JMS

What is the difference between queue and topic?

**Queue vs Topic**

Java message service, or simply JMS, is a medium which sends messages to two or more clients. It permits contact between diverse mechanisms of a distributed application. This message-oriented middleware has two models which are the point-to-point model and publish or subscribe model. These two models have other names, too. The point-to-point model is also known as the queue model, and the publisher or subscriber model is also simply known as the topic model.

The queue or point-to-point model works by having a sender place messages to a queue, and the [receiver](http://www.differencebetween.net/technology/difference-between-amplifier-and-receiver/) will be able to read the messages from the queue. However, publisher or subscriber or the topic model works by disseminating messages by posting messages about a particular topic and having subscribers read them.

There are many differences between these two models which we will simply [call](http://www.differencebetween.net/business/finance-business-2/difference-between-call-and-put/) “queue” and “topic.” In queue, the sender knows where the message will be going. There is a specific sender and a specific receiver, and there is the intention of being acknowledged as such. On the other hand, in topic you only have a publisher and a subscriber or subscribers. There is anonymity in the identity of both the publisher and subscriber.

Another main difference between the two is the number of recipients. In queue, you only have one receiver or consumer; unlike in topic where in you can have your message be disseminated to a number of subscribers. Also, in topic, the publisher has to be continuously active for a subscriber to receive the messages. Otherwise the message will be reallocated. In queue you do not have to worry about timing because the sender will have the luxury to send messages whenever he or she wants to. And the same goes for the receiver; he or she also has the liberty of reading it whenever he or she wants. In queue you will also be assured that as the sender you have successfully sent out your message because you will be notified by the receiver, but the same is not true for a topic system. There is even the risk of not having any subscribers.

SUMMARY:

1.The point-to-point or queue model works by the sender to receiver setup. On the other hand, publisher/subscriber or topic model works by bulletin setup.

2.In the queue model there is acknowledgement of the identity of the receiver and oftentimes the sender. In the topic model there is anonymity in the identities of both the subscriber and publisher.

3.Queue model is only allowed one recipient; topic, on the other hand, can have multiple recipients.

4.In queue model, the sender and receiver do not have to be both active at the same time. In the topic model, timing is very vital.

5.In the queue model, the sender will receive a notification when the message gets to the receiver. The topic model, on the other hand, will not notify you with such, and there is even a risk that you will have no subscribers.

The high-level  interfaces are:

* ConnectionFactory: An administered object that creates a Connection.
* Connection: An active connection to a provider.
* Destination: An administered object that encapsulates the identity of a message destination, such as where messages are sent to or received from.
* Session: A single-threaded context for sending and receiving messages. For reasons of simplicity and because Sessions control transactions, concurrent access by multiple threads is restricted. Multiple Sessions can be used for multithreaded applications.
* MessageProducer: Used for sending messages.
* MessageConsumer: Used for receiving messages.

The following table identifies the domain-specific interfaces inherited from each high-level interface.

|  |  |  |
| --- | --- | --- |
| High-levelinterface | PTP domain | Pub/sub domain |
| ConnectionFactory | QueueConnectionFactory | TopicConnectionFactory |
|  |  |  |
| Connection | QueueConnection | TopicConnection |
|  |  |  |
| Destination | Queue | Topic |
|  |  |  |
| Session | QueueSession | TopicSession |
|  |  |  |
| MessageProducer | QueueSender | TopicPublisher |
|  |  |  |
| MessageConsumer | QueueReceiver, | TopicSubscriber |

A JMS application is composed of the following parts:

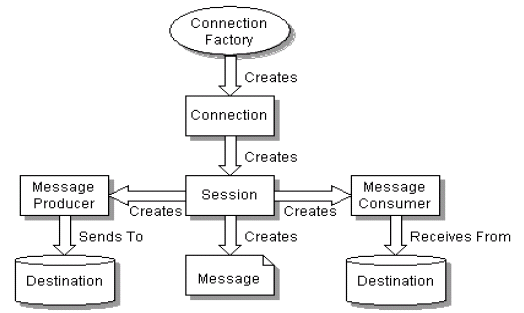
* A JMS provider: A messaging system that implements the JMS specification.
* JMS clients: Java applications that send and receive messages.
* Messages: Objects that are used to communicate information between JMS clients.
* Administered objects: Preconfigured JMS objects that are created by an administrator for the use of JMS clients.

