4

INTERPROCESS COMMUNICATION

This chapter is concerned with the characteristics of protocols for communication between processes in a distributed system, both in its own right and as support for communication between objects.

The Java API for interprocess communication in the Internet provides both datagram and stream communication. These are presented, together with a discussion of their failure models. They provide alternative building blocks for communication protocols.

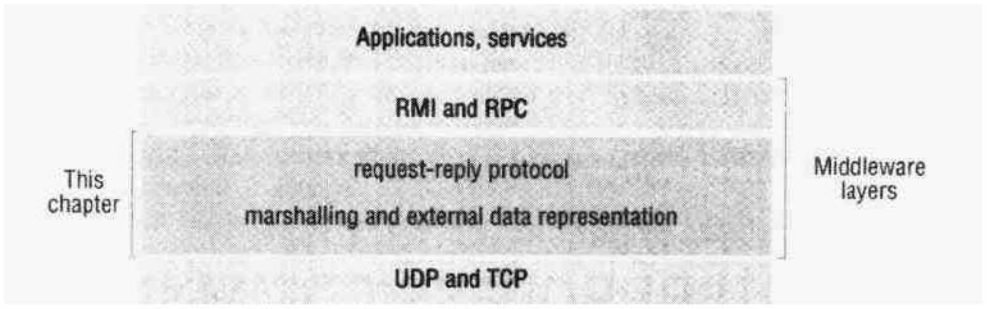
We discuss protocols for the representation of collections of data objects in messages and of references to remote objects.

We discuss the construction of protocols to support the two communication patterns that are most commonly used in distributed programs:

- client-server communication – in which request and reply messages provide the basis for remote method invocation or remote procedure call;
- group communication – in which the same message is sent to several processes.

Interprocess communication in UNIX is dealt with as a case study.

Figure 4.1 Middleware layers



4.1 Introduction

This chapter and the next are concerned with middleware. This one is concerned with the dark layer shown in Figure 4.1. The layer above is discussed in Chapter 5; it is concerned with integrating communication into a programming language paradigm, for example by providing remote method invocation (RMI) or remote procedure calling (RPC). *Remote method invocation* allows an object to invoke a method in an object in a remote process. Examples of systems for remote invocation are CORBA and Java RMI. In a similar way, a *remote procedure call* allows a client to call a procedure in a remote server.

Chapter 3 discusses the Internet transport-level protocols UDP and TCP without saying how middleware and application programs could use these protocols. The next section of this chapter introduces the characteristics of interprocess communication and then discusses UDP and TCP from a programmer's point of view, presenting the Java interface to these two protocols, together with a discussion of their failure model. The last section of this chapter presents the UNIX socket interface to UDP and TCP as a case study.

The application program interface to UDP provides a *message passing* abstraction – the simplest form of interprocess communication. This enables a sending process to transmit a single message to a receiving process. The independent packets containing these messages are called *datagrams*. In the Java and UNIX APIs, the sender specifies the destination using a socket – an indirect reference to a particular port used by the destination process at a destination computer.

The application program interface to TCP provides the abstraction of a two-way *stream* between pairs of processes. The information communicated consists of a stream of data items with no message boundaries. Streams provide a building block for producer–consumer communication [Bacon 1998]. A producer and a consumer form a pair of processes in which the role of the first is to produce data items and the role of the second is to consume them. The data items sent by the producer to the consumer are queued on arrival until the consumer is ready to receive them. The consumer must wait when no data items are available. The producer must wait if the storage used to hold the queued data items is exhausted.

The third section of this chapter is concerned with how the objects and data structures used in application programs can be translated into a form suitable for sending in messages over the network, taking into account the fact that different computers may use different representations for simple data items. It also discusses a suitable representation for object references in a distributed system.

The fourth and fifth sections of this chapter deal with the design of suitable protocols to support client-server and group communication. Request-reply protocols are designed to support client-server communication in the form of either RMI or RPC. Group multicast protocols are designed to support group communication. Group multicast is a form of interprocess communication in which one process in a group of processes transmits the same message to all members of the group.

Message-passing operations can be used to construct protocols to support particular process roles and communication patterns, for example remote method invocations. By examining the roles and communication patterns, it is possible to design suitable communication protocols based on the actual exchanges and avoid redundancy. In particular, these specialized protocols should not include redundant acknowledgements. For example, in a request-reply communication, it is generally considered redundant to acknowledge the request message, because the reply message serves as an acknowledgement. If a more specialized protocol requires sender acknowledgement or any other particular characteristics, these are supplied with the specialized operations. The idea is to add specialized functions only where they are needed, with a view to achieving protocols that use a minimum of message exchanges.

4.2 The API for the Internet protocols

In this section, we discuss the general characteristics of interprocess communication and then discuss the Internet protocols as an example, explaining how programmers can use them, either by means of UDP messages or through TCP streams.

Section 4.2.1 revisits the message communication operations *send* and *receive* introduced in Section 2.3.2 with a discussion of how they synchronize with one another and how message destinations are specified in a distributed system. Section 4.2.2 introduces *sockets*, which are used in the application programming interface to UDP and TCP. Section 4.2.3 discusses UDP and its API in Java. Section 4.2.4 discusses TCP and its API in Java. The APIs for Java are object-oriented but are similar to the ones designed originally in the Berkeley BSD 4.x UNIX operating system and discussed in Section 4.6. Readers studying the programming examples in this section should consult the on-line Java documentation or Flanagan [1997] for the full specification of the classes discussed, which are in the package *java.net*.

4.2.1 The characteristics of interprocess communication

Message passing between a pair of processes can be supported by two message communication operations: *send* and *receive*, defined in terms of destinations and messages. In order for one process to communicate with another, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message. This activity involves the communication of data from the sending process to the receiving process and may involve the synchronization of the two processes. Section 4.2.3 gives definitions for the *send* and *receive* operations in the Java API for the Internet protocols.

Synchronous and asynchronous communication ◊ A queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues. Communication between the sending and receiving process may be either synchronous or asynchronous. In the *synchronous* form of communication, the sending and receiving processes synchronize at every message. In this case, both *send* and *receive* are *blocking* operations. Whenever a *send* is issued the sending process is blocked until the corresponding *receive* is issued. Whenever a *receive* is issued the process blocks until a message arrives.

In the *asynchronous* form of communication, the use of the *send* operation is *non-blocking* in that the sending process is allowed to proceed as soon as the message has been copied to a local buffer, and the transmission of the message proceeds in parallel with the sending process. The *receive* operation can have blocking and non-blocking variants. In the non-blocking variant, the receiving process proceeds with its program after issuing a *receive* operation, which provides a buffer to be filled in the background, but it must separately receive notification that its buffer has been filled, by polling or interrupt.

In a system environment such as Java, which supports multiple threads in a single process, the blocking *receive* has few disadvantages, for it can be issued by one thread while other threads in the process remain active, and the simplicity of synchronizing the receiving threads with the incoming message is a substantial advantage. Non-blocking communication appears to be more efficient, but it involves extra complexity in the receiving process associated with the need to acquire the incoming message out of its flow of control. For these reasons, current systems do not generally provide the non-blocking form of *receive*.

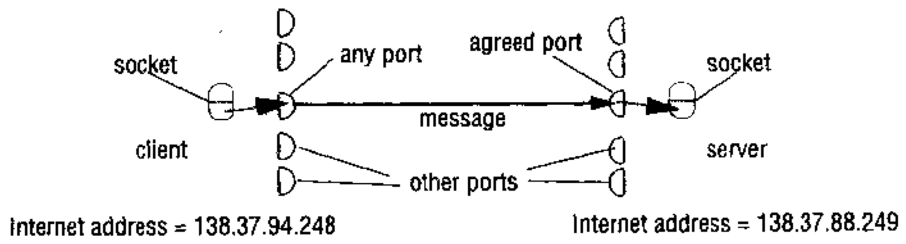
Message destinations ◊ Chapter 3 explains that in the Internet protocols, messages are sent to *(Internet address, local port)* pairs. A local port is a message destination within a computer, specified as an integer. A port has exactly one receiver but can have many senders. Processes may use multiple ports from which to receive messages. Any process that knows the number of a port can send a message to it. Servers generally publicize their port numbers for use by clients.

If the client uses a fixed Internet address to refer to a service, then that service must always run on the same computer for its address to remain valid. This can be avoided by using one of the following approaches to providing location transparency:

- Client programs refer to services by name and use a name server or binder (see Section 5.2.5) to translate their names into server locations at run time. This allows services to be relocated but not to migrate – to be moved while the system is running.
- The operating system, for example Mach (see Chapter 18), provides location-independent identifiers for message destinations, mapping them onto a lower-level address in order to deliver messages to ports, allowing service migration and relocation.

An alternative to ports is that messages should be addressed to processes, which was the case in the V system [Cheriton 1984]. However, ports have the advantage that they

Figure 4.2 Sockets and ports


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

provide several alternative points of entry to a receiving process. In some applications, it is very useful to be able to deliver the same message to the members of a set of processes. Therefore, some IPC systems provide the ability to send messages to groups of destinations – either processes or ports. For example, Chorus [Rozier *et al.* 1990] provides groups of ports.

Reliability ◊ Chapter 2 defines reliable communication in terms of validity and integrity. As far as the validity property is concerned, a point-to-point message service can be described as reliable if messages are guaranteed to be delivered despite a 'reasonable' number of packets being dropped or lost. In contrast, a point-to-point message service can be described as unreliable if messages are not guaranteed to be delivered in the face of even a single packet dropped or lost. For integrity, messages must arrive uncorrupted and without duplication.

Ordering ◊ Some applications require that messages be delivered in *sender order* – that is, the order in which they were transmitted by the sender. The delivery of messages out of sender order is regarded as a failure by such applications.

4.2.2 Sockets

Both forms of communication (UDP and TCP) use the *socket* abstraction, which provides an endpoint for communication between processes. Sockets originate from BSD UNIX but are also present in most versions of UNIX, including Linux as well as Windows NT and the Macintosh OS. Interprocess communication consists of transmitting a message between a socket in one process and a socket in another process, as illustrated in Figure 4.2. For a process to receive messages, its socket must be bound to a local port and one of the Internet addresses of the computer on which it runs. Messages sent to a particular Internet address and port number can be received only by a process whose socket is associated with that Internet address and port number. Processes may use the same socket for sending and receiving messages. Each computer has a large number (2^{16}) of possible port numbers for use by local processes for receiving messages. Any process may make use of multiple ports to receive messages, but a process cannot share ports with other processes on the same computer. Processes using IP multicast are an exception in that they do share ports – see Section 4.5.1. However, any number of processes may send messages to the same port. Each socket is associated with a particular protocol – either UDP or TCP.

Java API for Internet addresses ◊ As the IP packets underlying UDP and TCP are sent to Internet addresses, Java provides a class, *InetAddress*, that represents Internet addresses. Users of this class refer to computers by Domain Name Service (DNS) hostnames (see Section 3.4.7). For example, instances of *InetAddress* that contain Internet addresses can be created by calling a static method of *InetAddress*, giving a DNS hostname as argument. The method uses the DNS to get the corresponding Internet address. For example, to get an object representing the Internet address of the host whose DNS name is *bruno.dcs.qmw.ac.uk*, use:

InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmw.ac.uk");

This method can throw an *UnknownHostException*. Note that the user of the class does not need to state the explicit value of an Internet address. In fact, the class encapsulates the details of the representation of Internet addresses. Thus the interface for this class is not dependent on the number of bytes needed to represent Internet address – 4 bytes in IPv4 and 16 bytes in IPv6.

4.2.3 UDP datagram communication

A datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or retries. If a failure occurs, the message may not arrive. A datagram is transmitted between processes when one process *sends* it and another *receives* it. Any process needing to send or receive messages must first create a socket bound to an Internet address and local port. A server will bind its socket to a *server port* – one that it makes known to clients so that they can send messages to it. A client binds its socket to any free local port. The *receive* method returns the Internet address and port of the sender, in addition to the message, allowing the recipient to send a reply.

The following paragraphs discuss some issues relating to datagram communication:

Message size: The receiving process needs to specify an array of bytes of a particular size in which to receive a message. If the message is too big for the array, it is truncated on arrival. The underlying IP protocol allows packet lengths of up to 2^{16} bytes, which includes the headers as well as the message. However, most environments impose a size restriction of 8 kilobytes. Any application requiring messages larger than the maximum must fragment them into chunks of that size. Generally, an application will decide on a size that is not excessively large but is adequate for its intended use.

