1 Java Messaging Service (JMS)

1.1 Many to many communication

Whereas with RPC-like technologies (RMI, CORBA), there is a synchronous communication between a client and a server object, Java Messaging Service allows adding a mediation layer between one or more consumers and one or more producers. In RPC technologies, communicating between many to many can only be done using a mesh of one to one communication as in Fig. Xxx:

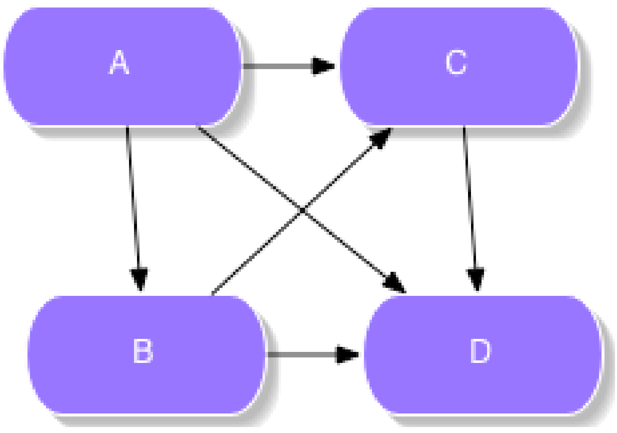


Fig 1

The JMS mediation layer allows many-to-many communication. Beware as all programs are called “JMS clients”, with respect to the JMS Server. Clients are all connected to the server, not to each other. The communication will follow two connection segments: one from the sender to the server, the other from the server to the receiver as in Fig. 2:

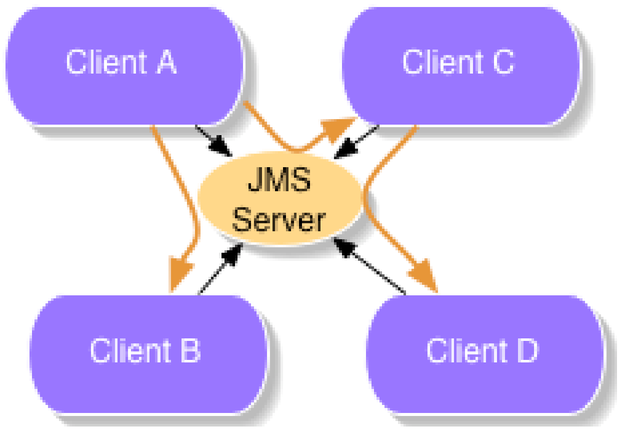


Fig 2

1.2 The consumer/producer model

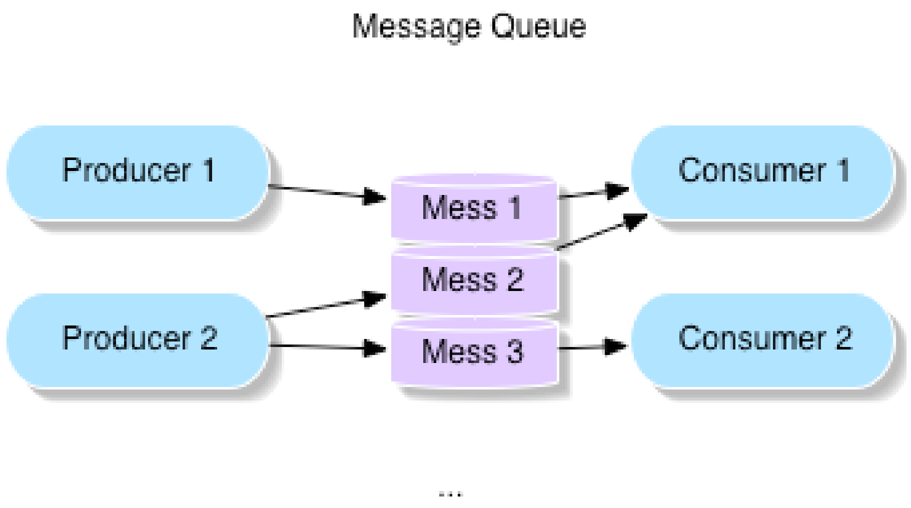
JMS comes from the unification of 2 communication approaches:

1) Communication through messages, or **point-to-point communication**, where a producer puts some messages in a storage box (called a Queue), to be retrieved later by a consumer,
2) Communication through events, or **publish/subscribe**, where all consumers listen to a channel (called a Topic) and get the same message when it is produced.

The server usually manages Queues and Topics, using some storage implementation.

The JMS API is defined in J2EE only, not in J2SE. JMS 1.1 has unified both approaches under common classes.

The generic term « Destination » refers both to Queues and Topics. Consumers and producers don't know each other, and only share the name of a Destination. In RPC models such as RMI, SOAP and Corba, they share both an interface (used for marshalling) and an address in a name service.



Fi g3 Point-to point : the destination is a Message Queue

In a point-to-point communication, only one consumer consumes one message. Generally, many producers and consumers may share the same destination, but at a single message level, the communication is only one to one.

The message may be stored for some time, thus the consumer may retrieve the message some time later without any correlation with the producer activity. In this sense, JMS is sometimes called *asynchronous*.

The message is deleted from the storage with some service-level acknowledgement from the consumer.