The JMS specification defined six type or classes of messages that a JMS provider must support:

* Message: This represents a message without a message body.
* StreamMessage: A message whose body contains a stream of Java primitive types. It is written and read sequentially.
* MapMessage: A message whose body contains a set of name/value pairs. The order of entries is not defined.
* TextMessage: A message whose body contains a Java string…such as an XML message.
* ObjectMessage: A message whose body contains a serialized Java object.
* BytesMessage: A message whose body contains a stream of uninterpreted bytes.

### Producing and Consuming Messages

Here are the necessary steps to develop clients to produce and consumer messages. Note that there are some common steps that shouldn’t be duplicated if the client is both producing and consuming messages. Figure below depicts the high-level view of the steps

[](http://www.j2eebrain.com/wp-content/uploads/Producing-and-Consuming-Messages.gif)

\* JMSDeliveryMode —   type int

Contains the value DeliveryMode.PERSISTENT or

DeliveryMode.NON\_PERSISTENT. A persistent message is delivered “once and only once”; a non-persistent message is delivered “at most once.” Be aware that “at most once” includes not being delivered at all. A non-persistent message may be lost by a provider during application or system failure. Extra care will be taken to assure that a persistent message is not affected by failures. There is often considerable overhead in sending persistent messages, and the trade-offs between reliability and performance must be carefully considered when deciding the delivery mode of a message.

 JMSPriority —   type int

The priority of the message; set by the provider during the send process. A priority of 0 is the lowest priority; a priority of 9 is the highest priority

A typical JMS program goes through the following steps to begin producing and consuming messages.  
1. Look up a ConnectionFactory through JNDI.  
2. Look up one or more Destinations through JNDI.  
3. Use the ConnectionFactory to create a Connection.  
4. Use the Connection to create one or more Sessions.  
5. Use a Session and a Destination to create the required MessageProducers and MessageConsumers.  
6. Start the Connection.  
At this point, messages can begin to flow and the application can receive, process, and send messages, as required.

Transactions

A JMS transaction groups a set of produced messages and a set of consumed messages into an atomic unit of work. If an error occurs during a transaction, the production and consumption of messages that occurred before the error can be “undone.”

Session objects control transactions and a Session may be denoted as transacted when it is created. A transacted Session always has a current transaction, that is, there is no begin(); commit() and rollback() end one transaction and automatically begin another.

Distributed transactions may be supported by the Java Transaction API (JTA) XAResource API, though this is optional for providers.

Acknowledgement

Acknowledgement is the mechanism whereby a provider is informed that a message has been successfully received.

If the Session receiving the message is transacted, acknowledgement is handled automatically. If the Session is not transacted, then the type of acknowledgement is determined when the Session is created.

There are three types of acknowledgement:

\* Session.DUPS\_OK\_ACKNOWLEDGE: Lazy acknowledgement of message delivery; reduces overhead by minimizing work done to prevent duplicates; should only be used if duplicate messages are expected and can be handled.

\* Session.AUTO\_ACKNOWLEDGE: Message delivery is automatically acknowledged upon completion of the method that receives the message.

\* Session.CLIENT\_ACKNOWLEDGE: Message delivery is explicitly acknowledged by calling the acknowledge() method on the Message.

#### What are the types of messaging?

There are two kinds of Messaging.  
Synchronous Messaging: Synchronous messaging involves a client that waits for the server to respond to a message.  
Asynchronous Messaging: Asynchronous messaging involves a client that does not wait for a message from the server. An event is used to trigger a message from a server.

#### What are the advantages of JMS?

JMS is asynchronous in nature. Thus not all the pieces need to be up all the time for the application to function as a whole. Even if the receiver is down the MOM

will store the messages on it’s behalf and will send them once it comes back up. Thus at least a part of application can still function as there is no blocking.

#### What is the basic difference between Publish Subscribe model and P2P model?

Publish Subscribe model is typically used in one-to-many situation. It is unreliable but very fast. P2P model is used in one-to-one situation. It is highly reliable.

What is the use of BytesMessage?

BytesMessage contains an array of primitive bytes in it’s payload. Thus it can be used for transfer of data between two applications in their native format which may

not be compatible with other Message types. It is also useful where JMS is used purely as a transport between two systems and the message payload is opaque to the JMS

client. Whenever you store any primitive type, it is converted into it’s byte representation and then stored in the payload. There is no boundary line between the

different data types stored. Thus you can even read a long as short. This would result in erroneous data and hence it is advisable that the payload be read in the same

order and using the same type in which it was created by the sender.