Blocking: UDP datagram communication uses non-blocking *sends* and blocking *receives*. The *send* operation returns when it has handed the message to the underlying UDP and IP protocols, which are responsible for transmitting it to its destination. On arrival, the message is placed in a queue for the socket that is bound to the destination port. The message can be collected from the queue by an outstanding or future invocation of *receive* on that socket. Messages are discarded at the destination if no process already has a socket bound to the destination port.

The method *receive* blocks until a datagram is received, unless a timeout has been set on the socket. If the process that invokes the *receive* method has other work to do while waiting for the message, it should arrange to use a separate thread.

Threads are discussed in Chapter 6. For example, when a server receives a message from a client, the message may specify work to do, in which case the server will use separate threads to do the work and to wait for messages from other clients.

Timeouts: The *receive* that blocks for ever is suitable for use by a server that is waiting to receive requests from its clients. But in some programs, it is not appropriate that a process that has used a *receive* operation should wait indefinitely in situations where the potential sending process has crashed or the expected message has been lost. To allow for such requirements, timeouts can be set on sockets. Choosing an appropriate timeout interval is difficult, but it should be fairly large in comparison with the time required to transmit a message.

Receive from any: The *receive* method does not specify an origin for messages. Instead, an invocation of *receive* gets a message addressed to its socket from any origin. The *receive* method returns the Internet address and local port of the sender, allowing the recipient to check where it came from. It is possible to connect a datagram socket to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that address.

Failure model ◊ Chapter 2 presents a failure model for communication channels and defines reliable communication in terms of two properties: integrity and validity. The integrity property requires that messages not be corrupted or duplicated. The use of a checksum ensures that there is a negligible probability that any message received is corrupted. The failure model can be used to provide a failure model for UDP datagrams, which suffer from the following failures:

Omission failures: Messages may be dropped occasionally, either because of a checksum error or because no buffer space is available at the source or destination. To simplify the discussion, we regard send-omission and receive-omission failures (see Figure 2.11) as omission failures in the communication channel.

Ordering: Messages can sometimes be delivered out of sender order.

Applications using UDP datagrams are left to provide their own checks to achieve the quality of reliable communication they require. A reliable delivery service may be constructed from one that suffers from omission failures by the use of acknowledgements. Section 4.4 discusses how reliable request-reply protocols for client-server communication may be built over UDP.

Use of UDP ◊ For some applications, it is acceptable to use a service that is liable to occasional omission failures. For example, the Domain Naming Service, which looks up DNS names in the Internet, is implemented over UDP. UDP datagrams are sometimes an attractive choice because they do not suffer from the overheads associated with guaranteed message delivery. There are three main sources of overhead:

1. the need to store state information at source and destination;
2. the transmission of extra messages; and
3. latency for the sender.

The reasons for these overheads are discussed in Section 4.2.4.

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
      try {
         DatagramSocket 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()));
         aSocket.close();
      }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
      }catch (IOException e){System.out.println("IO: " + e.getMessage());}
   }
}
```

Java API for UDP datagrams ◊ The Java API provides datagram communication by means of two classes: *DatagramPacket* and *DatagramSocket*.

DatagramPacket: This class provides a constructor that makes an instance out of an array of bytes comprising a message, the length of the message and the Internet address and local port number of the destination socket, as follows:

Datagram packet

array of bytes containing message	length of message	Internet address	port number

Instances of *DatagramPacket* may be transmitted between processes when one process *sends* it and another *receives* it.

This class provides another constructor for use when receiving a message. Its arguments specify an array of bytes in which to receive the message and the length of the array. A received message is put in the *DatagramPacket* together with its length and the Internet address and port of the sending socket. The message can be retrieved from the *DatagramPacket* by means of the method *getData*. The methods *getPort* and *getAddress* access the port and Internet address.

DatagramSocket: This class supports sockets for sending and receiving UDP datagrams. It provides a constructor that takes a port number as argument, for use by

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[]){
      try{
        DatagramSocket 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());}
    }
}
```

processes that need to use a particular port. It also provides a no-argument constructor that allows the system to choose a free local port. These constructors can throw a *SocketException* if the port is already in use or if a reserved port (a number below 1024) is specified when running over UNIX.

The class *DatagramSocket* provides methods that include the following:

send and *receive* These methods are for transmitting datagrams between a pair of sockets. The argument of *send* is an instance of *DatagramPacket* containing a message and its destination. The argument of *receive* is an empty *DatagramPacket* in which to put the message, its length and origin. The methods *send* and *receive* can throw *IOExceptions.*

setSoTimeout This method allows a timeout to be set. With a timeout set, the *receive* method will block for the time specified and then throw an *InterruptedIOException*.

connect This method is used for connecting it to a particular remote port and Internet address, in which case the socket is only able to send messages to and receive messages from that address.

Figure 4.3 shows the program for a client that creates a socket, sends a message to a server at port 6789 and then waits to receive a reply. The arguments of the *main* method supply a message and the DNS hostname of the server. The message is converted to an array of bytes, and the DNS hostname is converted to an Internet address. Figure 4.4 shows the program for the corresponding server, which creates a socket bound to its server port (6789) then repeatedly waits to receive a request message from a client, to which it replies by sending back the same message.

4.2.4 TCP stream communication

The API to the TCP protocol, which originates from BSD 4.x UNIX, provides the abstraction of a stream of bytes to which data may be written and from which data may be read. The following characteristics of the network are hidden by the stream abstraction:

Message sizes: The application can choose how much data it writes to a stream or reads from it. It may deal in very small or very large sets of data. The underlying implementation of a TCP stream decides how much data to collect before transmitting it as one or more IP packets. On arrival, the data is handed to the application as requested. Applications can, if necessary, force data to be sent immediately.

Lost messages: The TCP protocol uses an acknowledgement scheme. As an example of a simple scheme (which is not used in TCP), the sending end keeps a record of each IP packet sent and the receiving end acknowledges all the arrivals. If the sender does not receive an acknowledgement within a timeout, it retransmits the message. The more sophisticated sliding window scheme [Comer 1991] cuts down on the number of acknowledgement messages required.

Flow control: The TCP protocol attempts to match the speeds of the processes that read from and write to a stream. If the writer is too fast for the reader, then it is blocked until the reader has consumed sufficient data.

Message duplication and ordering: Message identifiers are associated with each IP packet, which enables the recipient to detect and reject duplicates, or to reorder messages that do not arrive in sender order.

Message destinations: A pair of communicating processes establish a connection before they can communicate over a stream. Once a connection is established, the processes simply read from and write to the stream without needing to use Internet addresses and ports. Establishing a connection involves a *connect* request from client to server followed by an *accept* request from server to client before any communication can take place. This could be a considerable overhead for a single client-server request and reply.

The API for stream communication assumes that when a pair of processes are establishing a connection, one of them plays the client role and the other plays the server role, but thereafter they could be peers. The client role involves creating a stream socket bound to any port and then making a *connect* request asking for a connection to a server at its server port. The server role involves creating a listening socket bound to a server port and waiting for clients to request connections. The listening socket maintains a queue of incoming connection requests. In the socket model, when the server *accepts* a connection, a new stream socket is created for the server to communicate with a client, meanwhile retaining its socket at the server port for listening for *connect* requests from other clients. Further details of the *connect* and *accept* operations are described in the UNIX case study at the end of this chapter.

The pair of sockets in client and server are connected by a pair of streams, one in each direction. Thus each socket has an input stream and an output stream. One of the pair of processes can send information to the other by writing to its output stream, and the other process obtains the information by reading from its input stream.

When an application *closes* a socket, this indicates that it will not write any more data to its output stream. Any data in the output buffer is sent to the other end of the stream and put in the queue at the destination socket with an indication that the stream is broken. The process at the destination can read the data in the queue, but any further reads after the queue is empty will result in an indication of end of stream. When a process exits or fails, all of its sockets are eventually closed and any process attempting to communicate with it will discover that its connection has been broken.

The following paragraphs address some outstanding issues related to stream communication:

Matching of data items: Two communicating processes need to agree as to the contents of the data transmitted over a stream. For example, if one process writes an *int* followed by a *double* to a stream, then the reader at the other end must read an *int* followed by a *double*. When a pair of processes do not cooperate correctly in their use of a stream, the reading process may experience errors when interpreting the data or may block due to insufficient data in the stream.

Blocking: The data written to a stream is kept in a queue at the destination socket. When a process attempts to read data from an input channel, it will get data from the queue or it will block until data becomes available. The process that writes data to a stream may be blocked by the TCP flow control mechanism if the socket at the other end is queuing as much data as the protocol allows.

Threads: When a server accepts a connection, it generally creates a new thread in which to communicate with the new client. The advantage of using a separate thread for each client is that the server can block when waiting for input without delaying other clients. In an environment in which threads are not provided, an alternative is to test whether input is available from a stream before attempting to read it; for example, in a UNIX environment the *select* system call may be used for this purpose.

Failure model ◊ To satisfy the integrity property of reliable communication, TCP streams use checksums to detect and reject corrupt packets and sequence numbers to detect and reject duplicate packets. For the sake of the validity property, TCP streams use timeouts and retransmissions to deal with lost packets. Therefore, messages are guaranteed to be delivered even when some of the underlying packets are lost.

But if the packet loss over a connection passes some limit or the network connecting a pair of communicating processes is severed or becomes severely congested, the TCP software responsible for sending messages will receive no acknowledgements and after a time will declare the connection to be broken. Thus TCP does not provide reliable communication, because it does not guarantee to deliver messages in the face of all possible difficulties.

When a connection is broken, a process using it will be notified if it attempts to read or write. This has the following effects:

- the processes using the connection cannot distinguish between network failure and failure of the process at the other end of the connection;

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
        try{
          int serverPort = 7896;
          Socket 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) ;
          s.close();
        }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());}
     }
}
```

- the communicating processes cannot tell whether their recent messages have been received or not.

Use of TCP ◊ Many frequently used services run over TCP connections, with reserved port numbers. These include the following:

HTTP: The hypertext transfer protocol is used for communication between web browsers and web servers; it is discussed later in this chapter.

FTP: The file transfer protocol allows directories on a remote computer to be browsed and files to be transferred from one computer to another over a connection.

Telnet: telnet provides access by means of a terminal session to a remote computer.

SMTP: The simple mail transfer protocol is used to send mail between computers.

Java API for TCP streams ◊ The Java interface to TCP streams is provided in the classes *ServerSocket* and *Socket*.

ServerSocket: This class is intended for use by a server to create a socket at a server port for listening for *connect* requests from clients. Its *accept* method gets a *connect* request from the queue, or if the queue is empty, it blocks until one arrives. The result of executing *accept* is an instance of *Socket* – a socket for giving access to streams for communicating with the client.

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());}
   }
}
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);
           clientSocket.close();
        } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
        } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
      }
 }
```

Socket: This class is for use by a pair of processes with a connection. The client uses a constructor to create a socket, specifying the DNS hostname and port of a server. This constructor not only creates a socket associated with a local port but also *connects* it to the specified remote computer and port number. It can throw an *UnknownHostException* if the hostname is wrong or an *IOException* if an IO error occurs.

The *Socket* class provides methods *getInputStream* and *getOutputStream* for accessing the two streams associated with a socket. The return types of these methods are *InputStream* and *OutputStream*, respectively – abstract classes that define methods for reading and writing bytes. The return values can be used as the arguments of constructors for suitable input and output streams. Our example uses *DataInputStream* and *DataOutputStream*, which allow binary representations of primitive data types to be read and written in a machine-independent manner.

Figure 4.5 shows a client program in which the arguments of the *main* method supply a message and the DNS hostname of the server. The client creates a socket bound to the hostname and server port 7896. It makes a *DataInputStream* and a *DataOutputStream* from the socket's input and output streams then writes the message to its output stream and waits to read a reply from its input stream. The server program in Figure 4.6 opens a server socket on its server port (7896) and listens for *connect* requests. When one arrives, it makes a new thread in which to communicate with the client. The new thread creates a *DataInputStream* and a *DataOutputStream* from its socket's input and output streams and then waits to read a message and write it back.

As our message consists of a string, the client and server processes use the method *writeUTF* of *DataOutputStream* to write it to the output stream and the method *readUTF* of *DataInputStream* to read it from the input stream. UTF is an encoding that represents strings in a particular format, which is described in Section 4.3.

