**JMS e delivery Garantita:**

***The notion of guaranteed delivery of Java Message Service messages has been lightly touched on in other recently published articles on JMS. But what really makes a JMS message "guaranteed"? Should you just take it on faith, or would you like to know what's behind it?***

This article answers these questions via a detailed discussion of message persistence, internal acknowledgment rules, and message redelivery. Using excerpts condensed from the book we coauthored, *Java Message Service*, we'll explain how JMS guaranteed messaging works - including once-and-only-once delivery semantics, durable subscriptions, failure and recovery scenarios, and transacted messages.

**JMS Guaranteed Messaging**   
  
There are three key parts to guaranteed messaging:   
1) message autonomy,   
2) store-and-forward, and   
3) the underlying message acknowledgment semantics.   
  
  
**Client runtime:**

Before we discuss these parts, we need to review and define some new terms.   
  
A JMS client application uses the JMS API.   
  
Each JMS vendor provides an implementation of the JMS API on the client, which we call the *client runtime*.

**Jms Provider:**

In addition to the client runtime, the JMS vendor provides some kind of message "server" that implements the routing and delivery of messages. The client runtime and the message server are collectively referred to as the *JMS provider*.

**"Provider failure"**

refers to any failure condition that is outside the domain of the application code.   
It could mean a hardware failure that occurs while the provider is entrusted with the processing of a message, an unexpected exception, the abnormal end of a process due to a software defect, or network failures.

**Message Autonomy**

Messages are self-contained autonomous entities.   
A message may be sent and re-sent many times across multiple processes throughout its lifetime. Each JMS client along the way will consume the message, examine it, execute business logic, modify it, or create new messages in order to accomplish the task at hand.

**Message lifecycle:**

Once a JMS client sends a message, its role is completed.   
The JMS provider guarantees that any other interested parties will receive the message.   
This contract between a sending JMS client and the JMS provider is much like the contract between a JDBC client and a database. Once the data is delivered, it is considered "safe" and out of the hands of the client.

**Store-and-Forward Messaging**   
When messages are marked persistent, it is the responsibility of the JMS provider to utilize a store-and-forward mechanism to fulfill its contract with the sender.   
The storage mechanism is used for persisting messages to disk to ensure that the message can be recovered in the event of a provider failure or a failure of the consuming client. The forwarding mechanism is responsible for retrieving messages from storage, and subsequently routing and delivering them.

**Message Acknowledgments**   
The message acknowledgment protocol is key to guaranteed messaging, and support for acknowledgment is required by the semantics of the JMS API. The acknowledgment protocol allows the JMS provider to monitor the progress of a message so that it knows whether the message was successfully produced and consumed. With this knowledge the JMS provider can manage the distribution of messages and guarantee their delivery.

The acknowledgment mode is set on a JMS session when it is created, as indicated below:

tSession = tConnect.createTopicSession(false, Session.CLIENT\_ACKNOWLEDGE);

qSession = qConnect.createQueueSession(false, Session.DUPS\_OK\_ACKNOWLEDGE);

Session bean state is not transactional.

**AUTO\_ACKNOWLEDGE**

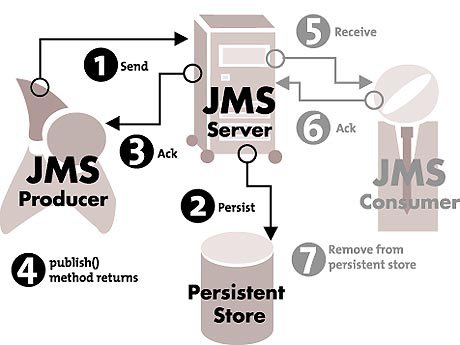
Let's start by examining the AUTO\_ACKNOWLEDGE mode. In the loosely coupled asynchronous environment of JMS, senders and receivers (producers and consumers) are intentionally decoupled from each other.   
  
Hence, the roles and responsibilities are divided among the   
1) message producer,   
2) the message server, and   
3) the message consumer.

***The Producer's Perspective***

Under the covers, the TopicPublisher.publish() or Queue- Sender.send() methods are synchronous.   
  
These methods are responsible for sending the message and blocking until an acknowledgment is received from the message server.   
  
If a failure condition occurs during this operation, an exception is thrown to the sending application and the message is considered undelivered.

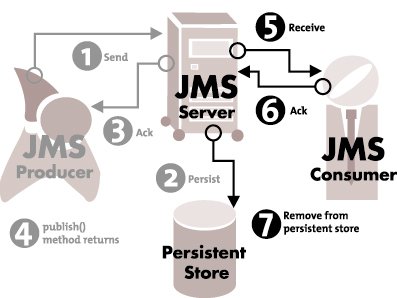
***The Server's Perspective***

The acknowledgment sent to the producer (sender) from the server means that the server has received the message and has accepted responsibility for delivering it.   
  
