Figure 4.1  Middleware layers

- Applications, services
- RMI and RPC
- Request-reply protocol
  marshalling and external data representation
- UDP and TCP

This chapter
Figure 4.2  Sockets and ports

Internet address = 138.37.94.248

Internet address = 138.37.88.249
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[])
        { // args give message contents and server hostname
            DatagramSocket aSocket = null;
            try {
                aSocket = new DatagramSocket();
                byte [] m = args[0].getBytes();
                InetAddress aHost = InetAddress.getByName(args[1]);
                int serverPort = 6789;
                DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
                aSocket.send(request);
                byte[] buffer = new byte[1000];
                DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(reply);
                System.out.println("Reply: " + new String(reply.getData()));
            }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
            catch (IOException e){System.out.println("IO: " + e.getMessage());}
        } finally { if(aSocket != null) aSocket.close();}
    }
}
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]){
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                        request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
        catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        finally {if(aSocket != null) aSocket.close();}
    }
}
Figure 4.5  TCP client makes connection to server, sends request and receives reply

```java
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname of destination
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out =
                new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]);        // UTF is a string encoding see Sn 4.3
            String data = in.readUTF();      System.out.println("Received: "+ data);
        }catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage());
        }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
        }catch (IOException e){System.out.println("IO:"+e.getMessage());
        } finally {if(s!=null) try {s.close();}catch (IOException e){/*close failed*/}}
    }
}
```
import java.net.*;
import java.io.*;

public class TCPServer {
    public static void main (String args[]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
    }
}

// this figure continues on the next slide
Figure 4.6 continued

class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
        try {
            clientSocket = aClientSocket;
            in = new DataInputStream( clientSocket.getInputStream());
            out =new DataOutputStream( clientSocket.getOutputStream());
            this.start();
        } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
    }
    public void run(){
        try {
            // an echo server
            String data = in.readUTF();
            out.writeUTF(data);
        } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
        } catch(IOException e) {System.out.println("IO:"+e.getMessage());
        } finally { try {clientSocket.close();}catch (IOException e){/*close failed*/}}
    }
}
<table>
<thead>
<tr>
<th>Type</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td>length (unsigned long) followed by elements in order</td>
</tr>
<tr>
<td>string</td>
<td>length (unsigned long) followed by characters in order (can also have wide characters)</td>
</tr>
<tr>
<td>array</td>
<td>array elements in order (no length specified because it is fixed)</td>
</tr>
<tr>
<td>struct</td>
<td>in the order of declaration of the components</td>
</tr>
<tr>
<td>enumerated</td>
<td>unsigned long (the values are specified by the order declared)</td>
</tr>
<tr>
<td>union</td>
<td>type tag followed by the selected member</td>
</tr>
</tbody>
</table>
Figure 4.8  CORBA CDR message

<table>
<thead>
<tr>
<th>index in sequence of bytes</th>
<th>4 bytes</th>
<th>notes on representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–3</td>
<td>5</td>
<td>length of string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘Smith’</td>
</tr>
<tr>
<td>4–7</td>
<td>&quot;Smit&quot;</td>
<td>length of string</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘London’</td>
</tr>
<tr>
<td>8–11</td>
<td>&quot;h___&quot;</td>
<td>unsigned long</td>
</tr>
<tr>
<td>12–15</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>16–19</td>
<td>&quot;Lond&quot;</td>
<td></td>
</tr>
<tr>
<td>20–23</td>
<td>&quot;on__&quot;</td>
<td></td>
</tr>
<tr>
<td>24–27</td>
<td>1934</td>
<td></td>
</tr>
</tbody>
</table>

The flattened form represents a Person struct with value: {'Smith', 'London', 1934}
### Figure 4.9  Indication of Java serialized form

<table>
<thead>
<tr>
<th>Person</th>
<th>8-byte version number</th>
<th>h0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>int year</td>
<td>java.lang.String</td>
</tr>
<tr>
<td></td>
<td>name:</td>
<td>place:</td>
</tr>
<tr>
<td>1934</td>
<td>5 Smith</td>
<td>6 London</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h1</td>
</tr>
</tbody>
</table>

The true serialized form contains additional type markers; h0 and h1 are handles.

