to send a message. It creates the message and publishes it by invoking `p.publish(m)`, where `p` is the publisher and `m` is the message.

3. Immediately, the topic delivers the message to the subscriber by invoking `l.onMessage(m)`. `l` is a reference to the message listener instance registered earlier.

Once all subscribers have been notified, the message is removed from the topic.

As you can see from the example, the publish/subscribe model provides asynchronous delivery of messages.

The code

Before launching into the code, I'd like to discuss a few architectural and design details. The implementation of the JMS API I'm about to present is built on Java's Remote Method Invocation

(RMI) package. RMI is used by clients to communicate with the queues and topics.

Since RMI provides a usable naming service (via the `rmiregistry` application), I've chosen to use that rather than JNDI. This particular choice shouldn't result in any functional deficiencies.

Also, as I mentioned earlier, I have made no heroic effort to implement the entire JMS API specification, nor have I implemented completely all the classes. I'll let you know where my implementation comes up short.

The message

As central as messages are to messaging, there is no such thing as a common message format supported by all vendors of messaging products. The JMS API specification (Sun claims, correctly, that the JMS API is the first messaging API to gain wide industry support) solves this problem by defining five types of messages. It is up to JMS API implementation providers to define the mapping between the message types in the specification and the types of messages provided natively.

The five types of messages are:

1. The bytes message -- consisting of a stream of uninterpreted bytes
2. The map message -- consisting of a set of name/value pairs
3. The stream message -- consisting of a stream of Java primitive types
4. The text message -- consisting of a stream of text
5. The object message -- consisting of a serialized Java object

For the sake of space, I've provided an implementation of only one of these five message types -- the text message. Here's the source code for both it and its supertype `Message`:

```
abstract
public
class Message
implements Serializable
{
  // Message headers contain identification and routing information.
  // I've implemented only the Destination and ReplyTo headers.

  private Destination _destinationDestination = null;
  private Destination _destinationReplyTo = null;

  protected
  void
  setDestination(Destination destinationDestination)
  {
    if (destinationDestination == null)
    {
      throw new NullPointerException("null Destination");
    }

    _destinationDestination = destinationDestination;
  }
```

```java
  protected
  void
  setReplyTo(Destination destinationReplyTo)
  {
    if (destinationReplyTo == null)
    {
      throw new NullPointerException("null ReplyTo");
    }

    _destinationReplyTo = destinationReplyTo;
  }

  public
  Destination
  getDestination()
  {
    return _destinationDestination;
  }

  public
  Destination
  getReplyTo()
  {
    return _destinationReplyTo;
  }

  // Properties are basically application specific headers.
  // I’ve implemented only Object properties.

  private Hashtable _hashtableProperties = null;

  public
  void
  setObjectProperty(String stringName, Object objectValue)
  {
    if (stringName == null)
    {
      throw new NullPointerException("null name");
    }

    if (objectValue == null)
    {
      throw new NullPointerException("null value");
    }

    if (_hashtableProperties == null)
    {
      _hashtableProperties = new Hashtable();
    }

    _hashtableProperties.put(stringName, objectValue);
  }

  public
  Object
  getObjectProperty(String stringName)
  {
    return _hashtableProperties.get(stringName);
  }
}
```

```
public
class Message_Text
extends Message
{
  // The _stringText variable holds the body of the message.

  private String _stringText = null;

  public
  Message_Text(String stringText)
  {
    _stringText = stringText;
  }

  public
  void
  setText(String stringText)
  {
    _stringText = stringText;
  }

  public
  String
  getText()
  {
    return _stringText;
  }

  public
  String
  toString()
  {
    return "Message_Text: " + _stringText;
  }
}
```