When a process has closed its socket, it will no longer be able to use its input and output streams. The process to which it has sent data can read the data in its queue, but any further reads after the queue is empty will result in an *EOFException*. Attempts to use a closed socket or to write to a broken stream result in an *IOException*.

4.3 External data representation and marshalling

The information stored in running programs is represented as data structures – for example by sets of interconnected objects – whereas the information in messages consists of sequences of bytes. Irrespective of the form of communication used, the data structures must be flattened (converted to a sequence of bytes) before transmission and rebuilt on arrival. The individual primitive data items transmitted in messages can be data values of many different types, and not all computers store primitive values such as integers in the same order. The representation of floating-point numbers also differs between architectures. Another issue is the set of codes used to represent characters: for example, UNIX systems use ASCII character coding, taking one byte per character, whereas the Unicode standard allows for the representation of texts in many different languages and takes two bytes per character. There are two variants for the ordering of integers: the so-called *big-endian* order, in which the most significant byte comes first; and *little-endian* order, in which it comes last.

One of the following methods can be used to enable any two computers to exchange data values:

- The values are converted to an agreed external format before transmission and converted to the local form on receipt; if the two computers are known to be the same type, the conversion to external format can be omitted.
- The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary.

Note, however, that bytes themselves are never altered during transmission. To support RMI or RPC, any data type that can be passed as an argument or returned as a result must be able to be flattened and the individual primitive data values represented in an agreed format. An agreed standard for the representation of data structures and primitive values is called an *external data representation*.

Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message. *Unmarshalling* is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination. Thus marshalling consists of the translation of structured data items and primitive values into an external data representation. Similarly, unmarshalling consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.

Two alternative approaches to external data representation and marshalling are discussed:

- CORBA's common data representation, which is concerned with an external representation for the structured and primitive types that can be passed as arguments and results of remote method invocations in CORBA. It can be used by a variety of programming languages (see Chapter 17).
- Java's object serialization, which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.

In both cases, the marshalling and unmarshalling activities are intended to be carried out by a middleware layer without any involvement on the part of the application programmer. Because marshalling requires the consideration of all the finest details of the representation of the primitive components of composite objects, the process is likely to be error-prone if carried out by hand. Efficiency is another issue that can be addressed in the design of automatically generated marshalling procedures.

The two approaches discussed here marshal the primitive data types into a binary form. An alternative approach is to marshal all of the objects to be transmitted into ASCII text, which is relatively simple to implement, but the marshalled form will generally be longer. The HTTP protocol, which is described in Section 4.4, is an example of the latter approach.

Although we are interested in the use of marshalling for the arguments and results of RMIs and RPCs, it does have a more general use for converting data structures or objects into a form suitable for transmission in messages or storing in files.

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

4.3.1 CORBA's Common Data Representation (CDR)

CORBA CDR is the external data representation defined with CORBA 2.0 [OMG 1998a]. CDR can represent all of the data types that can be used as arguments and return values in remote invocations in CORBA. These consist of 15 primitive types, which include *short* (16-bit), *long* (32-bit), *unsigned short*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit), and *any* (which can represent any basic or constructed type); together with a range of composite types, which are described in Figure 4.7. Each argument or result in a remote invocation is represented by a sequence of bytes in the invocation or result message.

Primitive types: CDR defines a representation for both big-endian and little-endian orderings. Values are transmitted in the sender's ordering, which is specified in each message. The recipient translates if it requires a different ordering. For example, a 16-bit *short* occupies two bytes in the message and for big-endian ordering, the most significant bits occupy the first byte and the least significant bits occupy the second byte. Each primitive value is placed at an index in the sequence of bytes according to its size. Suppose that the sequence of bytes is indexed from zero upwards. Then a primitive value of size n bytes (where $n = 1, 2, 4$ or 8) is appended to the sequence at an index that is a multiple of n in the stream of bytes. Floating-point values follow the IEEE standard – in which the sign, exponent and fractional part are in bytes $0–n$ for big-endian ordering and the other way round for little-endian. Characters are represented by a code set agreed between client and server.

Constructed types: The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order, as shown in Figure 4.7.

Figure 4.8 shows a message in CORBA CDR that contains the three fields of a *struct* whose respective types are *string*, *string* and *unsigned long*. The figure shows the sequence of bytes with four bytes in each row. The representation of each string consists of an *unsigned long* representing its length followed by the characters in the string. For simplicity, we assume that each character occupies just one byte. Variable length data is padded with zeros so that it has a standard form, enabling marshalled data or its

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}

checksum to be compared. Note that each *unsigned long*, which occupies four bytes, starts at an index that is a multiple of four. The figure does not distinguish between the big- and little-endian ordering. Although the example in Figure 4.8 is simple, CORBA CDR can represent any data structure that can be composed from the primitive and constructed types, but without using pointers.

Another example of an external data representation is the Sun XDR standard, which is specified in RFC 1832 [Srinivasan 1995b] and described in www.cdk3.net/ipc. It was developed by Sun for use in the messages exchanged between clients and servers in Sun NFS (see Chapter 8).

The type of a data item is not given with the data representation in the message in either the CORBA CDR or the Sun XDR standard. This is because it is assumed that the sender and recipient have common knowledge of the order and types of the data items in a message. In particular for RMI or RPC, each method invocation passes arguments of particular types, and the result is a value of a particular type.

Marshalling in CORBA ◊ Marshalling operations can be generated automatically from the specification of the types of data items to be transmitted in a message. The types of the data structures and the types of the basic data items are described in CORBA IDL (see Section 17.2.3), which provides a notation for describing the types of the arguments and results of RMI methods. For example, we might use CORBA IDL to describe the data structure in the message in Figure 4.8 as follows:

```
struct Person{
    string name;
    string place;
    long year;
};
```

The CORBA interface compiler (see Chapter 5) generates appropriate marshalling and unmarshalling operations for the arguments and results of remote methods from the definitions of the types of their parameters and results.

4.3.2 Java object serialization

In Java RMI, both objects and primitive data values may be passed as arguments and results of method invocations. An object is an instance of a Java class. For example, the Java class equivalent to the *Person struct* defined in CORBA IDL might be:

```
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;
    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
    }
    // followed by methods for accessing the instance variables
}
```

The above class states that it implements the *Serializable* interface, which has no methods. Stating that a class implements the *Serializable* interface (which is provided in the *java.io* package) has the effect of allowing its instances to be serialized.

In Java, the term *serialization* refers to the activity of flattening an object or a connected set of objects into a serial form that is suitable for storing on disk or transmitting in a message, for example as an argument or result of an RMI. Deserialization consists of restoring the state of an object or a set of objects from their serialized form. It is assumed that the process that does the deserialization has no prior knowledge of the types of the objects in the serialized form. Therefore, some information about the class of each object is included in the serialized form. This information enables the recipient to load the appropriate class when an object is deserialized.

The information about a class consists of the name of the class and a version number. The version number is intended to change when major changes are made to the class. It can be set by the programmer or calculated automatically as a hash of the name of the class, its instance variables, methods and interfaces. The process that deserializes an object can check that it has the correct version of the class.

Java objects can contain references to other objects. When an object is serialized, all the objects that it references are serialized together with it to ensure that when the object is reconstructed, all of its references can be fulfilled at the destination. References are serialized as *handles* – in this case, the handle is a reference to an object within the serialized form, for example the next number in a sequence of positive integers. The serialization procedure must ensure that there is a 1-1 correspondence between object references and handles. It must also ensure that each object is written once only – on the second or subsequent occurrence of an object, the handle is written instead of the object.

To serialize an object, its class information is written out, followed by the types and names of its instance variables. If the instance variables belong to new classes, then their class information must also be written out, followed by the types and names of their instance variables. This recursive procedure continues until the class information and types and names of instance variables of all of the necessary classes have been written

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

out. Each class is given a handle, and no class is written more than once to the stream of bytes – the handles being written instead where necessary.

The contents of the instance variables that are primitive types, such as integers, chars, booleans, bytes and longs, are written in a portable binary format using methods of the *ObjectOutputStream* class. Strings and characters are written by its method called *writeUTF* using Universal Transfer Format (UTF), which enables ASCII characters to be represented unchanged (in one byte), whereas Unicode characters are represented by multiple bytes. Strings are preceded by the number of bytes they occupy in the stream.

As an example, consider the serialization of the following object:

Person p = new Person("Smith", "London", 1934);

The serialized form is illustrated in Figure 4.9, which omits the values of the handles and of the type markers that indicate the objects, classes, strings and other objects in the full serialized form. The first instance variable (1934) is an integer that has a fixed length; the second and third instance variables are strings and are preceded by their lengths.

To make use of Java serialization, for example to serialize the *Person* object, create an instance of the class *ObjectOutputStream* and invoke its *writeObject* method, passing the *Person* object as argument. To deserialize an object from a stream of data, open an *ObjectInputStream* on the stream and use its *readObject* method to reconstruct the original object. The use of this pair of classes is similar to the use of *DataOutputStream* and *DataInputStream* illustrated in Figures 4.5 and 4.6.

Serialization and deserialization of the arguments and results of remote invocations are generally carried out automatically by the middeware, without any participation by the application programmer. If necessary, programmers with special requirements may write their own version of the methods that read and write objects. To find out how to do this and to get further information about serialization in Java, read the tutorial on object serialization [java.sun.com II]. Another way in which a programmer may modify the effects of serialization is by declaring variables that should not be serialized as *transient*. Examples of things that should not be serialized are references to local resources such as files and sockets.

The use of reflection ◊ The Java language supports *reflection* – the ability to enquire about the properties of a class, such as the names and types of its instance variables and methods. It also enables classes to be created from their names, and a constructor with given argument types to be created for a given class. Reflection makes it possible to do serialization and deserialization in a completely generic manner. This means that there is no need to generate special marshalling functions for each type of object as described above for CORBA. To find out more about reflection, see Flanagan [1997].

Java object serialization uses reflection to find out the class name of the object to be serialized and the names, types and values of its instance variables. That is all that is needed for the serialized form.

For deserialization, the class name in the serialized form is used to create a class. This is then used to create a new constructor with argument types corresponding to those specified in the serialized form. Finally, the new constructor is used to create a new object with instance variables whose values are read from the serialized form.

4.3.3 Remote object references

When a client invokes a method in a remote object, an invocation message is sent to the server process that hosts the remote object. This message needs to specify which particular object is to have its method invoked. A *remote object reference* is an identifier for a remote object that is valid throughout a distributed system. A remote object reference is passed in the invocation message to specify which object is to be invoked. Chapter 5 explains that remote object references are also passed as arguments and returned as results of remote method invocations, that each remote object has a single remote object reference and that remote object references can be compared to see whether they refer to the same remote object. We now discuss the external representation of remote object references.

Remote object references must be generated in a manner that ensures uniqueness over space and time. In general, there may be many processes hosting remote objects, so remote object references must be unique among all of the processes in the various computers in a distributed system. Even after the remote object associated with a given remote object reference is deleted, it is important that the remote object reference is not reused, because its potential invokers may retain obsolete remote object references. Any attempt to invoke a deleted object should produce an error rather than allow access to a different object.

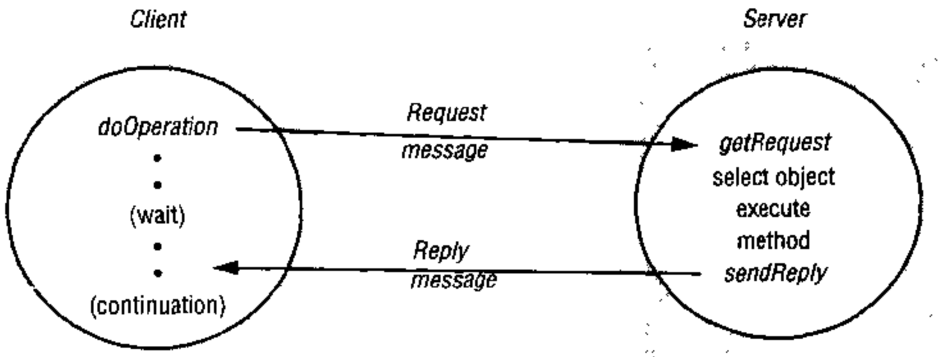
There are several ways to ensure that a remote object reference is unique. One way is to construct a remote object reference by concatenating the Internet address of its computer and the port number of the process that created it with the time of its creation and a local object number. The local object number is incremented each time an object is created in that process.