From the JMS server's perspective, the acknowledgment sent to the producer is not tied directly to the delivery of the message.   
  
They are logically two separate steps.   
  
For persistent messages, the server writes the message out to disk (the store part of store-and-forward), then acknowledges to the producer that the message was received (see Figure 1).   
  
For nonpersistent messages this means the server may acknowledge to the sender as soon as it has received the message and has the message in memory.   
  
If there are no subscribers for the message's topic, the message may be discarded.

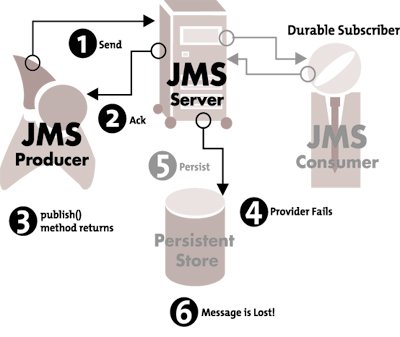
   
**Figure 1**

In a publish/subscribe model, the message server delivers a copy of a message to each of the subscribers. For durable subscriber, the message server doesn't consider a message fully delivered until it has received an acknowledgment from all of the message's intended recipients. It knows on a per-consumer basis which clients have received each message and which have not.

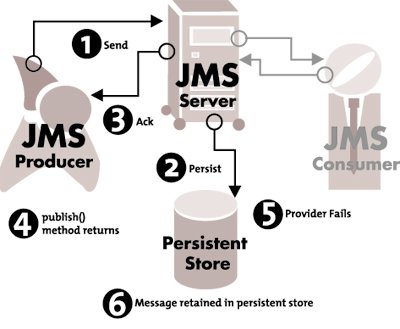
Once the message server has delivered the message to all of its known subscribers and has received acknowledgments from each of them, the message is removed from its persistent store (see Figure 2).

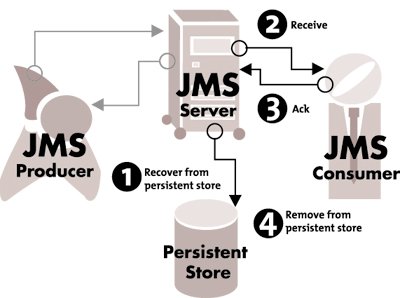
   
**Figure 2**

If the subscriptions are durable and the subscribers aren't currently connected, the message will be held by the message server until either the subscriber becomes available or the message expires.   
  
This is true even for nonpersistent messages.   
  
If a nonpersistent message is intended for a disconnected durable subscriber, the message server saves the message to disk as though it were a persistent message.   
  
In this case the difference between persistent and nonpersistent messages is subtle, but very important. For nonpersistent messages the JMS provider could fail before it's had a chance to **write the message out to disk on behalf of the disconnected durable subscribers**. Messages may be lost (see Figure 3).

   
**Figure 3**

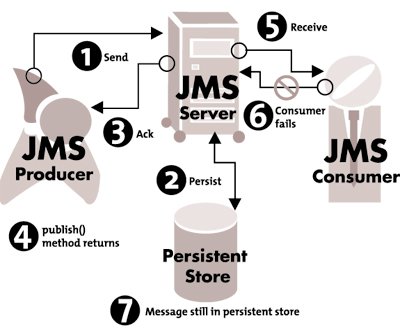
With persistent messages a provider may fail and recover gracefully, as illustrated in Figures 4 and 5. Since the messages are held in persistent storage, they're not lost and will be delivered to consumers when the provider starts up again. If the messages are sent using a p2p queue, they're guaranteed to be delivered. If the messages were sent via publish/subscribe, they're guaranteed to be delivered only if the consumers' subscriptions are durable.

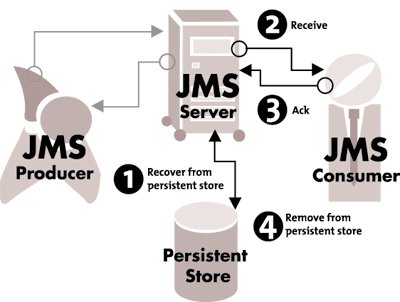
   
**Figure 4**

   
**Figure 5**

***The Consumer's Perspective***   
There are also rules governing acknowledgments and failure conditions from the consumer's perspective.  
  