What is the use of StreamMessage?

StreamMessage carries a stream of Java primitive types as it’s payload. It contains some conveient methods for reading the data stored in the payload. However

StreamMessage prevents reading a long value as short, something that is allwed in case of BytesMessage. This is so because the StreamMessage also writes the type

information alonwgith the value of the primitive type and enforces a set of strict conversion rules which actually prevents reading of one primitive type as another.

What is the use of TextMessage?

TextMessage contains instance of java.lang.String as it’s payload. Thus it is very useful for exchanging textual data. It can also be used for exchanging complex

character data such as an XML document.

What is the use of ObjectMessage?

ObjectMessage contains a Serializable java object as it’s payload. Thus it allows exchange of Java objects between applications. This in itself mandates that both the

applications be Java applications. The consumer of the message must typecast the object received to it’s appropriate type. Thus the consumer should before hand know

the actual type of the object sent by the sender. Wrong type casting would result in ClassCastException. Moreover the class definition of the object set in the payload

should be available on both the machine, the sender as well as the consumer. If the class definition is not available in the consumer machine, an attempt to type cast

would result in ClassNotFoundException. Some of the MOMs might support dynamic loading of the desired class over the network, but the JMS specification does not

mandate this behavior and would be a value added service if provided by your vendor. And relying on any such vendor specific functionality would hamper the portability

of your application. Most of the time the class need to be put in the classpath of both, the sender and the consumer, manually by the developer.

What is the use of MapMessage?

A MapMessage carries name-value pair as it’s payload. Thus it’s payload is similar to the java.util.Properties object of Java. The values can be Java primitives or

their wrappers.

What is the difference between BytesMessage and StreamMessage?

BytesMessage stores the primitive data types by converting them to their byte representation. Thus the message is one contiguous stream of bytes. While the

StreamMessage maintains a boundary between the different data types stored because it also stores the type information along with the value of the primitive being

stored. BytesMessage allows data to be read using any type. Thus even if your payload contains a long value, you can invoke a method to read a short and it will return

you something. It will not give you a semantically correct data but the call will succeed in reading the first two bytes of data. This is strictly prohibited in the

StreamMessage. It maintains the type information of the data being stored and enforces strict conversion rules on the data being read.

#### What is point-to-point messaging?

With point-to-point message passing the sending application/client establishes a named message queue in the JMS broker/server and sends messages to this queue. The

Receiving client registers with the broker to receive messages posted to this queue. There is a one-to-one relationship between the sending and receiving clients.

#### What is the advantage of persistent message delivery compared to nonpersistent delivery?

If the JMS server experiences a failure, for example, a power outage, any message that it is holding in primary storage potentially could be lost. With persistent storage, the JMS server logs every message to secondary storage. (The logging occurs on the front end, that is, as part of handling the send operation from the message producing client.) The logged message is removed from secondary storage only after it has been successfully delivered to all consuming clients.

**RabbitMQ is one of the leading implementation of the AMQP protocol (along with Apache Qpid). Therefore, it implements a broker architecture, meaning that messages are queued on a central node before being sent to clients. This approach makes RabbitMQ very easy to use and deploy, because advanced scenarios like routing, load balancing or persistent message queuing are supported in just a few lines of code. However, it also makes it less scalable and “slower” because the central node adds latency and message envelopes are quite big.  
  
ZeroMQ is a very lightweight messaging system specially designed for high throughput/low latency scenarios like the one you can find in the financial world. Zmq supports many advanced messaging scenarios but contrary to RabbitMQ, you’ll have to implement most of them yourself by combining various pieces of the framework (e.g : sockets and devices). Zmq is very flexible but you’ll have to study the 80 pages or so of the guide (which I recommend reading for anybody writing distributed system, even if you don’t use Zmq) before being able to do anything more complicated that sending messages between 2 peers.  
  
ActiveMQ is in the middle ground. Like Zmq, it can be deployed with both broker and P2P topologies. Like RabbitMQ, it’s easier to implement advanced scenarios but usually at the cost of raw performance. It’s the Swiss army knife of messaging :-).  
  
Finally, all 3 products:  
   •   Have client APIs for the most common languages (C++, Java, .Net, Python, Php, Ruby, …)  
   •   Have strong documentation  
   •   Are actively supported**