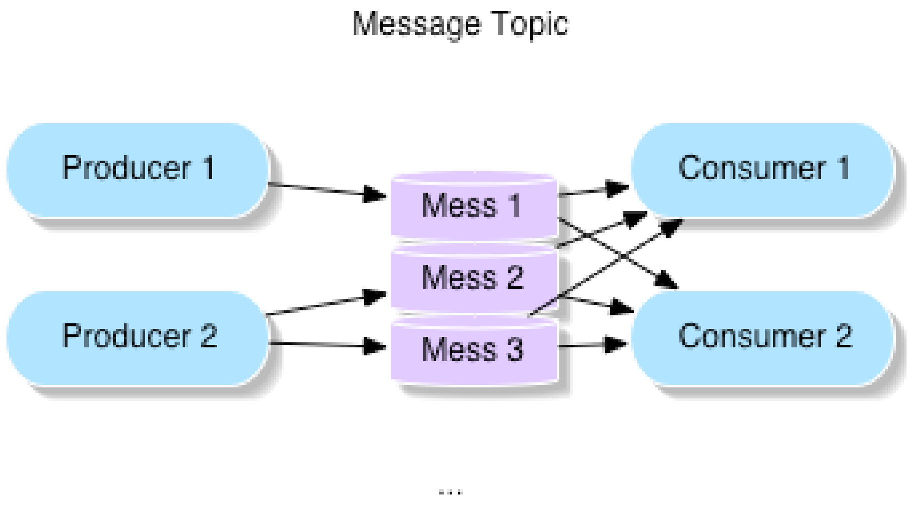


Fig 2 Publish /subscribe

In a publish/subscribe context consumers listen to the same topic, and collectively receive every message the producer publishes.

There is no way to decide when the message has been read by all consumers, and to decide when to delete it from storage: the message is thus simply deleted at the end of the session.

1.3 Concepts of a connection and a session

JMS introduces two key concepts, which do not exist in RMI and Corba : *Connection* and *Session*.

1.3.1 Connection helps reliability by defining a handshake protocol

Using network sockets, a TCP protocol *connection* means that a client and server need to exchange a handshake before being able to communicate: the connection is stopped either by program or when either of the two programs hangs up. A program can choose to close a connection. Both partners can detect the other has gone, and in this sense the TCP protocol is considered as safer than a protocol without connection.

In UDP protocol, there is no connection between clients and a server, and there is no way to detect when any of the partners has hung up. There is no way to ensure a client has received a packet.

In JMS, a client can be either a producer or a consumer, or both. A client connects to the Message Server, in the same sense a TCP client connects to a server. This ensures reliability between the client and the JMS server.

Some connection exceptions may be trapped if anything goes wrong in the process. Many server implementations provide a *heartbeat*: this is some a to regularly poll each partner across the network and anticipate any error in the communication.

In RMI or Corba, you cannot detect whether the server has hung up until you send it a method and retrieve either an exception or a timeout in the method answer. Of course, creating a connection handshake in JMS makes the communication no longer transparent, in contrast to RMI.

Connections are also used in JDBC (Java Database Connectivity) when connecting to databases, and in JCA (Java Connector Architecture): JCA provides a standard architecture to connect to any existing enterprise servers, such as legacy databases, ERPs (Enterprise Resource Planning) and mainframes (IBM CICS, etc.).

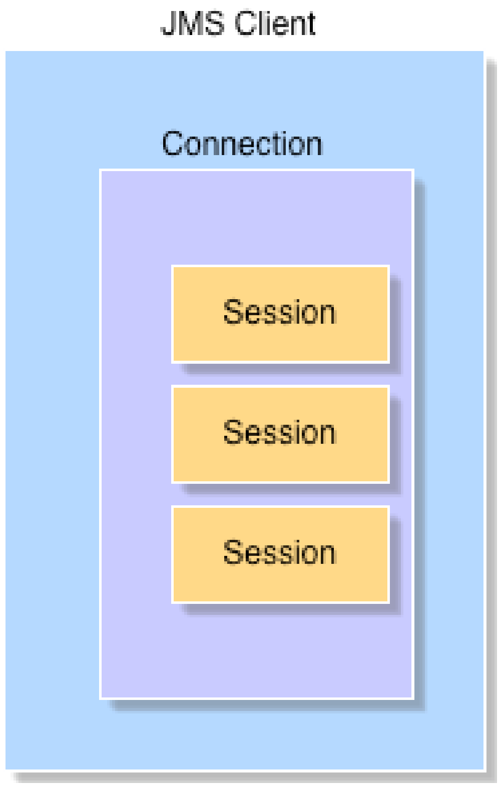


Fig 4

1.3.2 Session ensures higher Quality Of Service

Another concept used in JMS is a **Session**. A Session is defined over a connection and provides an abstraction to define a time interval with some specific properties: whether there is a transaction, what is the message sequence, etc. Two consecutive sessions may define different quality of service levels.

It is interesting to note that the new JCA specification issued in May 2005 does not have the concept of a Session. Instead it introduces as many contracts as non-functional aspects:

- A Connection Management Contract
- A Transaction Management Contract
- A Security Management Contract

In JMS, the Session handles transactions, but not security, which is handled at the connection level. Security has several aspects:

    - Encryption that can use SSL for JMS connections
    - Authentication using a login and password

The Session also provides a serial order between messages, handles priorities and acknowledges messages.

1.4 JMS

JMS (Java Messaging Service) is a Java API part of J2EE, not J2SE. In order to run JMS, you need a Message Server implementation.

There are a lot of implementations, including commercial implementations:

- Websphere MQ
- Swift MQ
- Sonic MQ
- Fiorano

And open source implementations:

- OpenJMS
- ActiveMQ
- Joram
- Mantaray

An updated list can be found on my web site: http://www.afceurope.com/JMS.html.