If the session is in AUTO\_ACKNOWLEDGE mode, the JMS provider's client runtime must automatically send an acknowledgment to the server as each consumer gets the message.   
If the server doesn't receive this acknowledgment, it considers the message undelivered and may attempt redelivery.

***Message Redelivery***   
The message may be lost if the provider fails while delivering a message to a consumer with a nondurable subscription.   
  
If a durable subscriber receives a message, and a failure occurs before the acknowledgment is returned to the provider (see Figure 6), the JMS provider considers the message undelivered and will attempt to redeliver it (see Figure 7).   
  
In this case the "once-and-only-once" requirement is in doubt. The consumer may receive the message again, because when delivery is guaranteed, it's better to risk delivering a message twice than to risk losing it entirely. A redelivered message will have the JMSRedelivered flag set. A client application can check this flag by calling the getJMSRedelivered() method on the Message object. Only the most recent message received is subject to this ambiguity.

   
**Figure 6**

   
**Figure 7**

***Point-to-Point Queues***   
For point-to-point queues, messages are marked by the producer as either persistent or nonpersistent. If the former, they are written to disk and subject to the same acknowledgment rules, failure conditions, and recovery as persistent messages in the publish/subscribe model.

From the receiver's perspective the rules are somewhat simpler since only one consumer can receive a particular instance of a message.   
  
A message stays in a queue until it is delivered to a consumer or expires. This is somewhat analogous to a durable subscriber in that a receiver can be disconnected while the message is being produced without losing the message. If the messages are nonpersistent they aren't guaranteed to survive a provider failure.

**DUPS\_OK\_ACKNOWLEDGE**   
Specifying the DUPS\_OK\_ACKNOWLEDGE mode on a session instructs the JMS provider that it is okay to send a message more than once to the same destination.   
  
This is different from the once-and-only-once or the at-most-once delivery semantics of AUTO\_ACKNOWLEDGE.   
  
The DUPS\_OK\_ACKNOWLEDGE delivery mode is based on the assumption that the processing necessary to ensure once-and-only-once delivery incurs extra overhead and hinders performance and throughput of messages at the provider level.   
An application that is tolerant of receiving duplicate messages can use the DUPS\_OK\_ACKNOWLEDGE mode to avoid incurring this overhead.

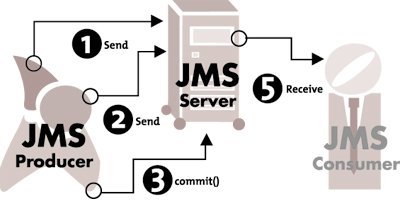
In practice, the performance improvement you gain from DUPS\_OK\_ACKNOWLEDGE may be something you want to measure before designing your application around it.

**CLIENT\_ACKNOWLEDGE**   
With AUTO\_ACKNOWLEDGE mode the acknowledgment is always the last thing to happen implicitly after the onMessage() handler returns.  
  
The client receiving the messages can get finer-grained control over the delivery of guaranteed messages by specifying the CLIENT\_ACKNOWLEDGE mode on the consuming session.

The use of CLIENT\_ACKNOWLEDGE allows the application to control when the acknowledgment is sent. For example, an application can acknowledge a message - thereby relieving the JMS provider of its duty - and perform further processing of the data represented by the message. The key to this is the acknowledge() method on the Message object, as shown in [Listing 1](http://www2.sys-con.com/itsg/virtualcd/Java/archives/0604/chappell/index.html#s1).

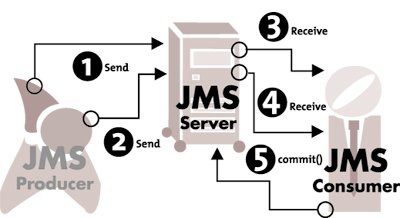
The acknowledge() method informs the JMS provider that the message has been successfully received by the consumer. This method throws an exception to the client if a provider failure occurs during the acknowledgment process. The provider failure results in the message being retained by the JMS server for redelivery.

**Transacted Messages**   
  
JMS transactions follow the convention of separating the send operations from the receive operations.   
Figure 8 shows a transactional send in which a group of messages are guaranteed to get to the message server, or none of them will.   
  
From the sender's perspective the messages are cached by the JMS provider until a commit() is issued. If a failure occurs or a rollback() is issued, the messages are discarded. Messages delivered to the message server in a transaction are not forwarded to the consumers until the producer commits the transaction.

   
**Figure 8**

The JMS provider won't start delivery of the messages to its consumers until the producer has issued a commit() on the session. The scope of a JMS transaction can include any number of messages.

