Unit 2. Message-oriented and peer to peer systems

Outcomes Summary. You will learn

- to program messaging among Java applications using the JMS standard
- to program a simple P2P system using JMS
- describe characteristics and benefits of P2P DS
- explain why peers need to implement a routing facility, giving at least two reasons
- explain how prefix routing works giving a simplified example

Further Reading: Sun JMS tutorial, CDK2005 10.

2.1 Introduction to Java Message Service

Java Message Service (JMS) is a standard developed by Sun in 1998 to enable Java applications to connect to existing message-oriented middleware such as IBM MQSeries. A Java API is the major part of this spectification and every Java EE-compliant application server (such as IBM WebSphere or Glassfish) provides messaging facilities via this API. Since such application servers are complex to manage and resource-intensive, there are also JMS implementations independent of other Java EE functionality, including Apache ActiveMQ and Mantaray. We will be using Mantaray in the labs because it implements JMS messaging *without* the need to pass messages through a central server, which is essential for peer-to-peer programming.

2.1.1 Sending a message in JMS

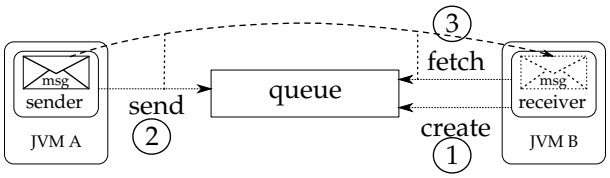


Figure 2.1: Abstract overview of JMS point-to-point message transfer.

To send a message, one needs access to a deployed delivery infrastructure (such as the Royal Mail), an address of the recipient understood by the delivery infrastructure and some kind of envelope to contain the message, onto which the recipient's address and other meta-data can be entered. In JMS the delivery infrastructure is provided by certain Java objects resident in the JVMs of all communicating participants. Mantaray is the JMS provider we will use in the examples here and in the practical sessions.

Continuing the analogy with ordinary mail, a participant who wishes to receive messages has to "live" at an address known to the delivery infrastructure and the location has to have a post box. In JMS post boxes are provided by special objects called "queues". A participant who wishes to receive some messages creates a queue object, assigning it a name (arbitrary string), which then serves as the address of the queue. Messages are sent to this queue using its name and the participant who created it can pick up the messages from the queue object and inspect them.

The sender needs to know the name of the queue it wants to send a message to. It can create a queue object using this name in exactly the same way as the receiver. The delivery infrastructure will automatically connect the two local queue objects — the one created by the receiver and the one created by the sender, even if they reside on different computers. When the sender sends something to its queue object, it will be available at the receiver's queue object after some (usually short) time.

The delivery is *asynchronous*, which means that the delivery infrastructure does not give the sender any way of finding out whether and when the message has reached some recipient. If a sender posts to an non-existent address, JMS implementations will usually wait for some time whether or not someone who connects to this queue (ie has a queue object with this name) picks up the message. If not, the message is lost without much ado (as occasionally happens to paper post).

We do not aim to understand all aspects of JMS in this module. For a complete conceptual introduction to JMS see the Sun tutorial.

2.1.2 JMS example code

The implementation of a simple JMS receiver is shown in Fig. 2.2. The JMS provider is visible only on line 27, where a certain *connection factory* is created. A connection factory is required to initialise the JMS provider on the local JVM. This happens in the form a *connection object*, from which at least one *session* needs to be created for managing queues. With the help of the session, the program creates a *queue object* and a *receiver object* that is capable of retrieving messages from this queue.

Fig. 2.3 is a slightly modified version of the receiver program, in which on line 31 instances of the `Receiver` class are made listeners for the queue they created. To be qualified a listener, the class has to implement the interface `MessageListener` which puts an obligation on it to implement method `onMessage`. This way there is no need to program a loop that keeps fetching one message after another. The method `onMessage` is called by the JMS infrastruture automatically as soon as a message reaches the queue on this JVM. Note that the method is not allowed to propagate any checked exceptions and thus has to deal with a potential `JVMException`.