1.4.1 Sample code for sending and receiving a message in JMS

Sending a message in JMS requires two things:

- A reference to the message server: here tcp://localhost:2506, port 2506 on the local machine
- A name for a destination: here the Queue named MyQueue

```
// only one package is needed  for JMS
import javax.jms.* ;

          try {
// create a connection : this is specific to the JMS server
//implementation – here is one for SonicMQ
String broker= "tcp://localhost:2506";
     ConnectionFactory qcf= (ConnectionFactory) new
progress.message.jclient.ConnectionFactory (broker);
// the connection requires a login, here User and Password
     Connection qc= qcf.createConnection("User", "Password");

// After the connection is created, a session is needed
// Here the session is not transacted (false)
Session s = qc.createSession(false, Session.AUTO_ACKNOWLEDGE);

// The Queue is usually a managed object: it has to be created in the
// administrative interface of the JMS server before use

              Queue queue = s.createQueue("MyQueue");
              MessageProducer sender = s.createProducer(queue);
              TextMessage mess= s.createTextMessage("Hello");
              for (int i = 0; i < 10; i ++) sender.send( mess);
              System.out.println("Message sent OK");
              qc.close();
         } catch (JMSException e) {
              e.printStackTrace();
         }
    }
```

Notice the code is exactly the same for a Topic. The only difference is :

```
          Topic topic = s.createTopic(“MyTopic”);
```

This is also true in the receiving code below.

1.4.2 Receiving some code in JMS

```
// Create a connection: this is specific to the JMS server
//implementation – here is one for SonicMQ
```

```
	String broker= "tcp://localhost:2506";
	ConnectionFactory qcf= (ConnectionFactory) new 
progress.message.jclient.ConnectionFactory (broker);
	Connection qc= qcf.createConnection("User","Password");
	Session s = qc.createSession(false, Session.AUTO_ACKNOWLEDGE);
	Queue queue = s.createQueue("MyQueue");
	MessageConsumer receiver = s.createConsumer(queue);
// Remember to start the connection to be able to receive messages
	qc.start();
// receive 10 messages and assume they are TextMessages
	for (int i = 0; i < 1; i ++) {
TextMessage mess= (TextMessage) receiver.receive();
	    System.out.println("Received message : "+mess.getText());}
	qc.close();
		} catch (JMSException e) {
			e.printStackTrace();
		}
```

Important:

- Remember to **start** the connection; otherwise the consumer will never receive the message
- Remember to close the connection, in order to save resources. The program will not end if the connection is not closed. Some implementations require closing the session properly as well.

The receive command blocks; a consumer will block waiting forever if there is no message in the Queue. There are some alternatives to this:

- `ReceiveNoWait():` if there is a message in the Queue, retrieve it, otherwise simply return null
- `Receive(long timeout):` wait for a message some timeout, otherwise simply return

1.5 Chain of creation – more on Sessions and Connections

1.5.1 Overview

There is a perfect parallel in the creation session for the sending and receiving code. Notice the following creation chain:

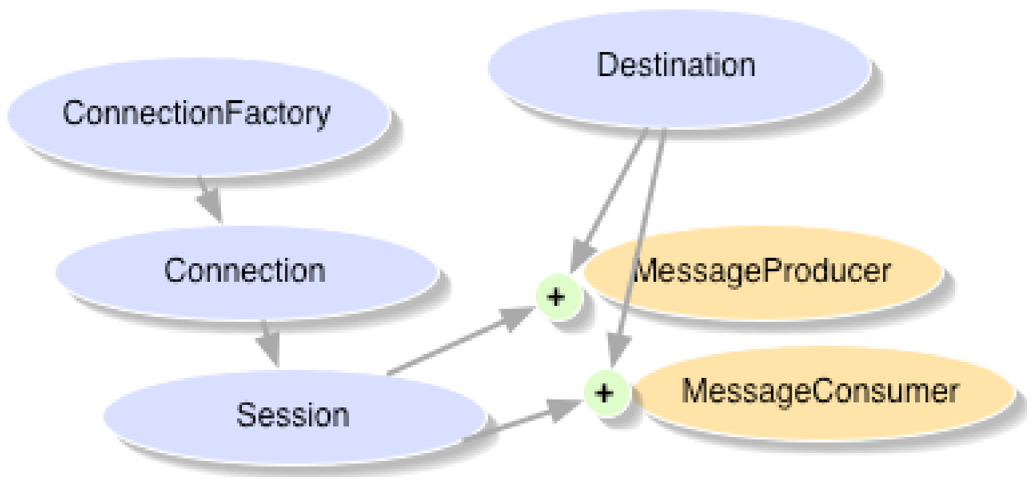


Fig 5

The Connection is created by a ConnectionFactory and then creates itself one or more Sessions.

A MessageProducer or a MessageConsumer is always created in a Session, but is attached to the name of a destination. Of course, a producer or a consumer can receive multiple messages. A Session can also create many producers and consumers to build a chain of communication and send and receive messages in the same JMS client.

1.5.2 Why would you use several Connections?

A Connection is linked to a machine *host and port*. Using several connections would enable the connection to multiple servers.

A Connection is also attached to a *login*. The administrative interface of the JMS server usually allows assigning rights to users on some Queues or Topics (reading, writing, etc.). Some administrative Queues can have restricted access. Using different logins requires using different connections.

1.5.3 Why use a ConnectionFactory ?

The ConnectionFactory name refers to the Factory pattern described in the Gamma book <*ref to be completed*>. A ConnectionFactory creates customized Connections, with some predefined initial parameters, like host, port or even login. It is usually stored in an enterprise directory like LDAP, and retrieved using JNDI (see chapter on JNDI).

<*to be completed : code to store and retrieve a ConnectionFactory in JNDI*>

A Destination can also be stored in JDNI. In this way, the two entry points needed by JMS can be accessed in a very standard way from all applications.