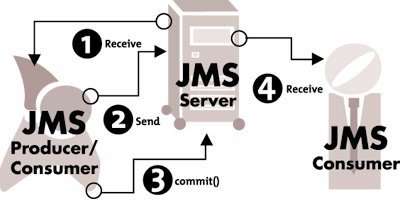
JMS also supports transactional receives, in which a group of transacted messages are received by the consumer on an all-or-nothing basis (see Figure 9).   
  
From the transacted receiver's perspective the messages are delivered to it as expeditiously as possible, yet they are held by the JMS provider until the receiver issues a commit() on the session object. If a failure occurs or a rollback() is issued, the provider will attempt to redeliver the messages, in which case the messages will have the redelivered flag set.

   
**Figure 9**

Transacted producers and transacted consumers can be grouped together in a single transaction if they are created from the same session object, as shown in Figure 10.   
  
This allows a JMS client to produce and consume messages as a single unit of work.

**Rollback transazione JMS:**

If the transaction is rolled back, the messages **produced** within the transaction won't be delivered by the JMS provider.   
  
The messages **consumed** within the same transaction won't be acknowledged and will be redelivered.

   
**Figure 10**

Unless you're doing a synchronous request/reply, you should avoid grouping a send followed by an asynchronous receive within a transaction. There could be a long interval between the time a message is sent and the related message is asynchronously received, depending on failures or downtime of other processes involved. It's more practical to group the receipt of a message followed by the send of another message.

**Creating a JMS Transaction**

Enabling a JMS transacted session happens as part of creating a Session object, as shown in [Listing 2](http://www2.sys-con.com/itsg/virtualcd/Java/archives/0604/chappell/index.html#s2).

The first parameter of createTopicSession() or createQueueSession() method is a Boolean indicating whether this is a transacted session. That is all we need to create a transactional session. There is no explicit begin() method.   
  
When a session is transacted, all messages sent or received using that session are automatically grouped in a transaction.   
  
The transaction remains open until either a session.rollback() or a session.commit() happens, at which point a new transaction is started.   
  
An additional Session method, isTransacted(), returns a Boolean true or false indicating whether the current session is transactional.

**Distributed Transactions**

Having all producers and all consumers participate in one global transaction would defeat the purpose of using a loosely coupled asynchronous messaging environment.

Sometimes it's necessary to coordinate the send or receipt of a JMS transaction with the update of another non-JMS resource, like a database or an EJB entity bean.   
  
This typically involves an underlying transaction manager that takes care of coordinating the prepare, commit, or rollback of each resource participating in the transaction.  
  
JMS provides JTA transaction interfaces for accomplishing this.

JMS providers that implement the JTA XA APIs can participate as a resource in a two-phase commit. The JMS specification provides XA versions of the following JMS objects : XAConnectionFactory, XAQueueConnection, XAQueueConnectionFactory, XAQueueSession, XASession, XATopicConnection, XATopicConnectionFactory, and XATopicSesion.

**Conclusion**

Our discussion of message acknowledgment shows that producers and consumers have different perspectives on the messages they exchange. The producer has a contract with the message server that ensures that the message will be delivered as far as the server. The server has a contract with the consumer that the message will be delivered to it. The two operations are separate, which is a key benefit of asynchronous messaging. It is the role of the JMS provider to ensure that messages get to where they are supposed to go.

Through message acknowledgment, message persistence, and transactions, JMS provides a strict set of rules that guarantees that your business critical data will travel reliably throughout your global enterprise. Note: This material is condensed from our book, which contains full working examples with detailed explanations of the concepts presented here.

**Author Bios**   
David Chappell is chief technology evangelist for Progress Software's SonicMQ and coauthor of O'Reilly's Java Message Service.   
[chappell@progress.com](mailto:chappell@progress.com)

Richard Monson-Haefel is the author of Enterprise JavaBeans and coauthor of Java Message Service, both from O'Reilly. He is also the lead architect of the OpenEJB server.   
[richard@monson-haefel.com](mailto:richard@monson-haefel.com)



**Listing 1**

public void onMessage(javax.jms.Message message){

int count = 1000;

try{

// perform some business logic with the message

...

message.acknowledge();

// perform more business logic with the message

...

}catch (javax.jms.JMSException jmse){

// catch the exception thrown and undo the results

// of partial processing

...

}

}

**Listing 2**

// pub/sub connection creates a transacted TopicSession

javax.jms.TopicSession session=connect.createTopicSession(true,Session.AUTO\_ACKNOWLEDGE);

// p2p connection creates a transacted QueueSession

javax.jms.QueueSession =

connect.createQueueSession(true,Session.AUTO\_ACKNOWLEDGE);

...

}