The port number and time together produce a unique process identifier on that computer. With this approach, remote object references might be represented with a

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



format such as that shown in Figure 4.10. In the simplest implementations of RMI, remote objects live only in the process that created them and survive only as long as that process continues to run. In such cases, the remote object reference can be used as an address of the remote object. In other words, invocation messages are sent to the Internet address in the remote reference and to the process on that computer using the given port number.

To allow remote objects to be relocated in a different process on a different computer, the remote object reference should not be used as the address of the remote object. Section 17.2.4 discusses a form of remote object reference that allows objects to be activated in different servers throughout its lifetime.

The last field of the remote object reference shown in Figure 4.10 contains some information about the interface of the remote object, for example the interface name. This information is relevant to any process that receives a remote object reference as an argument or result of a remote invocation, because it needs to know about the methods offered by the remote object. This point is explained again in Section 5.2.5.

4.4 Client-server communication

This form of communication is designed to support the roles and message exchanges in typical client-server interactions. In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server. It can also be reliable because the reply from the server is effectively an acknowledgement to the client. Asynchronous request-reply communication is an alternative that may be useful in situations where clients can afford to retrieve replies later – see Section 6.5.2.

The client-server exchanges are described in the following paragraphs in terms of the *send* and *receive* operations in the Java API for UDP datagrams, although many current implementations use TCP streams. A protocol built over datagrams avoids unnecessary overheads associated with the TCP stream protocol. In particular:

- acknowledgements are redundant, since requests are followed by replies;

- establishing a connection involves two extra pairs of messages in addition to the pair required for a request and a reply;
- flow control is redundant for the majority of invocations, which pass only small arguments and results.

The request-reply protocol ◊ The following protocol is based on a trio of communication primitives: *doOperation*, *getRequest* and *sendReply*, as shown in Figure 4.11. Most RMI and RPC systems expect to be supported by a similar protocol. The one we describe here is tailored for supporting RMI in that it passes a remote object reference for the object whose method is to be invoked in the request message.

This specially designed request-reply protocol matches requests to replies. It may be designed to provide certain delivery guarantees. If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message. Figure 4.12 outlines the three communication primitives.

The *doOperation* method is used by clients to invoke remote operations. Its arguments specify the remote object and which method to invoke, together with additional information (arguments) required by the method. Its result is an RMI reply. It is assumed that the client calling *doOperation* marshals the arguments into an array of bytes and unmarshals the results from the array of bytes that is returned. The first argument of *doOperation* is an instance of the class *RemoteObjectRef*, which represents references for remote objects, for example as shown in Figure 4.10. This class provides methods for getting the Internet address and port of the server of the remote object. The *doOperation* method sends a request message to the server whose Internet address and port are specified in the remote object reference given as argument. After sending the request message, *doOperation* invokes *receive* to get a reply message, from which it extracts the result and returns it to the caller. The caller of *doOperation* is blocked until the remote object in the server performs the requested operation and transmits a reply message to the client process.

GetRequest is used by a server process to acquire service requests as shown in Figure 4.11. When the server has invoked the method in the specified object it then uses *sendReply* to send the reply message to the client. When the reply message is received by the client the original *doOperation* is unblocked and execution of the client program continues.

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*

The information to be transmitted in a request message or a reply message is shown in Figure 4.13. The first field indicates whether the message is a *Request* or a *Reply* message. The second field *requestId* contains a message identifier. A *doOperation* in the client generates a *requestId* for each request message, and the server copies them into the corresponding reply messages. This enables *doOperation* to check that a reply message is the result of the current request, not from a delayed earlier call. The third field is a remote object reference marshalled in the form shown in Figure 4.10. The fourth field is an identifier for the method to be invoked, for example, the methods in an interface might be numbered 1, 2, 3, ... ; if client and server use a common language that supports reflection, then a representation of the method itself may be put in this field – in Java an instance of *Method* may be put in this field.

Message identifiers: Any scheme that involves the management of messages to provide additional properties such as reliable message delivery or request-reply communication requires that each message have a unique message identifier by which it may be referenced. A message identifier consists of two parts:

1. a *requestId*, which is taken from an increasing sequence of integers by the sending process; and
2. an identifier for the sender process, for example its port and Internet address.

The first part makes the identifier unique to the sender, and the second part makes it unique in the distributed system. (The second part can be obtained independently, for example if UDP is in use, from the message received).

When the value of the *requestId* reaches the maximum value for an unsigned integer (for example, $2^{32} - 1$) it is reset to zero. The only restriction here is that the lifetime of a message identifier should be much less than the time taken to exhaust the values in the sequence of integers.

Failure model of the request-reply protocol ◊ If the three primitives *doOperation*, *getRequest* and *sendReply* are implemented over UDP datagrams, then they suffer from the same communication failures. That is:

- they suffer from omission failures;
- messages are not guaranteed to be delivered in sender order.

In addition, the protocol can suffer from the failure of processes (see Section 2.3.2). We assume that processes have crash failures. That is, when they halt, they remain halted – they do not produce byzantine behaviour.

To allow for occasions when a server has failed or a request or reply message is dropped, *doOperation* uses a timeout when it is waiting to get the server's reply message. The action taken when a timeout occurs depends upon the delivery guarantees to be offered.

Timeouts: There are various options as to what *doOperation* can do after a timeout. The simplest option is to return immediately from *doOperation* with an indication to the client that the *doOperation* has failed. This is not the usual approach – the timeout may have been due to the request or reply message getting lost – and in the latter case, the operation will have been performed. To compensate for the possibility of lost messages, *doOperation* sends the request message repeatedly until either it gets a reply or else it is reasonably sure that the delay is due to lack of response from the server rather than to lost messages. Eventually, when *doOperation* returns it will indicate to the client by an exception that no result was received.

Discarding duplicate request messages: In cases when the request message is retransmitted, the server may receive it more than once. For example, the server may receive the first request message but take longer than the client's timeout to execute the command and return the reply. This can lead to the server executing an operation more than once for the same request. To avoid this, the protocol is designed to recognize successive messages (from the same client) with the same request identifier and to filter out duplicates. If the server has not yet sent the reply, it need take no special action – it will transmit the reply when it has finished executing the operation.

Lost reply messages: If the server has already sent the reply when it receives a duplicate request it will need to execute the operation again to obtain the result, unless it has stored the result of the original execution. Some servers can execute their operations more than once and obtain the same results each time. An *idempotent operation* is an operation that can be performed repeatedly with the same effect as if it had been performed exactly once. For example, an operation to add an element to a set is an idempotent operation because it will always have the same effect on the set each time it is performed, whereas an operation to append an item to a sequence is not an idempotent operation, because it extends the sequence each time it is performed. A server whose operations are all idempotent need not take special measures to avoid executing its operations more than once.

History: For servers that require retransmission of replies without re-execution of operations, a history may be used. The term 'history' is used to refer to a structure that contains a record of (reply) messages that have been transmitted. An entry in a history contains a request identifier, a message and an identifier of the client to which it was sent. Its purpose is to allow the server to retransmit reply messages when client processes request them. A problem associated with the use of a history is its memory cost. A history will become very large unless the server can tell when the messages will no longer be needed for retransmission.

As clients can make only one request at a time, the server can interpret each request as an acknowledgement of its previous reply. Therefore the history need contain

Figure 4.14 RPC exchange protocols

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

only the last reply message sent to each client. However, the volume of reply messages in a server's history may be a problem when it has a large number of clients. In particular, when a client process terminates, it does not acknowledge the last reply it has received – messages in the history are therefore normally discarded after a limited period of time.

RPC exchange protocols: Three protocols, with differing semantics in the presence of communication failures, are used for implementing various types of RPC. They were originally identified by Spector [1982]:

- the *request (R)* protocol;
- the *request-reply (RR)* protocol;
- the *request-reply-acknowledge reply (RRA)* protocol.

The messages passed in these protocols are summarized in Figure 4.14. The R protocol may be used when there is no value to be returned from the procedure and the client requires no confirmation that the procedure has been executed. The client may proceed immediately after the request message is sent as there is no need to wait for a reply message. The RR protocol is useful for most client-server exchanges because it is based on the request-reply protocol. Special acknowledgement messages are not required, because a server's reply message is regarded as an acknowledgement of the client's request message. Similarly, a subsequent call from a client may be regarded as an acknowledgement of a server's reply message.

The RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply. The acknowledge reply message contains the *requestId* from the reply message being acknowledged. This will enable the server to discard entries from its history. The arrival of a *requestId* in an acknowledgement message will be interpreted as acknowledging the receipt of all reply messages with lower *requestId*s, so the loss of an acknowledgement message is harmless. Although the exchange involves an additional message, it need not block the client, as the acknowledgement may be transmitted after the reply has been given to the client, but it does use processing and network resources. Exercise 4.22 suggests an optimization to the RRA protocol.

Use of TCP streams to implement the request-reply protocol ◊ The section on datagrams mentioned that it is often difficult to decide on an appropriate size for the buffer in which to receive datagrams. In the request-reply protocol, this applies to the buffers used by the server to receive request messages and by the client to receive replies. The limited length of datagrams (usually 8 kilobytes) may not be regarded as adequate for use in transparent RMI systems, since the arguments or results of procedures may be of any size.

The desire to avoid implementing multi-packet protocols is one of the reasons for choosing to implement request-reply protocols over TCP streams, allowing arguments and results of any size to be transmitted. In particular, Java object serialization is a stream protocol that allows arguments and results to be sent over streams between client and server, making it possible for collections of objects of any size to be transmitted reliably. If the TCP protocol is used, it ensures that request and reply messages are delivered reliably, so there is no need for the request-reply protocol to deal with retransmission of messages and filtering of duplicates or with histories. In addition the flow-control mechanism allows large arguments and results to be passed without taking special measures to avoid overwhelming the recipient. Thus, the TCP protocol is chosen for implementing request-reply protocols because it can simplify their implementation. If successive requests and replies between the same client-server pair are sent over the same stream, the connection overhead need not apply to every remote invocation. Also the overhead due to acknowledgement messages is reduced when a reply message follows soon after a request message.

Sometimes, the application does not require all of the facilities offered by TCP, and a more efficient, specially tailored protocol can be implemented over UDP. For example, as we mentioned earlier Sun NFS does not require messages of unlimited size, since it transmits fixed-size file blocks between client and server. In addition to that, its operations are designed to be idempotent, so that it does not matter if operations are executed more than once in order to retransmit lost reply messages, making it unnecessary to maintain a history.

HTTP: an example of a request-reply protocol ◊ Chapter 1 introduced the Hypertext Transfer Protocol (HTTP) used by web browser clients to make requests to web servers and to receive replies from them. To recap, web servers manage resources implemented in different ways:

- as data, for example the text of an HTML page, an image or the class of an applet;
- as a program, for example *cgi* programs and servlets (see [java.sun.com III]) that can be run on the web server.

Client requests specify a URL that includes the DNS hostname of a web server and an optional port number on the web server as well as the identifier of a resource on that server.

HTTP is a protocol that specifies the messages involved in a request-reply exchange, the methods, arguments and results and the rules for representing (marshalling) them in the messages. It supports a fixed set of methods (*GET*, *PUT*, *POST*, *etcetera*) that are applicable to all of its resources. It is unlike the above protocols, where each object has its own methods. In addition to invoking methods on web resources, the protocol allows for content negotiation and password-style authentication.

Content negotiation: Clients' requests can include information as to what data representation they can accept (for example language or media type), enabling the server to choose the representation that is the most appropriate for the user.

Authentication: Credentials and challenges are used to support password-style authentication. On the first attempt to access a password-protected area, the server reply contains a challenge applicable to the resource. Chapter 7 explains challenges. When it receives a challenge, the client gets the user to type a name and password and submits the associated credentials with subsequent requests. HTTP is implemented over TCP. In the original version of the protocol, each client-server interaction consists of the following steps:

- the client requests and the server accepts a connection at the default server port or at a port specified in the URL;
- the client sends a request message to the server;
- the server sends a reply message to the client;
- the connection is closed.