1.5.4 Why use several sessions?

The Session provides several capabilities: temporaryQueues, transactions, serial order of messages, synchronous/asynchronous reception — see below. For reception, a session can either be synchronous or asynchronous. A session can be transacted or not. <*to be completed*>

1.6 Asynchronous reception

With a traditional `receive` command a Consumer is blocked on reception from the JMS Server. In this sense, it is called “synchronous”. This term is slightly misleading, since JMS as a whole is also named as “asynchronous” (see above).

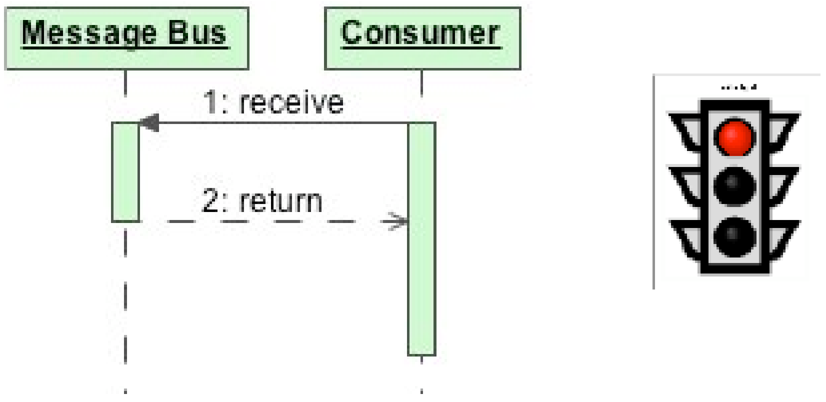


Fig 6

Waiting can lead to poor performance, or generate inter-blocking process issues.

JMS provides an API for an asynchronous reception using a `MessageListener`.

A MessageListener will spawn a new Thread to handle the incoming message, whenever it is available. The JMS Server manages this Thread internally.

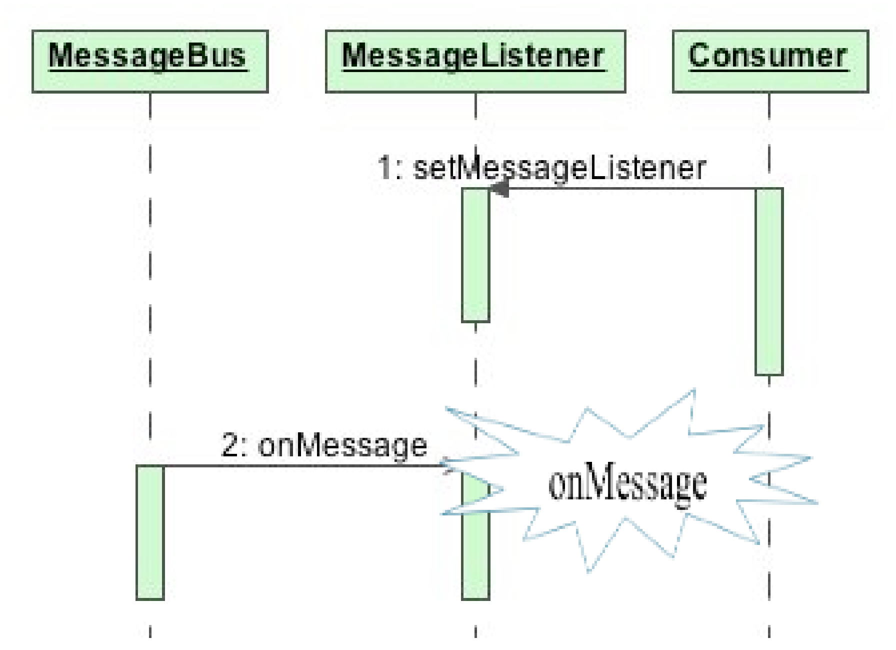


Fig 7

The sending code is unchanged.

For simplicity, the following code assumes that the class itself implements the `MessageListener` interface message `onMessage`().

The following code will check that the `onMessage` is triggered in a different Thread from the main one.

```
import java.io.*;
import javax.jms.*;
// The class needs to implements to MessageListener interface
public class Receiver implements MessageListener {
     static String broker= "tcp://maui:2506";

     public static void main(String[] args) {
          new Receiver().run();
     }
     private  void run()  {
          try {
// All this code is the same as above
                    ConnectionFactory qcf= (ConnectionFactory) new
progress.message.jclient.QueueConnectionFactory (broker);
                    Connection qc= qcf.createConnection();
                    Session s = qc.createSession(false,
Session.AUTO_ACKNOWLEDGE);
                    Queue queue = s.createQueue("MyQueue");
                    MessageConsumer receiver = s.createConsumer(queue);
// Now attach a listener to the consumer BEFORE starting the
// connection
                    receiver.setMessageListener(this);
// Do not start the connection until the Listener is attached
                    qc.start();
                    System.out.println("Waiting for a message");
// Java 5 allows printing a Thread Id, here the main Thread
               System.out.println("Thread id  "+ Thread.currentThread()
+ "  "+Thread.currentThread().getId());
               doSomething();


                    qc.close();
               } catch (JMSException e) {
                    e.printStackTrace();
               }
     }
     public void onMessage(Message message) {
          try {
// Cast the Message into a TextMessage, whose type should be tested
               System.out.println(((TextMessage) message).getText());
// Check onMessage is triggered in a different Thread
               System.out.println("Thread id onMessage "+
Thread.currentThread() + " "+Thread.currentThread().getId());
          } catch (JMSException e) {
               e.printStackTrace();
          }
     }
     private void doSomething() {
// This code avoids the program completing before receiving anything
          BufferedReader stdin =
          BufferedReader( new InputStreamReader(System.in ) );
          System.out.println ("Type end to exit");
           while ( true )
           {
                String word =null;
               try {
                    word = stdin.readLine();
               } catch (IOException e1) {
…
               }
```

```
                if (word.equals("end") )
                    System.exit(0);
            }
        }
}
```