A simple sender is shown in Fig. 2.4. It is fairly similar to the receiver except it uses the session object to create a *message object*, which serves as an envelope to the data that is to be sent. In this case the data sent is plain text originating from a Java string. `TextMessage` is a special JMS envelope for text. The most versatile "envelope" is called `ObjectMessage` — it can be used to wrap any Java serialisable object. Recall

```java
package jms.test.fetch;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueReceiver;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.mr.api.jms.MantaQueueConnectionFactory;

public class Receiver
{
    private String myName;
    private QueueConnection con;
    private QueueReceiver receiver;

    public Receiver(String myName) throws JMSException
    {
        this.myName = myName;

        // create a connection object via a factory:
        QueueConnectionFactory conFactory =
            (QueueConnectionFactory) new MantaQueueConnectionFactory();
        con = conFactory.createQueueConnection();

        // create a queue and an associated receiver:
        QueueSession session =
            (QueueSession) con.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        Queue receiveQueue = session.createQueue(myName);
        receiver = session.createReceiver(receiveQueue);

        // enable messaging to start:
        con.start();
    }

    public void fetchMessages() throws JMSException
    {
        while (true)
        {
            Message msg = receiver.receive();

            if (msg instanceof TextMessage)
            {
                TextMessage tmsg = (TextMessage) msg;
                System.out.println(myName +
                        ": received: " + tmsg.getText());
            }
        }
    }

    public static void main(String[] args) throws JMSException
    {
        Receiver r = new Receiver("receiver");
        r.fetchMessages();
    }
}
```

Figure 2.2: A simple JMS receiver program.

```java
package jms.test.listen;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
... etc

import org.mr.api.jms.MantaQueueConnectionFactory;

public class Receiver implements MessageListener
{
    private String myName;
    private QueueConnection con;

    public Receiver(String myName) throws JMSException
    {
        this.myName = myName;

        // create a connection object via a factory:
        QueueConnectionFactory conFactory =
            (QueueConnectionFactory) new MantaQueueConnectionFactory();
        con = conFactory.createQueueConnection();

        // create a queue and a receiver to listen on:
        QueueSession session =
            (QueueSession) con.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        Queue receiveQueue = session.createQueue(myName);
        QueueReceiver receiver = session.createReceiver(receiveQueue);

        // attach itself as a listener to the queue:
        receiver.setMessageListener(this);

        // enable messaging to start:
        con.start();
    }

    // this method will be called whenever a message arrives:
    @Override
    public void onMessage(Message msg)
    {
        if (msg instanceof TextMessage)
        {
            TextMessage tmsg = (TextMessage) msg;
            try
            {
                System.out.println(myName + ": received: " + tmsg.getText());
            }
            catch (JMSException e)
            {
                throw new Error(myName + ": could not extract text from message: "
                + e.getMessage());
            }
        }
    }

    public static void main(String[] args) throws JMSException
    {
        new Receiver("receiver");
    }
}
```

Figure 2.3: A simple JMS receiver program **listening** to the queue.

```java
package jms.test.fetch;

import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.mr.api.jms.MantaQueueConnectionFactory;

public class Sender
{
    private String myName;
    private QueueConnection con;
    private QueueSession session;

    public Sender(String myName) throws JMSException
    {
        this.myName = myName;

        // create a connection object via a factory:
        QueueConnectionFactory conFactory =
            (QueueConnectionFactory) new MantaQueueConnectionFactory();
        con = conFactory.createQueueConnection();

        // create a session for sending messages:
        session =
            (QueueSession) con.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

        // enable messaging to start:
        con.start();
    }

    public void sendMessage(String destination, String text) throws JMSException
    {
        Queue sendQueue = session.createQueue(destination);
        QueueSender sender = session.createSender(sendQueue);
        TextMessage tmsg = session.createTextMessage(myName + " says: " + text);
        sender.send(tmsg);
    }

    public static void main(String[] args) throws JMSException
    {
        Sender s = new Sender("sender");
        s.sendMessage("receiver", "hello!");
    }
}
```

Figure 2.4: A simple JMS sender program.

that an object is serialisable when it has standard methods to convert its attribute values to/from a format suitable for transfer outside the JVM, allowing it to be copied between different JVMs or stored in a file or database.