*Serialized values*  

- Class name, version number
- Number, type and name of instance variables
- Values of instance variables
Figure 4.10  Representation of a remote object reference

<table>
<thead>
<tr>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
<th>32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet address</td>
<td>port number</td>
<td>time</td>
<td>object number</td>
</tr>
<tr>
<td>interface of remote object</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 4.11 Request-reply communication

Client

\textit{doOperation}

•

•

(wait)

•

(continuation)

Server

\textit{getRequest}

select object

execute

method

\textit{sendReply}

\textit{Request message}

\textit{Reply message}
Figure 4.12  Operations of the request-reply protocol

```java
public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
    sends a request message to the remote object and returns the reply.
    The arguments specify the remote object, the method to be invoked and the
    arguments of that method.

public byte[] getRequest ();
    acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
    sends the reply message `reply` to the client at its Internet address and port.
```
**Figure 4.13  Request-reply message structure**

<table>
<thead>
<tr>
<th>messageType</th>
<th>int (0=Request, 1=Reply)</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestId</td>
<td>int</td>
</tr>
<tr>
<td>objectReference</td>
<td>RemoteObjectRef</td>
</tr>
<tr>
<td>methodId</td>
<td>int or Method</td>
</tr>
<tr>
<td>arguments</td>
<td>// array of bytes</td>
</tr>
<tr>
<td>Name</td>
<td>Client</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>R</td>
<td>Request</td>
</tr>
<tr>
<td>RR</td>
<td>Request</td>
</tr>
<tr>
<td>RRA</td>
<td>Request</td>
</tr>
</tbody>
</table>
### Figure 4.15  HTTP request message

<table>
<thead>
<tr>
<th>method</th>
<th>URL or pathname</th>
<th>HTTP version</th>
<th>headers</th>
<th>message body</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>//www.dcs.qmw.ac.uk/index.html</td>
<td>HTTP/ 1.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 4.16  HTTP reply message

<table>
<thead>
<tr>
<th>HTTP version</th>
<th>status code</th>
<th>reason</th>
<th>headers</th>
<th>message body</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/1.1</td>
<td>200</td>
<td>OK</td>
<td></td>
<td>resource data</td>
</tr>
</tbody>
</table>
import java.net.*;
import java.io.*;
public class MulticastPeer{
    public static void main(String args[]){
        // args give message contents & destination multicast group (e.g. "228.5.6.7")
        MulticastSocket s =null;
        try {
            InetAddress group = InetAddress.getByName(args[1]);
            s = new MulticastSocket(6789);
            s.joinGroup(group);
            byte [] m = args[0].getBytes();
            DatagramPacket messageOut =
                new DatagramPacket(m, m.length, group, 6789);
            s.send(messageOut);
        }
    }
}
Figure 4.17 continued

```
// get messages from others in group
byte[] buffer = new byte[1000];
for(int i=0; i< 3; i++) {
    DatagramPacket messageIn = 
        new DatagramPacket(buffer, buffer.length);
    s.receive(messageIn);
    System.out.println("Received:" + new String(messageIn.getData()));
}
s.leaveGroup(group);
}catch (SocketException e){System.out.println("Socket: " + e.getMessage());
}catch (IOException e){System.out.println("IO: " + e.getMessage());
} finally { if(s != null) s.close();
```
Figure 4.18 Sockets used for datagrams

Sending a message

```c
s = socket(AF_INET, SOCK_DGRAM, 0)
bind(s, ClientAddress)
sendto(s, "message", ServerAddress)
```

Receiving a message

```c
s = socket(AF_INET, SOCK_DGRAM, 0)
bind(s, ServerAddress)
amount = recvfrom(s, buffer, from)
```

*ServerAddress* and *ClientAddress* are socket addresses
Figure 4.19  Sockets used for streams

Requesting a connection

\[ s = \text{socket(AF_INET, SOCK_STREAM,0)} \]
\[ \cdot \]
\[ \text{connect}(s, \text{ServerAddress}) \]
\[ \cdot \]
\[ \text{write}(s, \text{"message"}, \text{length}) \]

Listening and accepting a connection

\[ s = \text{socket(AF_INET, SOCK_STREAM,0)} \]
\[ \cdot \]
\[ \text{bind}(s, \text{ServerAddress}) \]
\[ \cdot \]
\[ \text{listen}(s,5) \]
\[ \cdot \]
\[ sNew = \text{accept}(s, \text{ClientAddress}) \]
\[ \cdot \]
\[ n = \text{read}(sNew, \text{buffer}, \text{amount}) \]

`ServerAddress` and `ClientAddress` are socket addresses