Figure 4.1 Middleware layers

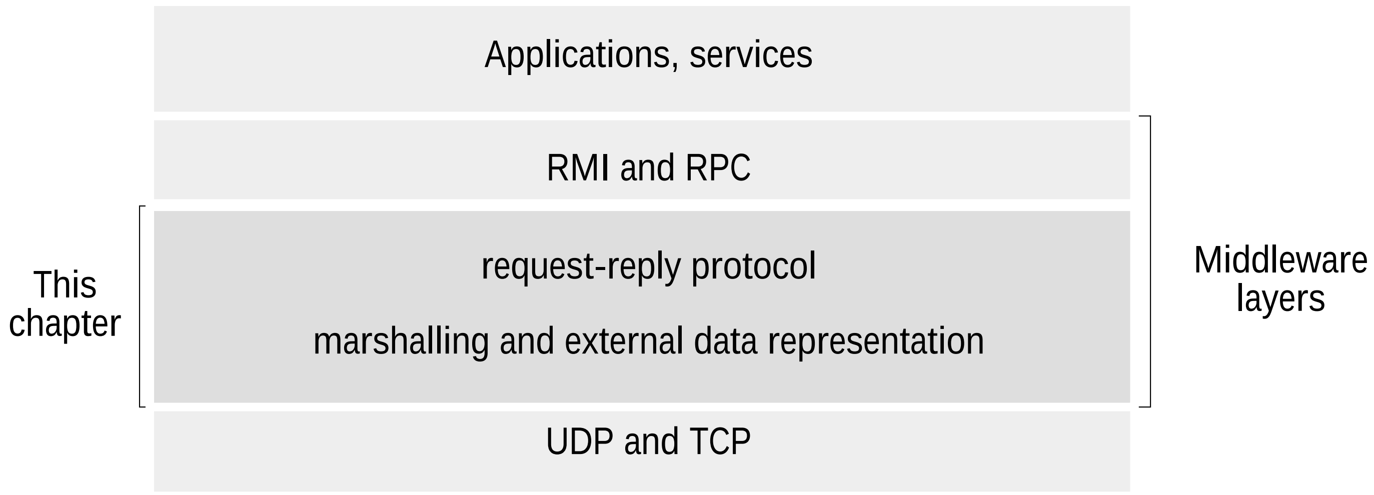


Figure 4.2 Sockets and ports

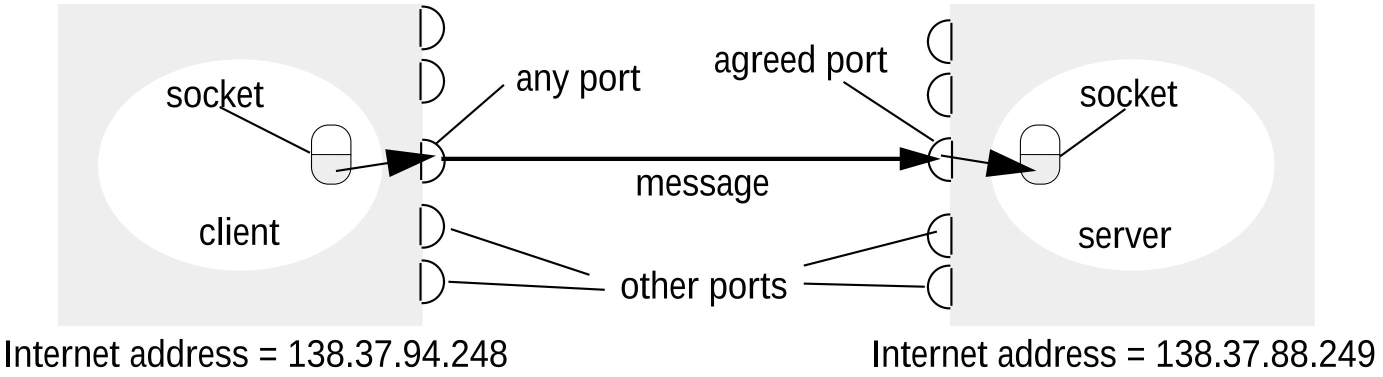

socket
client
any port
agreed port
message
socket
server
other ports
Internet address = 138.37.94.248
Internet address = 138.37.88.249

Figure 4.3 UDP client sends a message to the server and gets a reply

```
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();}
    }
}
```

Figure 4.4 UDP server repeatedly receives a request and sends it back to the client

```
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

```
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*/}}
    }
}
```

Figure 4.6 TCP server makes a connection for each client and then echoes the client’s request