1.7 Differences with JMS 1.0

JMS 1.0 necessitated making a distinction between Queue and Topic Connections. Two parallel hierarchies inherited from the same abstract root as illustrated below:

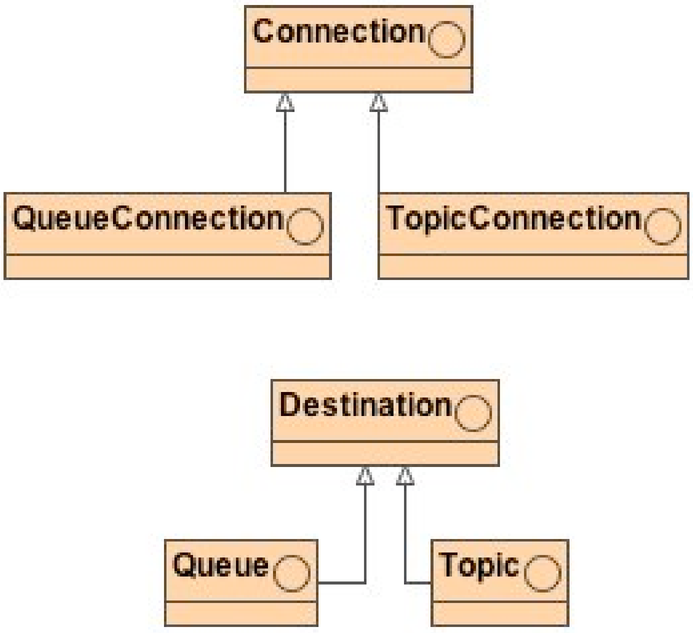


Fig 8

Table 1 : Differences in vocabulary from previous version

Abstract	**Point to point**	**Publish/subscribe**
Destination	Queue	Topic

Connection	QueueConnection	TopicConnection
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Session	QueueSession	TopicSession
MessageConsumer	QueueReceiver	TopicSubscriber
MessageProducer	QueueSender	TopicPublisher

In JMS 1.0, the code for subscribing to a Topic was as follows:

```
import javax.jms.*;

TopicConnectionFactory topicConnectionFactory =

// get some Factory, according to the implementation

                    …

TopicConnection topicConnection =

     topicConnectionFactory.createTopicConnection();

TopicSession topicSession = topicConnection.createTopicSession(false,

     Session.AUTO_ACKNOWLEDGE);

// A Topic for WebsphereMQ : it uses the implementation class MQTopic

Topic t = (Topic) new MQTopic();

          ((MQTopic) t).setBaseTopicName(name);

          ((MQTopic) t).setPersistence(1);

          ((MQTopic) t).setPriority(5);


topicSubscriber = topicSession.createSubscriber(topic);

          topicListener = new TextListener();

          topicSubscriber.setMessageListener(topicListener);
```

```
            topicConnection.start();
```

Notice that specific methods for Topics are required, e.g. `createTopicSession`(), which is unified in JMS 1.1 into a more generic `createSession`().

This code is fully described in "*Java Message Service*", Richard Monson-Haefel, David A. Chappell, O'Reilly, 2000.

1.8 Using hierarchies of Topics

<*To be completed*>

1.9 Durable subscribers and Topic persistence

<*to be completed*>

1.10 Marshalling in JMS

JMS messages basically include two parts:

- The header, used for quality of service and routing
- The body.

Quality of service will be described in paragraph …

Where the marshalling takes place is in the body (see Chapter … for marshalling).

JMS provides several types of messages, allowing various ways of marshalling:

- **A `TextMessage`** transports text messages, and may take advantage of the new Java IO character encoding properties. TextMessages can also transport XML content: this is how JMS can be considered as a SOAP transport protocol (see chapter …)

<to be completed : example of a TextMessage>

- **A `BytesMessage`** transports binary data. This is very convenient to transport images, programs, and files such as spreadsheets or documents, in a word everything used as attachment in traditional mail. In this sense, JMS can be viewed as a mail system between Java applications. BytesMessage can also transport any non-standard or homemade encoding such as ASN1. ASN1 is considered a much more concise marshalling support than XML and is especially used in the Telecom domain.

Abstract Syntax Notation One (**ASN.1**) is a formal language for abstractly describing messages to be exchanged among an extensive range of applications involving the Internet, intelligent network, cellular phones, ground-to-air communications, electronic commerce, secure electronic services, interactive television, intelligent transportation systems, Voice Over IP and others. Due to its streamlined encoding rules, ASN.1 is also reliable and ideal for wireless broadband and other resource-constrained environments. Its extensibility facilitates communications between newer and older versions of applications. In a world of change, ASN.1 is core technology, constantly adapting to new technologies.

Example: sending a file over JMS using a BytesMessage

```
// create a Session as usual
        BytesMessage outMessage = session.createBytesMessage();
        System.out.println("Adding Bytes");
        //<to be completed>
```

```
      //readFile to be rewritten with java.nio.FileChannel
      byte[] buffer=readFile("photo.jpg");
      outMessage.writeBytes(buffer);
// to transport the message size, put it as a property of the message
       outMessage.setIntProperty("size",buffer.length);
```