2.1.3 Connecting Mantaray queues

So far we made the assumption that when JVM A creates a queue named "receiver" and JVM B creates a queue named "receiver", the two JVMs automatically connect and exchange messages sent to this queue. This indeed happens automatically on local networks, eg within one Aston lab. JVMs discover one another using IP broadcasting, ie sending a message to *all* computers on the network. Obviously, it would be neigher practical nor safe to send such a message to all computers in the world to find out whether some of them have a JVM running Mantaray with a queue of a certain name. Routers between local networks and firewalls therefore IP broadcast messages, which means that Mantaray peers cannot automatically connect unless they are on the same local network. To enable wider deployment of Mantaray systems, Mantaray provides a so-called *WAN*[1] *Bridge*, which is a lightweight server that enables peers to register their existence with and in return, it facilitates a broadcast across from one local network with some registered peers to another such local network, making it possible for peers to connect their queues across a WAN.

2.2 Peer to peer systems

These are systems in which responsibility for providing a service is evenly divided among all nodes. More precisely, a DS using peer to peer (P2P) architecture is characterised by:

- making all nodes contribute resources to the system;
- having all nodes functionally equivalent (although holding potentially different data);
- having each item of data placed in multiple nodes.

The benefits of these architectures is that P2P systems

- can be as scalable as their underlying network
  (ie their capacity grows with their size);
- can be immune to the failures of any individual nodes;
- can offer a good degree of anonymity to providers and users of resources.

The most successful application of P2P systems has been for sharing large chunks of immutable data (such as CD-ROM images, media files) across the Internet. The Freenet system (Clarke *et al* 2000) capitalises on the ability to make users anonymous,

[1] WAN = wide area network

providing a safe and independent Internet publication service to people in countries with oppressive regimes.

It is not easy to develop reliable P2P systems because of the need for an almost complete transparency of the Internet as an underlying network, as well as for replication transparency. In P2P systems, resources cannot be easily mapped to particular computers and messages often need to be routed towards these elusive resources rather than towards concrete peers.

To simplify the development of P2P systems, special *P2P middleware* has been developed that provides a very abstract view of the network suitable for P2P applications. It is easier to deal with immutable resources in this setting — the only hard issues with these are: how to locate *nearby* peers who together posses complete knowledge of a given resource and how to communicate with them efficiently, allowing for them to move, disappear or misbehave at any time. Middleware such as Pastry and Tapestry provide good generic support for such resource management and communication.

More advanced middleware supports also resources that are modified from time to time. The main trick is to explicitly identify the versions of the resource as it changes and always refer to the resource with a particular version. Thus the mutable resource becomes a collection of immutable resources (one for each version) linked together by a common identity. It may be necessary to compromise the P2P architecture and give some nodes a higher responsibility in order to allow peers to reliably order the versions to a sequence and identify the latest/current version of a given resource. Examples of middleware that supports mutable resources are OceanStore and Ivy (see Coulouris *et al* 2005 for further details).

2.2.1 Decentralised broadcasting

As an example, let us look at a very simple P2P version of a chat service (also used in Practical 2). Each peer contains one `Dispatcher` object that sets up an incoming channel and inspects every message that arrives on the channel and reponds to it appropriately.

Each peer also has one `Neighbours` object that maintains a set of references to neighbouring peers. When a peer gets a wisdom it has not seen before, it forwards it to all neighbours (see Figure 2.5). This routing algorithm is ok for broadcasting messages across all peers. Nevertheless, the peers also support one-to-one messaging and for this purpose this algorithm is rather inefficient and would not scale well to large number of peers. We will explore how to improve it a bit later.