However, the need to establish and close a connection for every request-reply exchange is expensive, both in overloading the server and in sending too many messages over the network. Bearing in mind that browsers generally make multiple requests to the same server, a later version of the protocol (HTTP 1.1: see RFC 2616 [Fielding *et al.* 1999]) uses *persistent connections* – connections that remain open over a series of request-reply exchanges between client and server. A persistent connection can be closed by client or server at any time by sending an indication to the other participant. Servers will close a persistent connection when it has been idle for a period of time. It is possible that a client may receive a message from the server saying that the connection is closed while it is in the middle of sending another request or requests. In such cases, the browser will resend the requests without user involvement, provided that the operations involved are idempotent. For example, the method GET described below is idempotent. Where non-idempotent operations are involved, the browser should consult the user as to what to do next.

Requests and replies are marshalled into messages as ASCII text strings, but resources can be represented as byte sequences and may be compressed. The use of text in the external data representation has simplified the use of HTTP for application programmers who work directly with the protocol. In this context, a textual representation does not add much to the length of the messages.

Resources implemented as data are supplied as MIME-like structures in arguments and results. Multipurpose Internet Mail Extensions (MIME) is a standard for sending multipart data containing, for example, text, images and sound in email messages. Data is prefixed with its *Mime type* so that the recipient will know how to handle it. A *Mime type* specifies a type and a subtype, for example *text/plain*, *text/html*, *image/gif*, *image/jpeg*. Clients can also specify the Mime types that they are willing to accept.

HTTP methods: Each client request specifies the name of a method to be applied to a resource at the server and the URL of that resource. The reply reports on the status of the request. Requests and replies may also contain resource data, the contents of a form

Figure 4.15 HTTP *request* message

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

or the output of a program resource run on the web server. The methods include the following:

GET: requests the resource whose URL is given as argument. If the URL refers to data, then the web server replies by returning the data identified by that URL. If the URL refers to a program, then the web server runs the program and returns its output to the client. Arguments may be added to the URL; for example, GET can be used to send the contents of a form to a *cgi* program as an argument. The *GET* operation can be made conditional on the date a resource was last modified. *GET* can also be configured to obtain parts of the data.

HEAD: this request is identical to *GET*, but it does not return any data. However, it does return all the information about the data, such as the time of last modification, its type or its size.

POST: specifies the URL of a resource (for example a program) that can deal with the data supplied with the request. The processing carried out on the data depends on the function of the program specified in the URL. This method is designed to deal with:

- providing a block of data (for example, a form) to a data-handling process such as a servlet or a *cgi* program;
- posting a message to a bulletin board, mailing list or newsgroup;
- extending a database with an append operation.

PUT: requests that the data supplied in the request is stored with the given URL as its identifier, either as a modification of an existing resource or as a new resource.

DELETE: the server deletes the resource identified by the given URL. Servers may not always allow this operation, in which case the reply indicates failure.

OPTIONS: the server supplies the client with a list of methods it allows to be applied to the given URL (e.g. *GET*, *HEAD*, *PUT*) and its special requirements.

TRACE: the server sends back the request message. Used for diagnostic purposes.

The requests described above may be intercepted by a proxy server (see Section 2.2.2). The responses to *GET* and *HEAD* may be cached by proxy servers.

Message contents: The *Request* message specifies the name of a method, the URL of a resource, the protocol version, some headers and an optional message body. Figure 4.15 shows the contents of an HTTP *Request* message whose method is GET. When the URL specifies a data resource, the GET method does not have a message body.

Figure 4.16 HTTP *reply* message

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

The header fields contain request modifiers and client information, such as conditions on the latest date of modification of the resource or acceptable content type (for example, HTML text, audio or JPEG images). An authorization field can be used to provide the client's credentials in the form of a certificate specifying their rights to access a resource.

A *Reply* message specifies the protocol version, a status code and 'reason', some headers and an optional message body, as shown in Figure 4.16. The *status code* and *reason* provide a report on the success or otherwise in carrying out the request: the former is a three-digit integer for interpretation by a program, and the latter is a textual phrase that can be understood by a person. The header fields are used to pass additional information about the server or concerning access to the resource. For example, if the request requires authentication, the status of the response indicates this and a header field contains a challenge. Some status returns have quite complex effects. In particular, a 303 status response tells the browser to look under a different URL, which is supplied in a header field in the reply. It is intended for use in a response from a program activated by a POST request when the program needs to redirect the browser to a selected resource.

The message body in request or reply messages contains the data associated with the URL specified in the request. The message body has its own headers specifying information about the data, such as its length, its Mime type, its character set, its content encoding, and the last date it was modified. The Mime type field specifies the type of the data, for example *image/jpeg* or *text/plain*. The content-encoding field specifies the compression algorithm to be used.

4.5 Group communication

The pairwise exchange of messages is not the best model for communication from one process to a group of other processes, as for example when a service is implemented as a number of different processes in different computers, perhaps to provide fault tolerance or to enhance availability. A *multicast operation* is more appropriate – this is an operation that sends a single message from one process to each of the members of a group of processes, usually in such a way that the membership of the group is transparent to the sender. There is a range of possibilities in the desired behaviour of a multicast. The simplest provides no guarantees about message delivery or ordering.

5

DISTRIBUTED OBJECTS AND REMOTE INVOCATION

In this chapter we introduce communication between distributed objects by means of remote method invocation (RMI). Objects that can receive remote method invocations are called remote objects and they implement a remote interface. Due to the possibility of independent failure of invoker and invoked objects, RMIs have different semantics from local calls. They can be made to look very similar to local invocations, but total transparency is not necessarily desirable. The code for marshalling and unmarshalling arguments and sending request and reply messages can be generated automatically by an interface compiler from the definition of the remote interface.

Remote procedure call is to RMI as procedure call is to object invocation. It is described briefly and illustrated by a case study of Sun RPC.

Distributed event-based systems allow objects to subscribe to events occurring at remote objects of interest and in turn to receive notifications when such events occur. Events and notifications provide a way for heterogeneous objects to communicate with one another asynchronously. The Jini distributed event specification is presented as a case study.

The use of RMI is illustrated in a case study of Java RMI.

Chapter 17 contains a case study on CORBA that includes CORBA RMI and the CORBA Event Service.

5.1 Introduction

This chapter is concerned with programming models for distributed applications – that is, those applications that are composed of cooperating programs running in several different processes. Such programs need to be able to invoke operations in other processes, often running in different computers. To achieve this, some familiar programming models have been extended to apply to distributed programs:

- The earliest and perhaps the best-known of these was the extension of the conventional procedure call model to the *remote procedure call* model, which allows client programs to call procedures in server programs running in separate processes and generally in different computers from the client.
- More recently, the object-based programming model has been extended to allow objects in different processes to communicate with one another by means of *remote method invocation* (RMI). RMI is an extension of local method invocation that allows an object living in one process to invoke the methods of an object living in another process.
- The event-based programming model allows objects to receive notification of the events at other objects in which they have registered interest. This model has been extended to allow distributed event-based programs to be written.

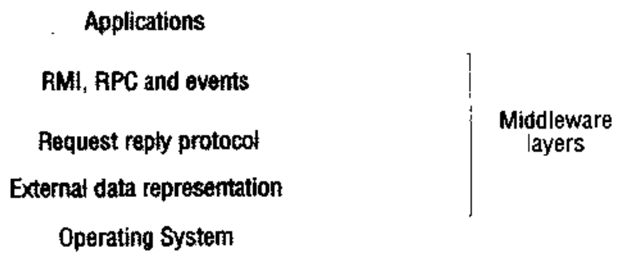
Note that we use the term 'RMI' to refer to remote method invocation in a generic way – this should not be confused with particular examples of remote method invocation such as Java RMI. Most current distributed systems software is written in object-oriented languages, and RPC can be understood in relation to RMI. Therefore this chapter concentrates on the RMI and event paradigms, each of which applies to distributed objects. Communication between distributed objects is introduced in Section 5.2, followed by a discussion of the design and implementation of RMI. A Java RMI case study is given in Section 5.5. RPC is discussed in the context of a case study of Sun RPC in Section 5.3. Events and distributed notifications are discussed in Section 5.4. A further case study on CORBA is given in Chapter 17.

Middleware ◊ Software that provides a programming model above the basic building blocks of processes and message passing is called middleware. The middleware layer uses protocols based on messages between processes to provide its higher-level abstractions such as remote invocations and events, as illustrated in Figure 5.1. For example, the remote method invocation abstraction is based on the request-reply protocol discussed in Section 4.4.

An important aspect of middleware is the provision of location transparency and independence from the details of communication protocols, operating systems and computer hardware. Some forms of middleware allow the separate components to be written in different programming languages.

Location transparency: In RPC, the client that calls a procedure cannot tell whether the procedure runs in the same process or in a different process, possibly on a different computer. Nor does the client need to know the location of the server. Similarly, in RMI the object making the invocation cannot tell whether the object it invokes is local or not and does not need to know its location. Also in distributed event-based programs, the

Figure 5.1 Middleware layer



objects generating events and the objects that receive notifications of those events need not be aware of one anothers' locations.

Communication protocols : The protocols that support the middleware abstractions are independent of the underlying transport protocols. For example, the request-reply protocol can be implemented over either UDP or TCP.

Computer hardware : Two agreed standards for external data representation are described in Section 4.3. These are used when marshalling and unmarshalling messages. They hide the differences due to hardware architectures, such as byte ordering.

Operating systems : The higher-level abstractions provided by the middleware layer are independent of the underlying operating systems.

Use of several programming languages : Some middleware is designed to allow distributed applications to use more than one programming language. In particular, CORBA (see Chapter 17) allows clients written in one language to invoke methods in objects that live in server programs written in another language. This is achieved by using an *interface definition language* or IDL to define interfaces. IDLs are discussed in the next section.

5.1.1 Interfaces

Most modern programming languages provide a means of organizing a program as a set of modules that can communicate with one another. Communication between modules can be by means of procedure calls between modules or by direct access to the variables in another module. In order to control the possible interactions between modules, an explicit *interface* is defined for each module. The interface of a module specifies the procedures and the variables that can be accessed from other modules. Modules are implemented so as to hide all the information about them except that which is available through its interface. So long as its interface remains the same, the implementation may be changed without affecting the users of the module.

Interfaces in distributed systems ◊ In a distributed program, the modules can run in separate processes. It is not possible for a module running in one process to access the variables in a module in another process. Therefore, the interface of a module that is intended for RPC or RMI cannot specify direct access to variables. Note that CORBA IDL interfaces can specify attributes, which seems to break this rule. However, the attributes are not accessed directly but by means of some getter and setter procedures added automatically to the interface.

The parameter-passing mechanisms, for example call by value and call by reference, used in local procedure call are not suitable when the caller and procedure are in different processes. The specification of a procedure or method in the interface of a module in a distributed program describes the parameters as *input* or *output* or sometimes both. *Input* parameters are passed to the remote module by sending the values of the arguments in the request message and then supplying them as arguments to the operation to be executed in the server. *Output* parameters are returned in the reply message and are used as the result of the call or to replace the values of the corresponding variables in the calling environment. When a parameter is used for both input and output the value must be transmitted in both the request and reply messages.

Another difference between local and remote modules is that pointers in one process are not valid in another remote one. Therefore, pointers cannot be passed as arguments or returned as results of calls to remote modules.

The next two paragraphs discuss the interfaces used in the original client-server model for RPC and in the distributed object model for RMI:

Service interfaces : In the client-server model, each server provides a set of procedures that are available for use by clients. For example, a file server would provide procedures for reading and writing files. The term *service interface* is used to refer to the specification of the procedures offered by a server, defining the types of the input and output arguments of each of the procedures.

Remote interfaces: In the distributed object model, a *remote interface* specifies the methods of an object that are available for invocation by objects in other processes, defining the types of the input and output arguments of each of them. However, the big difference is that the methods in remote interfaces can pass objects as arguments and results of methods. In addition, references to remote objects may also be passed – these should not be confused with pointers, which refer to specific memory locations. (Section 4.3.3 describes the contents of remote object references.)

Neither service interfaces nor remote interfaces may specify direct access to variables. In the latter case, this prohibits direct access to the instance variables of an object.

Interface definition languages ◊ An RMI mechanism can be integrated with a particular programming language if it includes an adequate notation for defining interfaces, allowing input and output parameters to be mapped onto the language's normal use of parameters. Java RMI is an example in which an RMI mechanism has been added to an object-oriented programming language. This approach is useful when all the parts of a distributed application can be written in the same language. It is also convenient because it allows the programmer to use a single language for local and remote invocation.