Receiving a file over JMS: <to be rewritten using java.nio Channels>

```
import javax.jms.*;
import java.io.*;
// create a Session as usual, as well as a message consumer
// wait only for 1000 ms
          Message inMessage = consumer.receive();
      // Check to see if the receive call has actually returned a
      // message. If it hasn't, report this and throw an exception...
          if (inMessage == null) {
               System.out.println(
                    "The attempt to read the message back again "
                    + "failed, apparently because it wasn't there");
               throw new JMSException("Failed to get message back
again");
               }
               // ...otherwise display the message
               System.out.println("Got message ");
      // Check that the message received is of the correct type
```

```
                    if (inMessage instanceof BytesMessage) {
                         // Extract the content size from the property
                         int size = inMessage.getIntProperty("size");
                         if (size > 0){
// allocate a buffer to store the file contents
                         byte[] replyBytes = new byte[size];
                         ((BytesMessage) inMessage).readBytes(replyBytes);
                         FileOutputStream outFile =
                              new FileOutputStream("photo_copy.jpg"););
                         outFile.write(replyBytes);
                         outFile.close();}}
```

- A **`MapMessage`** object is used to send a set of name-value pairs: this is typically what you would use in connection to a relational database, since each MapMessage type (int, float, etc.) has more a less a JDBC (Java Data Base Connectivity) counterpart.

```
// This message transports a detailed order item, with quantity and //
type of item
     MapMessage message = session.createMapMessage();
     message.setInt("Quantity", 4);
     message.setStringProperty("itemType", "Screen");
```

- A **`StreamMessage`** object is used to send a stream of primitive types in Java. It is filled and read sequentially. It inherits from the `Message` interface and adds a stream message body. Its methods are based largely on those found in `java.io.DataInputStream` and `java.io.DataOutputStream`.

- An **`ObjectMessage`** object is used to send a message that contains a `Serializable` object in Java. It inherits from the `Message` interface and adds a body containing a single reference to an object. Only `Serializable` Java objects can be used. Marshalling using an `ObjectMessage` is very close to RMI `Serializable` arguments.

As a conclusion, JMS is not strict on exchange format: the format is only a data structure, not an application interface. This can be modified easily on both sides without any compilation. This is one of the successes of JMS, since it requires only a minimal change to make two applications communicate. JMS is very tolerant of what is really transported and it adapts easily to new versions.
The down side is that compilation cannot detect format inconsistency, and parsers are usually needed to analyze structured data, be it in ASN1, text or XML. And parsers are slow most of the time.

1.11 Routing in JMS

A Message header can contain properties, outside of the message body. These properties can be used to filter messages or to redirect them to some receivers. This works both for Topics and Queues.

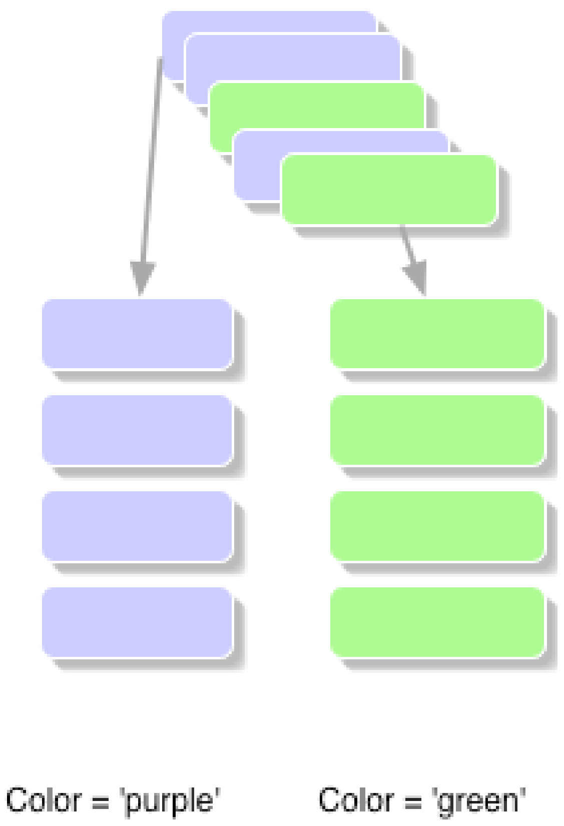


Fig 9

Filtering of properties uses a SQL-like syntax, so beware of single quotes for Strings in filter expressions.

Property names are usually Strings.

Property values can be `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, and `String`.

Sending a message with a property:

```
// create a Session as usual then …
          Topic topic= session.createTopic("Topic.Football");
          MessageProducer publisher = session.createProducer(topic);
          TextMessage message=session.createTextMessage("hello");
          message.setStringProperty("team", "blue");
          publisher.send(message);
// close Session and Connection
```

The filtering on reception is attached to the message consumer.

```
// create a Session as usual then …
     Topic topic= s.createTopic("Topic.Football");
// Beware of simple quotes here, like in SQL
     MessageConsumer receiver = s.createConsumer(topic, "team='blue'");
     qc.start();
     System.out.println("I'm waiting for blue teams results");
     TextMessage mess= (TextMessage) receiver.receive();
     System.out.println("received message : "+mess.getText());
// close Session and Connection
```

The consumer can also be attached to a MessageListener for asynchronous reception.

Routing can of course handle less mundane properties than colors: it could be used in a call center to route requests to operators according to some specialty, to route stock exchange orders according to importance of customers, etc.

2 Design issues with JMS

2.1 Using Queues or Topics

There are several criteria to select between a Queue and a Topic design.

- **Administration**: Queues are created by an administrator and are usually not created dynamically by program. Topics can be dynamically created.
- **Performance**: since Topics broadcast messages to a number of consumers, they are generally slower than Queues.

- **Single/multiple consumers**: one and only one consumer reads a Queue message. For multiple consumers, it is required to use Topics.
- **Reliability**: the acknowledgement protocol — see below — is pretty clear for a Queue: when a message is consumed, it is deleted from the Queue. Since there might be many subscribers to Topics, and this might vary through time, there is no way to determine when to safely acknowledge a message. For simplicity, the message is simply deleted at the end of the session. The inactive consumers will not get a new chance to get the message, except if there are durable subscribers— see above.