The queue

The destination of messages in the point-to-point model is the *queue.* In the JMS API specification, clients don't interact with the queue directly. Instead, they interact through instances of the `QueueSender` and `QueueReceiver` interface classes. Next month, we'll design these classes. This month, I'll present the machinery behind the queue itself -- that is, I'll show how queues work on the inside. The following code implements a simple queue:

```
public
interface HQueue
extends Remote
{
   // A queue provides two methods.  One for adding messages
   // and one for removing messages.  I haven't provided any
   // provision for message selectors.

   public
   void
   send(Message message)
   throws RemoteException;

   public
   Message
   receive()
```

```
   throws RemoteException;
}

public
class HHQueue
extends UnicastRemoteObject
implements HQueue
{
   // Messages are held in a simple linked-list data structure.

   private LinkedList _linkedlistMessages = new LinkedList();

   // Add a message to the queue and notify all waiting threads.

   public
   void
   send(Message message)
   throws RemoteException
   {
      synchronized (_linkedlistMessages)
      {
         _linkedlistMessages.add(message);

         _linkedlistMessages.notifyAll();
      }
   }

   // Remove a message from the queue.  Block if no messages
   // are present in the queue.

   public
   Message
   receive()
   throws RemoteException
   {
      Message message = null;

      synchronized (_linkedlistMessages)
      {
         while ((message = (Message)_linkedlistMessages.remove()) == null)
         {
            try
            {
               _linkedlistMessages.wait();
            }
            catch (InterruptedException exception)
            {
               break;
            }
         }
      }

      return message;
   }

   // Start a queue and register it with the RMI registry.

   public
   static
   void
   main(String [] rgstring)
```

```
   {
      try
      {
         Naming.rebind(rgstring[0], new HHQueue());
      }
      catch (Exception exception)
      {
         System.out.println(exception);
      }
   }
}
```

The topic

The destination of messages in the publish/subscribe model is the *topic*. In the JMS API specification, clients don't interact with the topic directly. Instead, they interact through instances of the `TopicSubscriber` and `TopicPublisher` interface classes. Once again, we'll design these classes next month, but this month I'll show you how topics work on the inside.

The following code implements a simple topic:

```
public
interface HTopic
extends Remote
{
   // A topic provides two methods.  One for subscribing to the
   // topic and one for publishing messages to the topic.  I
   // haven't provided any provision for durable topic
   // subscribers.

   public
   void
   setMessageListener(MessageListener messagelistener)
   throws RemoteException;

   public
   void
   publish(Message message)
   throws RemoteException;
}

public
class HHTopic
extends UnicastRemoteObject
implements HTopic
{
   // Subscribers are held in a simple linked-list data structure.

   private LinkedList _linkedlistSubscribers = new LinkedList();

   // Subscribe to this topic.  This is accomplished by registering
   // a message listener.

   public
   void
   setMessageListener(MessageListener messagelistener)
   throws RemoteException
   {
      synchronized (_linkedlistSubscribers)
```

```
      {
         _linkedlistSubscribers.add(messagelistener);
      }
   }

   // Deliver a message to all of the subscribers subscribed to a this
   // topic.

   public
   void
   publish(Message message)
   throws RemoteException
   {
      synchronized (_linkedlistSubscribers)
      {
         Enumeration enumeration = _linkedlistSubscribers.elements();

         while (enumeration.hasMoreElements())
         {
            MessageListener messagelistener = (MessageListener)enumeration.nextEleme

            messagelistener.onMessage(message);
         }
      }
   }

   // Start a topic and register it with the RMI registry.

   public
   static
   void
   main(String [] rgstring)
   {
      try
      {
         Naming.rebind(rgstring[0], new HHTopic());
      }
      catch (Exception exception)
      {
         System.out.println(exception);
      }
   }
}
```

If you'd like, you can download either a zip file or a gzipped tar file containing the source files, the compiled class files, and several utilities for manipulating queues and topics.

Conclusion

From these three classes (and a couple of helpers such as a linked list) you can build a basic messaging service. In fact, as they are, these classes actually a provide a functional message service, but one that isn't compliant with the JMS API specification. Next month, I'll bring them into compliance by adding the required JMS veneer. See you then. ■

Also this month in JavaWorld
Go

Resources

- Java Message Service home page
  http://www.javasoft.com/products/jms/
- JMS Specification
  http://www.javasoft.com/products/jms/docs.html
- Michael Shoffner's article on message-oriented middleware (MOMs)
  http://www.javaworld.com/javaworld/jw-05-1998/jw-05-step.html
- Read Todd's previous **How-To Java** columns
  http://www.javaworld.com/topicalindex/jw-ti-howto.html

About the author

Todd Sundsted has been working with computers since computers became available in convenient desktop models. Though originally interested in building distributed object applications in C++, Todd moved on to the Java programming language when it became the obvious choice for that sort of thing. In addition to writing, Todd is President of Etcee which offers training, mentoring, consulting, and software development services. Reach Todd at todd.sundsted@javaworld.com.