However, many existing useful services are written in C++ and other languages. It would be beneficial to allow programs written in a variety of languages, including Java, to access them remotely. *Interface definition languages* (or IDLs) are designed to allow objects implemented in different languages to invoke one another. An IDL provides a notation for defining interfaces in which each of the parameters of a method may be described as for *input* or *output* in addition to having its type specified.

Figure 5.2 CORBA IDL example

```
// In file Person.idl
struct Person {
    string name;
    string place;
    long year;
} ;
interface PersonList {
    readonly attribute string listname;
    void addPerson(in Person p) ;
    void getPerson(in string name, out Person p);
    long number();
};
```

Figure 5.2 shows a simple example of CORBA IDL. The *Person* structure is the same as the one used to illustrate marshalling in Section 4.<<CDR>>. The interface named *PersonList* specifies the methods available for RMI in a remote object that implements that interface. For example, the method *addPerson* specifies its argument as *in*, meaning that it is an *input* argument; and the method *getPerson* that retrieves an instance of *Person* by name specifies its second argument as *out*, meaning that it is an *output* argument. Our case studies include CORBA IDL as an example of an IDL for RMI (in Chapter 17) and Sun XDR as an IDL for RPC.

Other examples include the interface definition language for the RPC system in the OSF's Distributed Computing Environment (DCE) [OSF 1997], which uses C language syntax and is called IDL; and DCOM IDL which is based on DCE IDL [Box 1998] and is used in Microsoft's Distributed Common Object Model.

5.2 Communication between distributed objects

The object-based model for a distributed system introduced in Chapter 1 extends the model supported by object-oriented programming languages to make it apply to distributed objects. This section addresses communication between distributed objects by means of RMI. The material is presented under the following headings:

The object model: A brief review of the relevant aspects of the object model, suitable for the reader with a basic knowledge of an object-oriented programming language, for example Java or C++.

Distributed objects: A presentation of object-based distributed systems, which argues that the object model is very appropriate for distributed systems.

The distributed object model: A discussion of the extensions to the object model necessary for it to support distributed objects.

Design issues: A set of arguments about the design alternatives:

1. Local invocations are executed exactly once, but what suitable semantics is possible for remote invocations?
2. How can RMI semantics be made similar to those of local method invocation and what differences cannot be eliminated?

Implementation: An explanation as to how a layer of middleware above the request-reply protocol may be designed to support RMI between application-level distributed objects.

Distributed garbage collection: A presentation of an algorithm for distributed garbage collection that is suitable for use with the RMI implementation.

5.2.1 The object model

An object-oriented program, for example in Java or C++, consists of a collection of interacting objects, each of which consists of a set of data and a set of methods. An object communicates with other objects by invoking their methods, generally passing arguments and receiving results. Objects can encapsulate their data and the code of their methods. Some languages, for example Java and C++, allow programmers to define objects whose instance variables can be accessed directly. But for use in a distributed object system, an object's data should be accessible only via its methods.

Object references ◊ Objects can be accessed via object references. For example, in Java, a variable that appears to hold an object actually holds a reference to that object. To invoke a method in an object, the object reference and method name are given, together with any necessary arguments. The object whose method is invoked is sometimes called the *target* and sometimes the *receiver*. Object references are first-class values, meaning that they may, for example, be assigned to variables, passed as arguments and returned as results of methods.

Interfaces ◊ An interface provides a definition of the signatures of a set of methods (that is, the types of their arguments, return values and exceptions) without specifying their implementation. An object will provide a particular interface if its class contains code that implements the methods of that interface. In Java, a class may implement several interfaces, and the methods of an interface may be implemented by any class. An interface also defines a type that can be used to declare the type of variables or of the parameters and return values of methods. Note that interfaces do not have constructors.

Actions ◊ Action in an object-oriented program is initiated by an object invoking a method in another object. An invocation can include additional information (arguments) needed to carry out the method. The receiver executes the appropriate method and then returns control to the invoking object, sometimes supplying a result. An invocation of a method can have two effects:

1. the state of the receiver may be changed, and
2. further invocations on methods in other objects may take place.

As an invocation can lead to further invocations of methods in other objects, an action is a chain of related method invocations, each of which eventually returns. This explanation does not take account of exceptions.

Exceptions ◊ Programs can encounter many sorts of errors and unexpected conditions of varying seriousness. During the execution of a method, many different problems may be discovered: for example, inconsistent values in the object's variables, or failure in attempts to read or write to files or network sockets. When programmers need to insert tests in their code to deal with all possible unusual or erroneous cases, this detracts from the clarity of the normal case. Exceptions provide a clean way to deal with error conditions without complicating the code. In addition, each method heading explicitly lists as exceptions the error conditions it might encounter, allowing users of the method to deal with them. A block of code may be defined to *throw* an exception whenever particular unexpected conditions or errors arise. This means that control passes to another block of code that *catches* the exception. Control does not return to the place where the exception was thrown.

Garbage collection ◊ It is necessary to provide a means of freeing the space occupied by objects when they are no longer needed. A language, for example Java, that can detect automatically when an object is no longer accessible recovers the space and makes it available for allocation to other objects. This process is called *garbage collection*. When a language (for example C++) does not support garbage collection, the programmer has to cope with the freeing of space allocated to objects. This can be a major source of errors.

5.2.2 Distributed objects

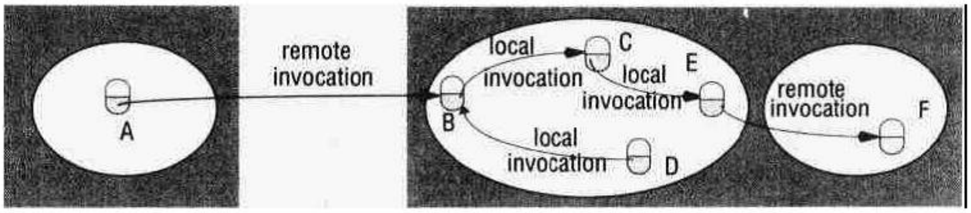
The state of an object consists of the values of its instance variables. In the object-based paradigm the state of a program is partitioned into separate parts, each of which is associated with an object. Since object-based programs are logically partitioned, the physical distribution of objects into different processes or computers in a distributed system is a natural extension.

Distributed object systems may adopt the client-server architecture. In this case, objects are managed by servers and their clients invoke their methods using remote method invocation. In RMI, the client's request to invoke a method of an object is sent in a message to the server managing the object. The invocation is carried out by executing a method of the object at the server and the result is returned to the client in another message. To allow for chains of related invocations, objects in servers are allowed to become clients of objects in other servers.

Distributed objects can assume the other architectural models described in Chapter 2. For example, objects can be replicated in order to obtain the usual benefits of fault tolerance and enhanced performance, and objects can be migrated with a view to enhancing their performance and availability.

Having client and server objects in different processes enforces encapsulation. That is, the state of an object can be accessed only by the methods of the object, which means that it is not possible for unauthorized methods to act on the state. For example, the possibility of concurrent RMIs from objects in different computers implies that an object may be accessed concurrently. Therefore, the possibility of conflicting accesses

Figure 5.3 Remote and local method invocations



arises. However, the fact that the data of an object is accessed only by its own methods allows objects to provide methods for protecting themselves against incorrect accesses. For example, they may use synchronization primitives such as condition variables to protect access to their instance variables.

Another advantage of treating the shared state of a distributed program as a collection of objects is that an object may be accessed via RMI or it may be copied into a local cache and accessed directly provided that the class implementation is available locally.

The fact that objects are accessed only via their methods gives another advantage for heterogeneous systems in that different data formats may be used at different sites – these formats will be unnoticed by clients that use RMI to access the methods of the objects.

5.2.3 The distributed object model

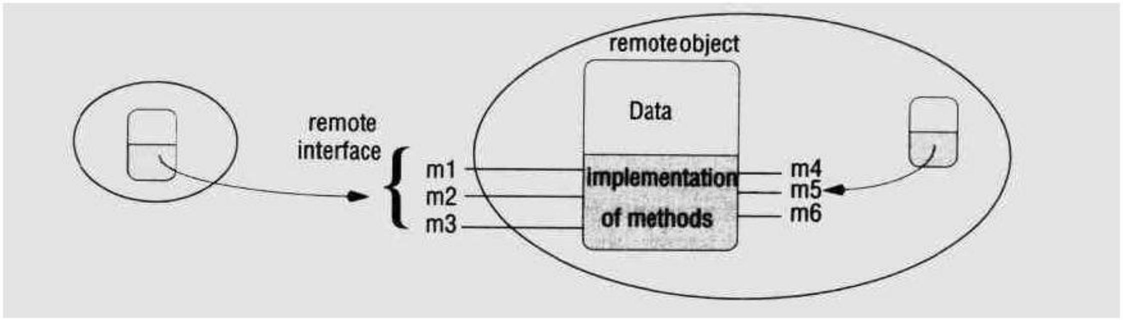
This section discusses extensions to the object model to make it applicable to distributed objects. Each process contains a collection of objects, some of which can receive both local and remote invocations, whereas the other objects can receive only local invocations, as shown in Figure 5.3. Method invocations between objects in different processes, whether in the same computer or not, are known as *remote method invocations*. Method invocations between objects in the same process are local method invocations.

We refer to objects that can receive remote invocations as *remote objects*. In Figure 5.3, the objects B and F are remote objects. All objects can receive local invocations, although they can receive them only from other objects that hold references to them. For example, object C must have a reference to object E so that it can invoke one of its methods. The following two fundamental concepts are at the heart of the distributed object model:

Remote object reference: Other objects can invoke the methods of a remote object if they have access to its *remote object reference*. For example, a remote object reference for B in Figure 5.3 must be available to A.

Remote interface: Every remote object has a *remote interface* that specifies which of its methods can be invoked remotely. For example, the objects B and F must have remote interfaces.

Figure 5.4 A remote object and its remote interface



The following paragraphs discuss remote object references, remote interfaces and other aspects of the distributed object model.

Remote object references ◊ The notion of object reference is extended to allow any object that can receive an RMI to have a remote object reference. A remote object reference is an identifier that can be used throughout a distributed system to refer to a particular unique remote object. Its representation, which is generally different from that of local object references, is discussed in the Section 4.3.3. Remote object references are analogous to local ones in that:

1. the remote object to receive a remote method invocation is specified as a remote object reference; and
2. remote object references may be passed as arguments and results of remote method invocations.

Remote interfaces ◊ The class of a remote object implements the methods of its remote interface, for example as public instance methods in Java. Objects in other processes can invoke only the methods that belong to its remote interface, as shown in Figure 5.4. Local objects can invoke the methods in the remote interface as well as other methods implemented by a remote object. Note that remote interfaces, like all interfaces, do not have constructors.

The CORBA system provides an interface definition language (IDL), which is used for defining remote interfaces. See Figure 5.2 for an example of a remote interface defined in CORBA IDL. The classes of remote objects and the client programs may be implemented in any language such as C++ or Java for which an IDL compiler is available. CORBA clients need not use the same language as the remote object in order to invoke its methods remotely.

In Java RMI, remote interfaces are defined in the same way as any other Java interface. They acquire their ability to be remote interfaces by extending an interface named *Remote*. Both CORBA IDL (Section 17.2.3) and Java support multiple inheritance of interfaces. That is, an interface is allowed to extend one or more other interfaces.

Actions in a distributed object system ◊ As in the non-distributed case, an action is initiated by a method invocation, which may result in further invocations on methods in other objects. But in the distributed case, the objects involved in a chain of related invocations may be located in different processes or different computers. When an invocation crosses the boundary of a process or computer, RMI is used, and the remote reference of the object must be available to make the RMI possible. In Figure 5.3, the object A needs to hold a remote object reference to object B. Remote object references may be obtained as the results of remote method invocations. For example, object A in Figure 5.3 might obtain a remote reference to object F from object B.

Garbage collection in a distributed-object system ◊ If a language, for example Java, supports garbage collection, then any associated RMI system should allow garbage collection of remote objects. Distributed garbage collection is generally achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection, usually based on reference counting. Section 5.2.6 describes such a scheme in detail.

Exceptions ◊ Any remote invocation may fail for reasons related to the invoked object being in a different process or computer from the invoker. For example, the process containing the remote object may have crashed or may be too busy to reply, or the invocation or result message may be lost. Therefore, remote method invocation should be able to raise exceptions such as timeouts that are due to distribution as well as those raised during the execution of the method invoked. Examples of the latter are an attempt to read beyond the end of a file, or to access a file without the correct permissions.