```
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*/}}
  }
}
```

Figure 4.7 CORBA CDR for constructed types

Type	*Representation*
sequence	length (unsigned long) followed by elements in order
string	length (unsigned long) followed by characters in order (can also can have wide characters)
array	array elements in order (no length specified because it is fixed)
struct	in the order of declaration of the components
enumerated	unsigned long (the values are specified by the order declared)
union	type tag followed by the selected member

Figure 4.8 CORBA CDR message

index in sequence of bytes	◄— 4 bytes —►	*notes on representation*
0–3	5	*length of string*
4–7	"Smit"	*‘Smith’*
8–11	"h___"	
12–15	6	*length of string*
16–19	"Lond"	*‘London’*
20–23	"on__"	
24–27	1934	*unsigned long*

The flattened form represents a *Person* struct with value: {‘Smith’, ‘London’, 1934}

Figure 4.9 Indication of Java serialized form

Serialized values				*Explanation*
Person	8-byte version number		h0	*class name, version number*
3	int year	java.lang.String name:	java.lang.String place:	*number, type and name of instance variables*
1934	5 Smith	6 London	h1	*values of instance variables*

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

Figure 4.10 Representation of a remote object reference

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

Figure 4.11 Request-reply communication

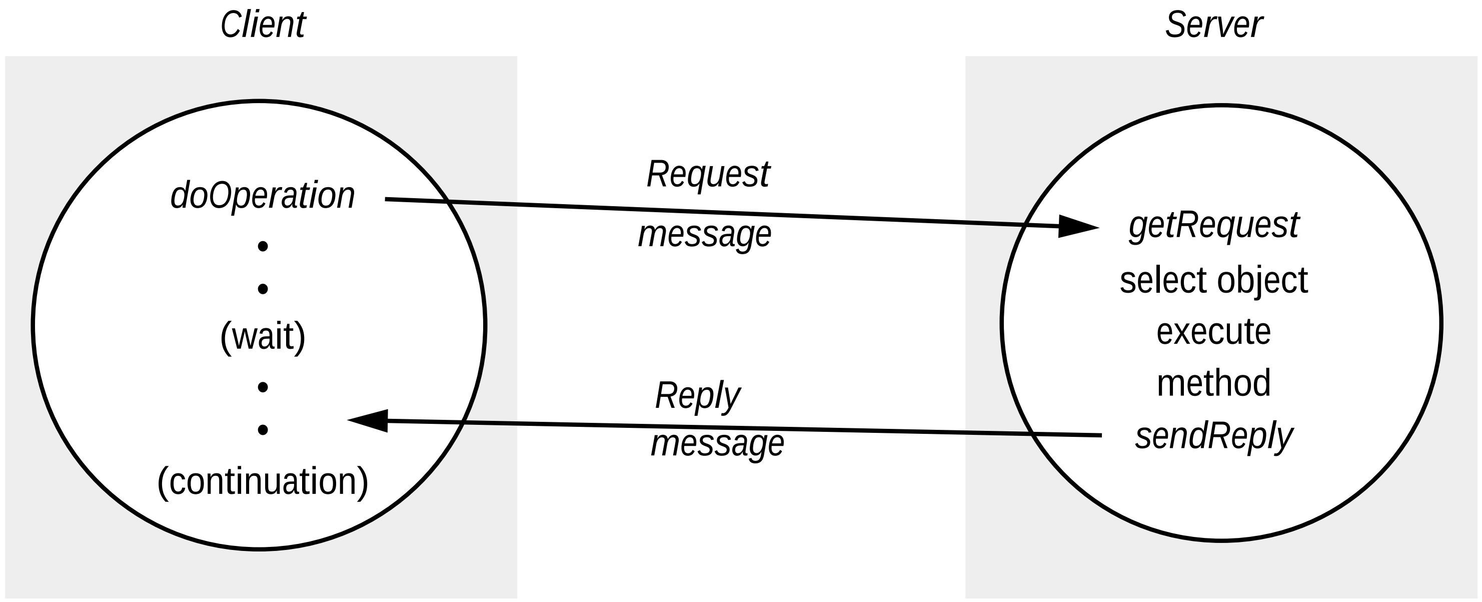


Figure 4.12 Operations of the request-reply protocol

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

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
objectReference	*RemoteObjectRef*
methodId	*int or Method*
arguments	*// array of bytes*

Figure 4.14 RPC exchange protocols

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

Figure 4.15 HTTP *request* message

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Figure 4.16 HTTP *reply* message

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

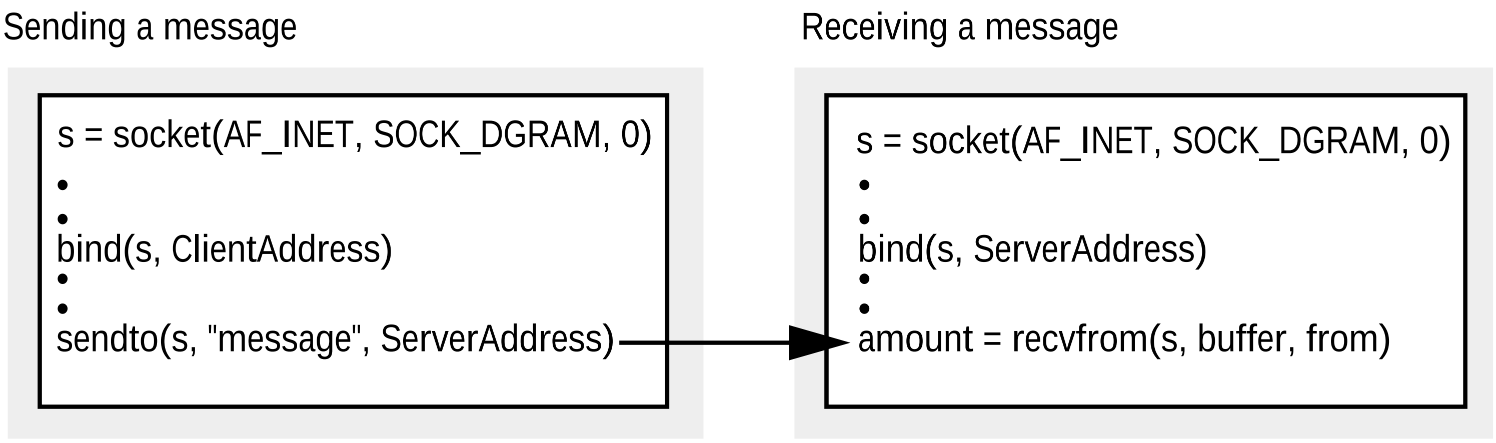
Figure 4.17 Multicast peer joins a group and sends and receives datagrams