2.2 Using synchronous/asynchronous reception

<To be completed>

2.3 Browsing a Queue without an administrative tool

Sometimes, one needs to peek into the Queue without actually consuming any message, for instance to check the number of messages in the Queue.

<to be completed : rewrite using new java 5 iterators>

```
import javax.jms.*;
// create a Session and a Queue as usual
     QueueBrowser browser = session.createBrowser(aQueue);
     int cnt = 0;
     Enumeration e = browser.getEnumeration();
     if(!e.hasMoreElements())
          {
               System.out.println ("<no messages in queue>");
          }
     else
          {
          while(e.hasMoreElements())
               {
               System.out.print(" --> getting message " + (++cnt));
// Assume abusively it is a TextMessage – should test for type
               TextMessage message = (TextMessage) e.nextElement();
               if (message != null)
                    {
                    System.out.println(message.getText()) ;
                    }
               } // end while
          }// end else
// Free any resources in the browser.
    browser.close();
```

2.4 Intermediate storage

Consider an application receiving sampled data at a regular rate in some very sophisticated physical device. The detection of any distortion in the linear ramp of the data must trigger an alarm in the monitoring application.

Data can be transmitted either by JMS or Corba, both the monitoring application and the physical device software can implement it. Since JMS uses intermediate storage, which can be more or less loaded, the processing time of a message can vary with a larger deviation than in Corba. As a result the distortion in the sampling rate of data is enhanced by the JMS intermediate storage.

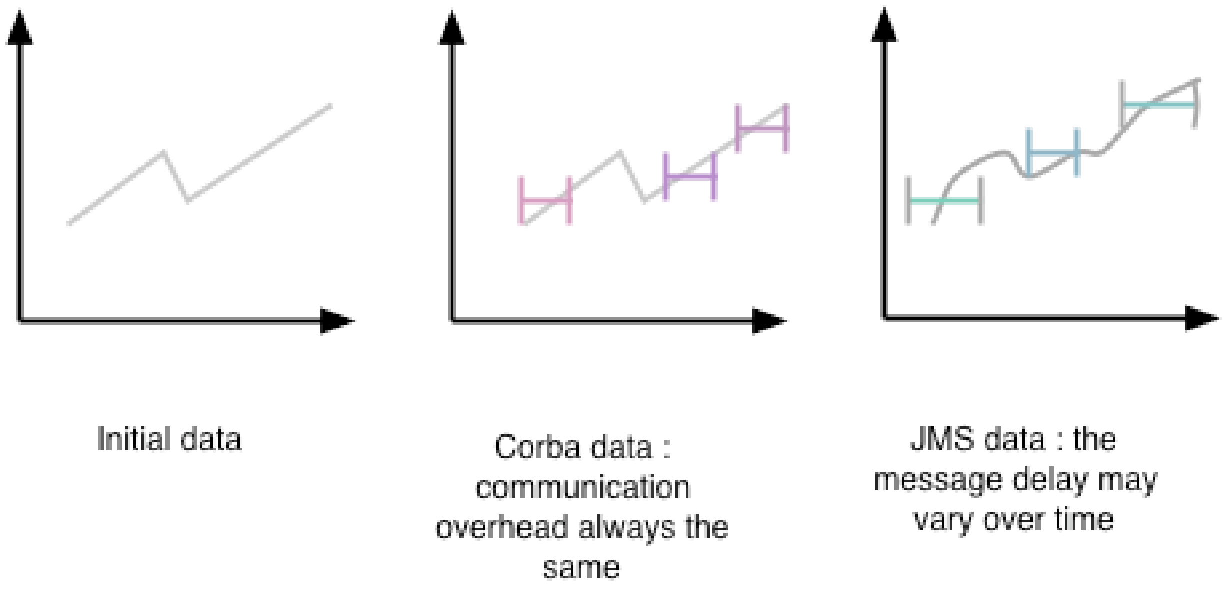

Initial data
Corba data : communication overhead always the same
JMS data : the message delay may vary over time

Fig 12

2.5 Temporal series of messages

If there is only one server, there is usually a good chance that all messages are sent and received in the same order, except if there have different priorities (see paragraph …). One task of the JMS server is to ensure such reliability.

Yet, it is possible with some JMS clusters using load balancing that the messages do not follow the same route. This way, they can arrive in any order. There are some applications where this can matter a lot. For instance suppose a voice-on-IP telephone billing application, where the computer telephone sends messages at the beginning and the end of a conversation.

Of course, if the order of the messages is changed, there is no way to bill the client computer.