CORBA IDL provides a notation for specifying application-level exceptions, and the underlying system generates standard exceptions when errors due to distribution occur. CORBA client programs need to be able to handle exceptions. For example, a C++ client program will use the exception mechanisms in C++.

5.2.4 Design Issues for RMI

The previous section suggested that RMI is a natural extension of local method invocation. In this section, we discuss two design issues that arise in making this extension:

- Although local invocations are executed exactly once, this cannot always be the case for remote method invocations. The alternatives are discussed.
- The level of transparency that is desirable for RMI.

In the remainder of Section 5.2, we refer to the processes that host remote objects as servers and the processes that host their invokers as clients. Servers can also be clients.

RMI invocation semantics ◊ Request-reply protocols were discussed in Section 4.4, where we showed that *doOperation* can be implemented in different ways to provide different delivery guarantees. The main choices are:

Retry request message: whether to retransmit the request message until either a reply is received or the server is assumed to have failed.

Duplicate filtering: when retransmissions are used, whether to filter out duplicate requests at the server.

Retransmission of results: whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server.

Figure 5.5 Invocation semantics

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Combinations of these choices lead to a variety of possible semantics for the reliability of remote invocations as seen by the invoker. Figure 5.5 shows the choices of interest, with corresponding names for the invocation semantics that they produce. Note that for local method invocations, the semantics are *exactly once*, meaning that every method is executed exactly once. The invocation semantics are defined as follows:

Maybe invocation semantics: With *maybe* invocation semantics, the invoker cannot tell whether a remote method has been executed once or not at all. Maybe semantics arises when none of the fault tolerance measures is applied. This can suffer from the following types of failure:

- omission failures if the invocation or result message is lost;
- crash failures when the server containing the remote object fails.

If the result message has not been received after a timeout and there are no retries, it is uncertain whether the method has been executed. If the invocation message was lost, then the method will not have been executed. On the other hand, the method may have been executed and the result message lost. A crash failure may occur either before or after the method is executed. Moreover, in an asynchronous system, the result of executing the method may arrive after the timeout. Maybe semantics is useful only for applications in which occasional failed invocations are acceptable.

At-least-once invocation semantics: With *at-least-once* invocation semantics, the invoker receives either a result, in which case the invoker knows that the method was executed at least once, or an exception informing it that no result was received. *At-least-once* invocation semantics can be achieved by the retransmission of request messages, which masks the omission failures of the invocation or result message. *At-least-once* invocation semantics can suffer from the following types of failure:

- crash failures when the server containing the remote object fails;
- arbitrary failures. In cases when the invocation message is retransmitted, the remote object may receive it and execute the method more than once, possibly causing wrong values to be stored or returned.

Chapter 4 defines an *idempotent operation* as one that can be performed repeatedly with the same effect as if it had been performed exactly once. Non-idempotent operations can have the wrong effect if they are performed more than once. For example, an operation to increase a bank balance by $10 should be performed only once; if it were to be repeated, the balance would grow and grow! If the objects in a server can be designed so that all of the methods in their remote interfaces are idempotent operations, then at-least-once call semantics may be acceptable.

At-most-once invocation semantics: With *at-most-once* invocation semantics, the invoker receives either a result, in which case the invoker knows that the method was executed exactly once, or an exception informing it that no result was received, in which case the method will have been executed either once or not at all. *At-most-once* invocation semantics can be achieved by using all of the fault tolerance measures. As in the previous case, the use of retries masks any omission failures of the invocation or result messages. The additional fault tolerance measures prevent arbitrary failures by ensuring that for each RMI a method is never executed more than once. In both Java RMI and CORBA, the invocation semantics is *at-most-once*, but CORBA allows *maybe* semantics to be requested for methods that do not return results. Sun RPC provides at-least-once call semantics.

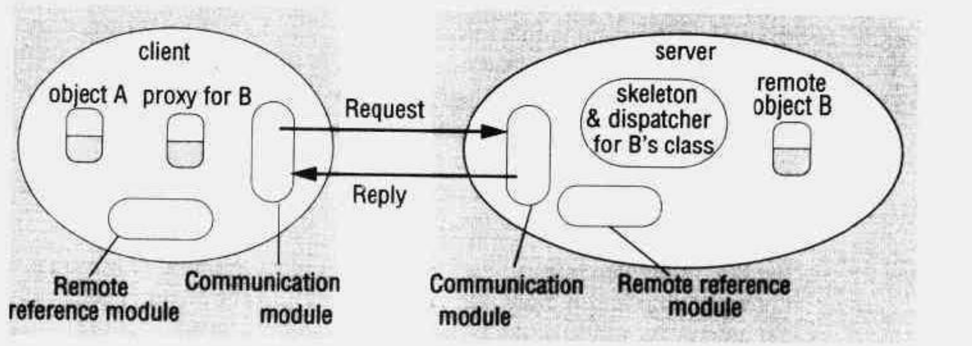
Transparency ◊ The originators of RPC, Birrell and Nelson [1984], aimed to make remote procedure calls as much like local procedure calls as possible, with no distinction in syntax between a local and a remote procedure call. All the necessary calls to marshalling and message-passing procedures were hidden from the programmer making the call. Although request messages are retransmitted after a timeout, this is transparent to the caller – to make the semantics of remote procedure calls like that of local procedure calls. This notion of transparency has been extended to apply to distributed objects, but it involves hiding not only marshalling and message passing but also the task of locating and contacting a remote object. As an example, Java RMI makes remote method invocations very like local ones by allowing them to use the same syntax.

However, remote invocations are more vulnerable to failure than local ones, since they involve a network, another computer and another process. Whichever of the above invocation semantics is chosen, there is always the chance that no result will be received and in the case of failure, it is impossible to distinguish between failure of the network and of the remote server process. This requires that objects making remote invocations be able to recover from such situations.

The latency of a remote invocation is several orders of magnitude greater than that of a local one. This suggests that programs that make use of remote invocations need to be able to take this factor into account, perhaps by minimizing remote interactions. The designers of Argus [Liskov and Scheifler 1982] suggested that a caller should be able to abort a remote procedure call that is taking too long in such a way that it has no effect on the server. To allow this, the server would need to be able to restore things to how they were before the procedure was called. These issues are discussed in Chapter 12.

Waldo *et al.* [1994] say that the difference between local and remote objects should be expressed at the remote interface, to allow objects to react in a consistent way to possible partial failures. Other systems went further than this by arguing that the syntax of a remote call should be different from that of a local call: in the case of Argus, the language was extended to make remote operations explicit to the programmer.

Figure 5.6 The role of proxy and skeleton in remote method invocation



The choice as to whether remote invocations should be transparent is also available to the designers of IDLs. For example, in CORBA, a remote invocation throws an exception when the client is unable to communicate with a remote object. This requires that the client program handle such exceptions, allowing it to deal with such failures. An IDL can also provide a facility for specifying the call semantics of a method. This can help the designer of the remote object – for example, if at-least-once call semantics is chosen to avoid the overheads of at-most-once, the operations of the object are designed to be idempotent.

The current consensus seems to be that remote invocations should be made transparent in the sense that the syntax of a remote invocation is the same as that of a local invocation, but that the difference between local and remote objects should be expressed in their interfaces. In the case of Java RMI, remote objects can be distinguished by the fact that they implement the *Remote* interface and throw *RemoteExceptions*. Implementors of a remote object whose interface is specified in an IDL are also aware of the difference. The knowledge that an object is intended to be accessed by remote invocation has another implication for its designer: it should be able to keep its state consistent in the presence of concurrent accesses from multiple clients.

5.2.5 Implementation of RMI

Several separate objects and modules are involved in achieving a remote method invocation. These are shown in Figure 5.6, in which an application-level object A invokes a method in a remote application-level object B for which it holds a remote object reference. This section discusses the roles of each of the components shown in that figure, dealing first with the communication and remote reference modules and then with the RMI software that runs over them.

The remainder of this section deals with the following related topics: the generation of proxies, the binding of names to their remote object references, the activation and passivation of objects and the location of objects from their remote object references.

Communication module ◊ The two cooperating communication modules carry out the request-reply protocol, which transmits *request* and *reply* messages between client and server. The contents of *request* and *reply* messages are shown in Figure 4.13. The communication module uses only the first three items, which specify the message type, its *requestId* and the remote reference of the object to be invoked. The *methodId* and all the marshalling and unmarshalling is the concern of the RMI software discussed below. The communication modules are together responsible for providing a specified invocation semantics, for example at-most-once.

The communication module in the server selects the dispatcher for the class of the object to be invoked, passing on its local reference, which it gets from the remote reference module in return for the remote object identifier in the *request* message. The role of dispatcher is discussed under RMI software below.

Remote reference module ◊ A remote reference module is responsible for translating between local and remote object references and for creating remote object references. To support its responsibilities, the remote reference module in each process has a *remote object table* that records the correspondence between local object references in that process and remote object references (which are system-wide). The table includes:

- An entry for all the remote objects held by the process. For example, in Figure 5.6, the remote object B will be recorded in the table at the server.
- An entry for each local proxy. For example, in Figure 5.6 the proxy for B will be recorded in the table at the client.

The role of a proxy is discussed under RMI software below. The actions of the remote reference module are as follows:

- When a remote object is to be passed as argument or result for the first time, the remote reference module is asked to create a remote object reference, which it adds to its table.
- When a remote object reference arrives in a request or reply message, the remote reference module is asked for the corresponding local object reference, which may refer either to a proxy or to a remote object. In the case that the remote object reference is not in the table, the RMI software creates a new proxy and asks the remote reference module to add it to the table.

This module is called by components of the RMI software when they are marshalling and unmarshalling remote object references. For example, when a request message arrives, the table is used to find out which local object is to be invoked.

The RMI software ◊ This consists of a layer of software between the application-level objects and the communication and remote reference modules. The roles of the middleware objects shown in Figure 5.6 are as follows:

Proxy: The role of a proxy is to make remote method invocation transparent to clients by behaving like a local object to the invoker; but instead of executing an invocation, it forwards it in a message to a remote object. It hides the details of the remote object reference, the marshalling of arguments, unmarshalling of results and sending and receiving of messages from the client. There is one proxy for each remote object for which a process holds a remote object reference. The class of a proxy implements the methods in the remote interface of the remote object it represents. This ensures that remote method invocations are suitable for the type of the remote object. However, the proxy implements them quite differently. Each method of the proxy marshals a reference to the target object, its own *methodId* and its arguments into a *request* message and sends it to the target, awaits the *reply* message, unmarshals it and returns the results to the invoker.

Dispatcher: A server has one dispatcher and skeleton for each class representing a remote object. In our example, the server has a dispatcher and skeleton for the class of remote object B. The dispatcher receives the *request* message from the communication module. It uses the *methodId* to select the appropriate method in the skeleton, passing on the *request* message. The dispatcher and proxy use the same allocation of *methodIds* to the methods of the remote interface.

Skeleton: The class of a remote object has a *skeleton*, which implements the methods in the remote interface. They are implemented quite differently from the methods in the remote object. A skeleton method unmarshals the arguments in the *request* message and invokes the corresponding method in the remote object. It waits for the invocation to complete and then marshals the result, together with any exceptions, in a *reply* message to the sending proxy's method.

Remote object references are marshalled in the form shown in Figure 4.10, which includes information about the remote interface of the remote object, for example the the name of the remote interface or the class of the remote object. This information enables the proxy class to be determined so that a new proxy may be created when it is needed. For example, the proxy class name may be generated by appending "*_proxy*" to the name of the remote interface.

Generation of the classes for proxies, dispatchers and skeletons ◊ The classes for the proxy, dispatcher and skeleton used in RMI are generated automatically by an interface compiler. For example, in the Orbix implementation of CORBA, interfaces of remote objects are defined in CORBA IDL, and the interface compiler can be used to generate the classes for proxies, dispatchers and skeletons in C++. For Java RMI, the set of methods offered by a remote object is defined as a Java interface that is implemented within the class of the remote object. The Java RMI compiler generates the proxy, dispatcher and skeleton classes from the class of the remote object.