```
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);



         // this figure continued on the next slide
```

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

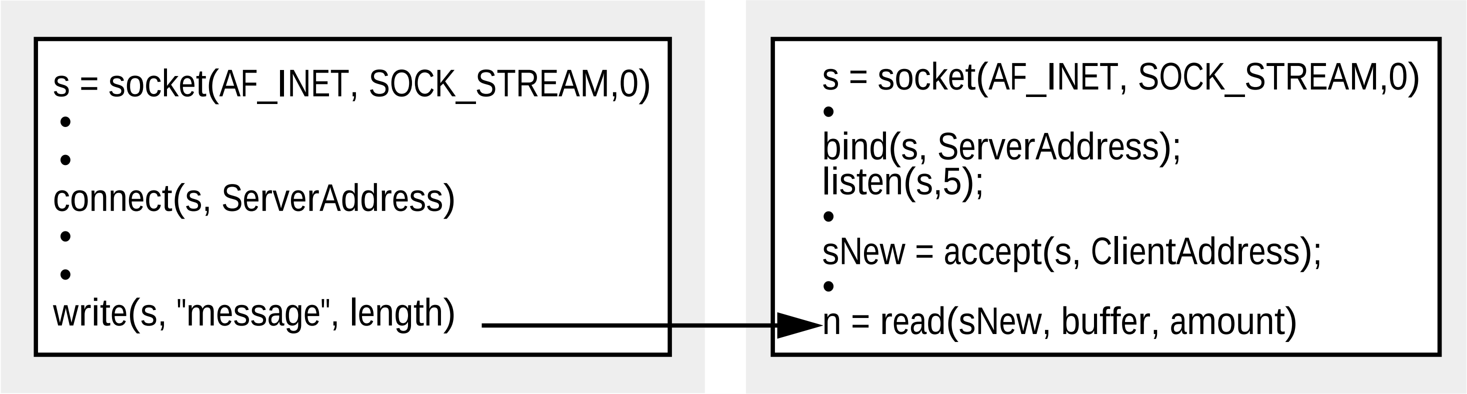
```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ClientAddress)
•
•
sendto(s, "message", ServerAddress)
```

Receiving a message

```
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 = socket(AF_INET, SOCK_STREAM,0)
•
•
connect(s, ServerAddress)
•
•
write(s, "message", length)
```

Listening and accepting a connection

```
s = socket(AF_INET, SOCK_STREAM,0)
•
bind(s, ServerAddress);
listen(s,5);
•
sNew = accept(s, ClientAddress);
•
n = read(sNew, buffer, amount)
```

ServerAddress and *ClientAddress* are socket addresses