The relevant parts of `Dispatcher` and `Neighbours` classes are shown in Figure 2.6.

Neighbour discovery. It is easy to see how the broadcast routing in a P2P system works once each peer knows at least 2 other peers and there is path via neighbours between any two peers. How can one start-off such a network of neighbouring peers? In Practical 2, the users had to manually establish these connections, which is not very satisfactory.

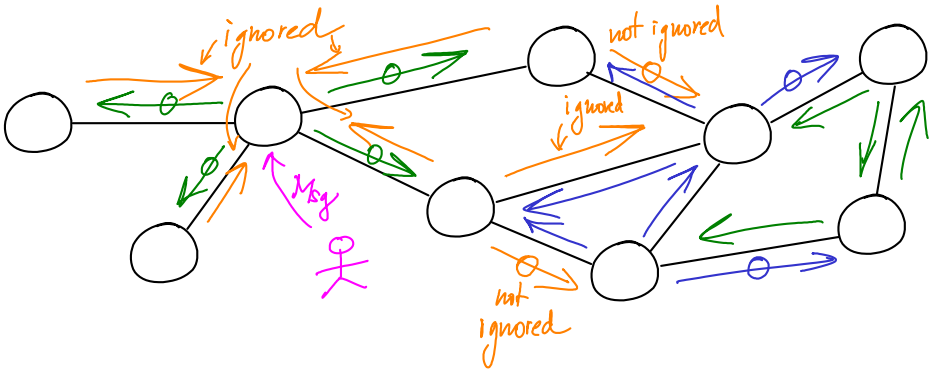


Figure 2.5: Execution of a simple P2P broadcast routing algorithm

One could discover fellow peers on a local network using IP broadcasting facilities. Nevertheless, this way there is a danger that everyone could be everyone's neighbour and the benefits of the P2P architecture would be lost. Various algorithms are used to avoid such problems, either randomly selecting a small number of neighbours for each peer or conducting tests of proximity using bandwidth measurements.

As these algorithms are quite complicated, it is usually wise to make use of a P2P middleware (eg Mantaray) to take care of neighbour discovery or make use of a centralised service to determine neighbourhood relations (as done eg in Napster). Some systems use a hybrid between these two approaches: some peers, called "supernodes", assume a special role, besides the role shared by all peers, to organise neighbourhood relations among peers in its vicinity (for example Kazaa and Skype use this approach).

Logical not physical neighbours. It is important to remember that 'neighbours' for P2P nodes are not necessarily physical neighbours in the underlying Internet-level network. In some cases they could be on the opposite sides of the World, with their TCP channels being routed over dozens of computers. Nevertheless, it is a good idea to try and keep these P2P neighbours physically as close as possible to improve efficiency. P2P middleware usually does a good job at achieving this goal.

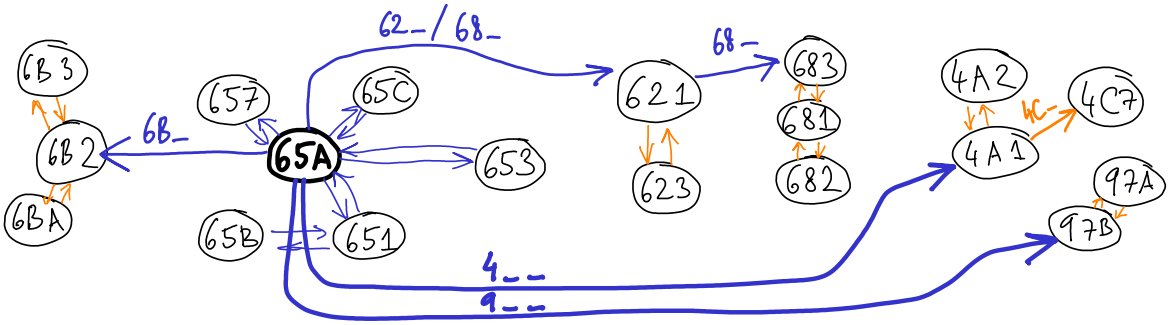
The network formed by neighbours in a P2P system is often called the *overlay network* of the system. This overlay network is often provided not by the system directly but by its P2P middleware. Overlay network is obviouly more abstract than its underlying network and, moreover, peers have to implement their own routing mechanism, sometimes called a *routing overlay*.