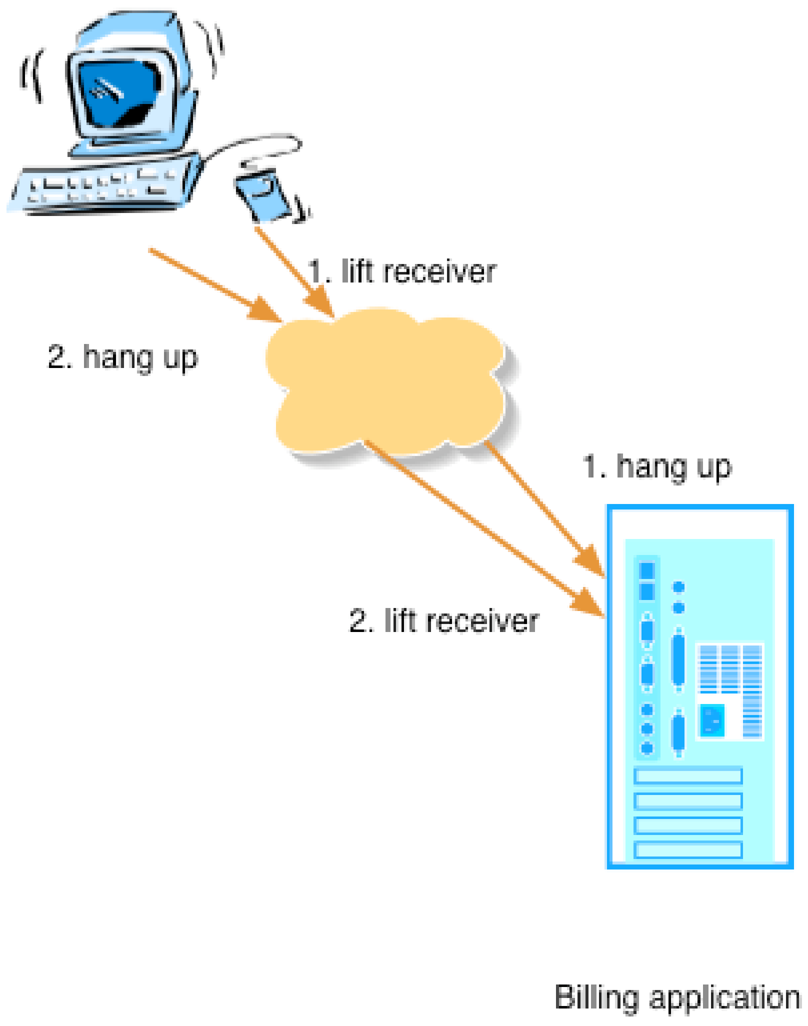


Fig 13

Of course, this is a very extreme case, but since JMS is "asynchronous", there is never any guarantee of message delivery order. One should be careful to mentally check an application for:

- Determinism of message routing,
- Risk of message reverse order.

3 Quality of service in JMS

3.1 Message header

<to be completed : priorities, message lifetime, persistence>

3.2 Message acknowledgement

Message acknowledgement is very important for JMS quality of service.

Suppose you want to withdraw money from a cash dispenser, and the server sends to your machine a "timed-out connection" message.

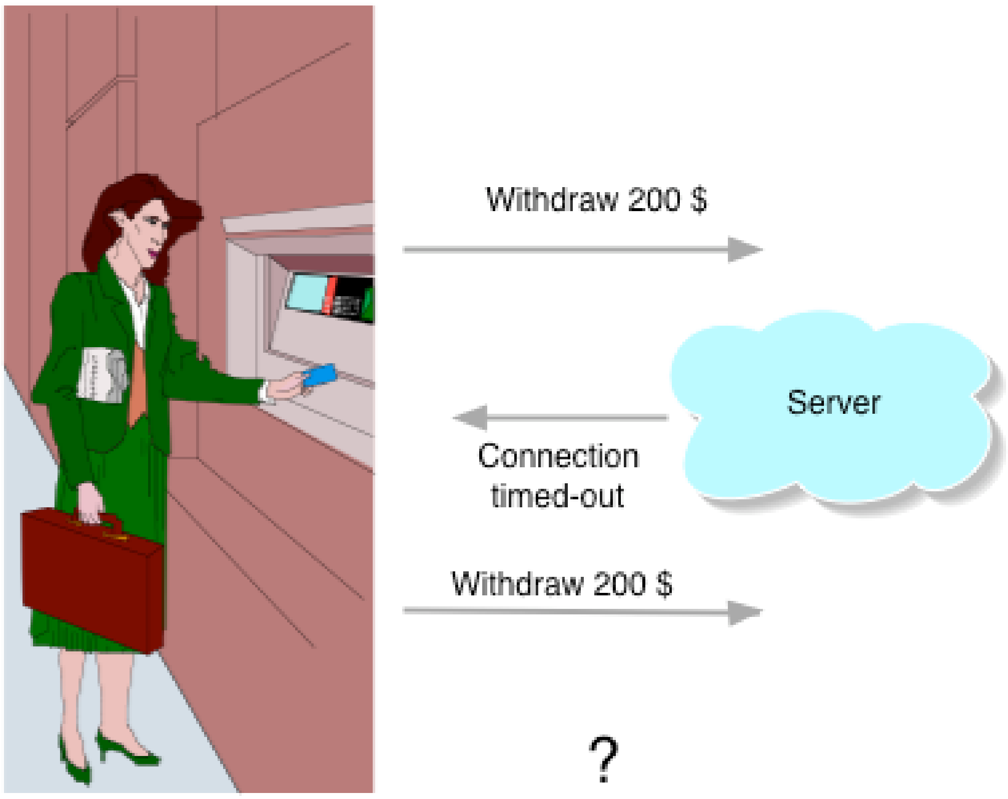


Fig 14

Then several cases can occur among which:

- You send the message once again, and the customer might to be debited twice for only one real transaction

- You don't send the message again, and the bank might have given money without writing down the balance.

Depending on the back-end processing, the result can vary a lot if you do not really know how many identical messages have been sent: 0, 1 or 2. Achieving "one only" semantics is one of the most difficult issues in distributed programming.

3.2.1 Service-level message acknowledgement

<to be completed, using acknowledge()>

3.2.2 Application-level message acknowledgement

<to be completed, using Requestor>

3.3 Using transactions in JMS

As a quick introduction, a transaction allows an all-or-nothing sequence of send-receive operations: this is called atomicity. In order to do that, the JMS Server manages some journaling and checkpoints for recovery by saving state at the beginning of a transaction.

Suppose your application wants to ensure non shipping of a product without simultaneous invoicing. *<to be completed>*

3.4 From JMS to EJBs

<to be completed : describe how JMS transaction context can be propagated to a DB transaction context using XATransactions and EJB userTransaction context>