2.2.2 Decentralised routing to objects

When a message is routed to a single P2P node, the routers do not broadcast it but use a hierarchical scheme similar to the one used at the IP level. When the destination is very near, it is often in the list of neighbours and the message can be routed directly without any 'hops' over other nodes. When it is further away, the router will decide which of its neighbours it will use as the first hop.

```
1  public class Dispatcher implements javax.jms.MessageListener
2  {
3    ...
4      public void onMessage(Message msg)
5      {
6          // pull out an object from the message:
7          ObjectMessage omsg = (ObjectMessage) msg;
8          Serializable obj;
9          try
10         {
11             obj = omsg.getObject();
12         }
13         catch (JMSException e)
14         {
15             throw new Error("wisdom.peer.Dispatcher.onMessage: "
16                     + "failed to retrieve message object: " + e.getMessage());
17         }
18         ...
19         else if (obj instanceof NewWisdom)
20         {
21             NewWisdom nw = (NewWisdom) obj;
22
23             if (wisdomIDs.add(nw.getWisdom().getId())) // test if new
24             {
25                 System.out.println("received new wisdom: " + nw);
26                 processNewWisdom(nw);
27             }
28         }
29         ...
30     }
31     private void processNewWisdom(NewWisdom nw)
32     {
33         // store this wisdom:
34         ...
35         // forward the whole wisdom to the neighbours:
36         forwardWisdom(nw);
37     }
38     private void forwardWisdom(NewWisdom nw)
39     {
40         try { neighbours.newWisdom(nw); } catch ...
41     }
42   ...
43     private Neighbours neighbours;
44 }
45
46 public class Neighbours
47 {
48   ...
49     public synchronized void newWisdom(NewWisdom nw) throws JMSException
50     {
51         ...
52         ObjectMessage msg = sendSession.createObjectMessage(nw);
53
54         for (QueueSender neighbour : neighbours.values())
55         {
56             neighbour.send(msg, DeliveryMode.NON_PERSISTENT,
57                     Message.DEFAULT_PRIORITY, MESSAGE_TTL);
58         }
59     }
60   ...
61 }
```

Figure 2.6: Simple boadcast routing in a P2P chat system.



Routing table at peer with address 65A:

level 1		level 2		level 3	
651	pc0**4**.dom1	62_	pc03.dom2	4__	pc02.dom5
652	pc01.dom1	68_	pc03.dom2	9__	pc07.dom3
657	pc02.dom**2**	6B_	pc11.dom6		⋮
65B	pc0**4**.dom1		⋮		
65C	pc08.dom1				
	⋮				

Figure 2.7: Illustration of prefix routing

How do routers work out how far is the destination? The only information the router has is the destination address and its routing table. The distance is usually established by comparing the destination address with its own address. For example, in so called *prefix routing*, two nodes are considered close if their addresses start with the same prefix. The longer the common prefix, the closer the nodes are considered to be. See Figure 2.7 for an illustration of what a routing table based on prefix comparison could look like. The closer the nodes, the more detailed the peer's routing table is for them. In practice, there would be more than 3 levels of routing and the addresses would be much longer than 3 characters. The most common address format is the so-called Globally Unique Identifier (GUID) consisting of 16 hexadecimal digits.

What addresses to use? In a P2P network, one cannot usually use fixed IP or DNS addresses for peers because the same routing mechanism is used to locate resources or *objects* stored at peers as well as peers themselves. Objects are often replicated between multiple nodes and routing tables direct different peers to different copies of the object managed by different nodes. This is illustrated in Figure 2.8.

Another reasons why P2P networks must use their own peer addressing and routing are:

- Sometimes peers are relocated from one computer to another while keeping their identity whichever computer they run on.
- Some P2P systems are deployed on wireless networks where computers may change their IP every now and then.

How are peers and objects assigned addresses? Usually the middleware gives them an address that starts with a prefix common with most of their physically close peers and

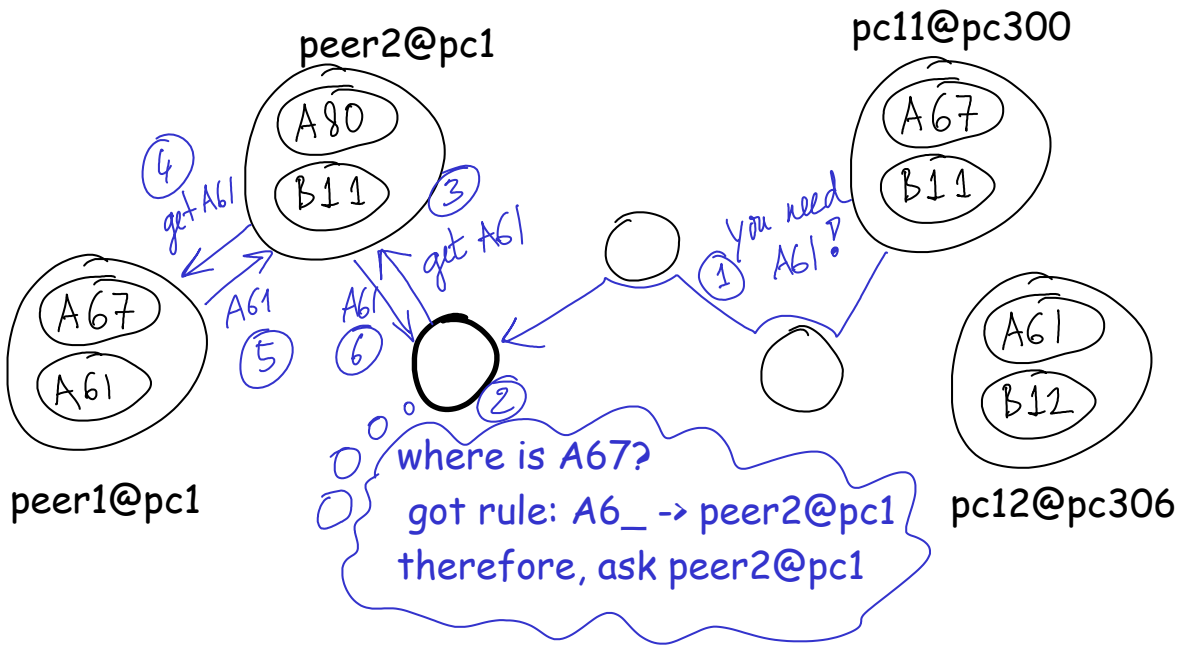


Figure 2.8: Routing to a nearby replica of an object

objects and add a randomly generated portion which makes it very likely to be unique. If it happens that the randomly generated address already exists, the middleware must detect it soon afterwards and rename the new peer/object.

2.2.3 Distributed hash table (DHT)

A very common instance of routing to resources is where the whole P2P system holds a hash table, ie some assignment from keys to values. Each peer holds information about the values for a subset of keys. When a peer needs to work out a value for some specific key, they send request for the value stating the key and this request is routed towads a peer that holds the value for that key. The key acts as the resource name in this case. The key is usually a hash of the value. Prefixes of the hash value are used in routing tables to quickly locate a peer that holds the required key-value pair. Eg in BitTorrent the values held in the distributed table are small pieces of large files and their keys consist of hash values of these file pieces.