Server and client programs ◊ The server program contains the classes for the dispatchers and skeletons, together with the implementations of the classes of all of the remote objects that it supports. The latter are sometimes called servant classes. In addition, the server program contains an *initialization* section (for example in a *main* method in Java or C++). The *initialization* section is responsible for creating and initializing at least one of the remote objects to be hosted by the server. Additional remote objects may be created in response to requests from clients. The *initialization* section may also register some of its remote objects with a binder (see the next paragraph). Generally, it will register just one remote object, which can be used to access the rest.

The client program will contain the classes of the proxies for all of the remote objects that it will invoke. It can use a binder to look up remote object references.

Factory methods: We noted earlier that remote object interfaces cannot include constructors. This means that remote objects cannot be created by remote invocation on constructors. Remote objects are created either in the *initialization* section or in methods in a remote interface designed for that purpose. The term *factory method* is sometimes used to refer to a method that creates remote objects, and a *factory object* is an object with factory methods. Any remote object that needs to be able to create new remote objects on demand for clients must provide methods in its remote interface for this purpose. Such methods are called factory methods, although they are really just normal methods.

The binder ◊ Client programs generally require a means of obtaining a remote object reference for at least one of the remote objects held by a server. For example, in Figure 5.3, object A would require a remote object reference for object B. A *binder* in a distributed system is a separate service that maintains a table containing mappings from textual names to remote object references. It is used by servers to register their remote objects by name and by clients to look them up. Chapter 17 contains a discussion of the CORBA Naming Service. The Java binder, RMIregistry, is discussed briefly in the case study on Java RMI in Section 5.5.

Server threads ◊ Whenever an object executes a remote invocation, that execution may lead to further invocations of methods in other remote objects, which may take some time to return. To avoid the execution of one remote invocation delaying the execution of another, servers generally allocate a separate thread for the execution of each remote invocation. When this is the case, the designer of the implementation of a remote object must allow for the effects on its state of concurrent executions.

Activation of remote objects ◊ Some applications require that information survive for long periods of time. However, it is not practical for the objects representing such information to be kept in running processes for unlimited periods, particularly since they are not necessarily in use all of the time. To avoid the potential waste of resources due to running all of the servers that manage remote objects all of the time, the servers can be started whenever they are needed by clients, as is done for the standard set of TCP services such as FTP, which are started on demand by a service called *Inetd*. Processes that start server processes to host remote objects are called *activators* for the following reasons.

A remote object is described as *active* when it is available for invocation within a running process, whereas it is called *passive* if is not currently active but can be made active. A passive object consists of two parts:

1. the implementation of its methods; and
2. its state in the marshalled form.

Activation consists of creating an active object from the corresponding passive object by creating a new instance of its class and initializing its instance variables from the stored state. Passive objects can be activated on demand, for example when they need to be invoked by other objects.

An *activator* is responsible for:

- Registering passive objects that are available for activation, which involves recording the names of servers against the URLs or file names of the corresponding passive objects.
- Starting named server processes and activating remote objects in them.

- Keeping track of the locations of the servers for remote objects that it has already activated.

The CORBA case study describes its activator, which is called the implementation repository. Java RMI uses one activator on each server computer, which is responsible for activating objects on that computer.

Persistent object stores ◊ An object that is guaranteed to live between activations of processes is called a *persistent object*. Persistent objects are generally managed by persistent object stores which store their state in a marshalled form on disk. Examples include the CORBA persistent object service (see Section 17.3) and Persistent Java [Jordan 1996, java.sun.com IV].

In general, a persistent object store will manage very large numbers of persistent objects, which are stored on disk until they are needed. They will be activated when their methods are invoked by other objects. Activation is generally designed to be transparent – that is, the invoker should not be able to tell whether an object is already in main memory or has to be activated before its method is invoked. Persistent objects that are no longer needed in main memory can be passivated. In most cases, objects are saved in the persistent object store whenever they reach a consistent state, for the sake of fault tolerance. The persistent object store needs a strategy for deciding when to passivate objects. For example, it may do so in response to a request in the program that activated the objects, for example at the end of a transaction or when the program exits. Persistent object stores generally attempt to optimize passivation by saving only those objects that have been modified since the last time they were saved.

Persistent object stores generally allow collections of related persistent objects to have human-readable names such as pathnames or URLs. In practice, each human-readable name is associated with the root of a connected set of persistent objects.

There are two approaches to deciding whether an object is persistent or not:

- The persistent object store maintains some persistent roots and any object that is reachable from a persistent root is defined to be persistent. This approach is used by Persistent Java and by PerDiS [Ferreira *et al.* 2000]. Persistent object stores using this approach make use of a garbage collector to dispose of objects that are no longer reachable from the persistent roots.

- The persistent object store provides some classes on which persistence is based – persistent objects belong to their subclasses. For example, in Arjuna [Parrington *et al.* 1995], persistent objects are based on C++ classes that provide transactions and recovery. Unwanted objects must be deleted explicitly.

Some persistent object stores, for example PerDiS and Khazana [Carter *et al.* 1998] allow objects to be activated in multiple caches local to users, instead of in servers. In this case, a cache consistency protocol is required; Chapter 16 discusses a variety of consistency models.

Object location ◊ Section 4.3.3 describes a form of remote object reference that contains the Internet address and port number of the process that created the remote object as a way of guaranteeing uniqueness. This form of remote object reference can also be used as an address for a remote object so long as that object remains in the same process for the rest of its life. But some remote objects will exist in a series of different processes, possibly on different computers, throughout their lifetime. In this case, a remote object reference cannot act as an address. Clients making invocations require both a remote object reference and an address to which to send invocations.

A *location service* helps clients to locate remote objects from their remote object references. It uses a database that maps remote object references to their probable current locations – the locations are probable because an object may have migrated again since it was last heard of. For example, the Clouds system [Dasgupta *et al.* 1991] and the Emerald system [Jul *et al.* 1988] used a cache/broadcast scheme in which a member of a location service on each computer holds a small cache of remote object reference-to-location mappings. If a remote object reference is in the cache, that address is tried for the invocation and will fail if the object has moved. To locate an object that has moved or whose location is not in cache, the system broadcasts a request. This scheme may be enhanced by the use of forward location pointers, which contain hints as to the new location of an object.

5.2.6 Distributed garbage collection

The aim of a distributed garbage collector is to ensure that if a local or remote reference to an object is still held anywhere in a set of distributed objects, then the object itself will continue to exist, but as soon as no object any longer holds a reference to it, the object will be collected and the memory it uses recovered.

We describe the Java distributed garbage collection algorithm, which is similar to the one described by Birrell *et al.* [1995]. It is based on reference counting. Whenever a remote object reference enters a process, a proxy will be created and will stay there for as long as it is needed. The process where the object lives (its server) should be informed of the new proxy at the client. Then later when there is no longer a proxy at the client, the server should be informed. The distributed garbage collector works in cooperation with the local garbage collectors as follows:

- Each server process maintains a set of the processes that hold remote object references for each of its remote objects; for example, *B.holders* is the set of client processes (virtual machines) that have proxies for object B. (In Figure 5.6, this set will include the client process illustrated.) This set can be held in an additional column in the remote object table.
- When a client *C* first receives a remote reference to a particular remote object, *B*, it makes an *addRef(B)* invocation to the server of that remote object and then creates a proxy; the server adds *C* to *B.holders*.
- When a client *C*'s garbage collector notices that a proxy for remote object *B* is no longer reachable, it makes a *removeRef(B)* invocation to the corresponding server and then deletes the proxy; the server removes *C* from *B.holders*.
- When *B.holders* is empty, the server's local garbage collector will reclaim the space occupied by *B* unless there are any local holders.

This algorithm is intended to be carried out by means of pairwise request-reply communication with at-most-once invocation semantics between the remote reference modules in processes – it does not require any global synchronization. Note also that the extra invocations made on behalf of the garbage collection algorithm do not affect every normal RMI; they occur only when proxies are created and deleted.

There is a possibility that one client may make a *removeRef(B)* invocation at about the same time as another client makes an *addRef(B)* invocation. If the *removeRef* arrives first and *B.holders* is empty, the remote object *B* could be deleted before the *addRef* arrives. To avoid this situation, if the set *B.holders* is empty at the time when a remote object reference is transmitted, a temporary entry is added until the *addRef* arrives.

The Java distributed garbage collection algorithm tolerates communication failures by using the following approach. The *addRef* and *removeRef* operations are idempotent. In the case that an *addRef(B)* call returns an exception (meaning that the method was either executed once or not at all), the client will not create a proxy but will make a *removeRef(B)* call. The effect of *removeRef* is correct whether or not the *addRef* succeeded. The case where *removeRef* fails is dealt with by leases, as described in the next paragraph.

The Java distributed garbage collection algorithm can tolerate the failure of client processes. To achieve this, servers *lease* their objects to clients for a limited period of time. The lease period starts when the client makes an *addRef* invocation to the server. It ends either when the time has expired or when the client makes a *removeRef* invocation to the server. The information stored by the server concerning each lease contains the identifier of the client's virtual machine and the period of the lease. Clients are responsible for requesting the server to renew their leases before they expire.

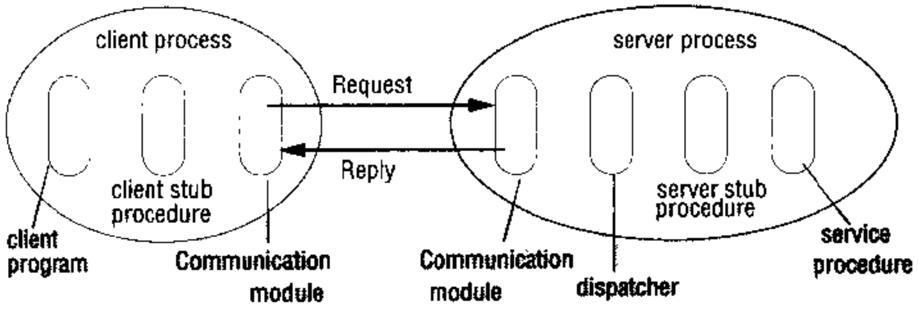
Leases in Jini ◊ The Jini distributed system includes a specification for leases [Arnold *et al.* 1999] that can be used in a variety of situations when one object offers a resource to another object, as for example when remote objects offer references to other objects. Objects that offer such resources are at risk of having to maintain the resources when the users are no longer interested or their programs may have exited. To avoid complicated protocols to discover whether the resource users are still interested, the resources are offered for a limited period of time. The granting of the use of a resource for a period of time is called a *lease*. The object offering the resource will maintain it until the time in the lease expires. The resource users are responsible for requesting their renewal when they expire.

The period of a lease may be negotiated between the grantor and the recipient, although this does not happen with the leases used in Java RMI. An object representing a lease implements the *Lease* interface. It contains information about the period of the lease and methods enabling the lease to be renewed or cancelled. The grantor returns an instance of a *Lease* when it supplies a resource to another object.

5.3 Remote procedure call

A remote procedure call is very similar to a remote method invocation in that a client program calls a procedure in another program running in a server process. Servers may be clients of other servers to allow chains of RPCs. As mentioned in the introduction to this chapter, a server process defines in its *service interface* the procedures that are available for calling remotely. RPC, like RMI, may be implemented to have one of the choices of invocation semantics discussed in Section 5.2.4 – at-least-once or at-most-

Figure 5.7 Role of client and server stub procedures in RPC



once are generally chosen. RPC is generally implemented over a request-reply protocol like the one discussed in Section 4.4, which is simplified by the omission of object references from request messages. The contents of request and reply messages are the same as those illustrated for RMI in Figure 4.13, except that the *ObjectReference* field is omitted.

The software that supports RPC is shown in Figure 5.7. It is similar to that shown in Figure 5.6 except that no remote reference modules are required, since procedure call is not concerned with objects and object references. The client that accesses a service includes one *stub procedure* for each procedure in the service interface. The role of a stub procedure is similar to that of a proxy. It behaves like a local procedure to the client, but instead of executing the call, it marshals the procedure identifier and the arguments into a request message, which it sends via its communication module to the server. When the reply message arrives, it unmarshals the results. The server process contains a dispatcher together with one server stub procedure and one service procedure for each procedure in the service interface. The dispatcher selects one of the server stub procedures according to the procedure identifier in the request message. A *server stub procedure* is like a skeleton method in that it unmarshals the arguments in the request message, calls the corresponding service procedure and marshals the return values for the reply message. The service procedures implement the procedures in the service interface.

The client and server stub procedures and the dispatcher can be generated by an interface compiler from the interface definition of the service.

5.3.1 Sun RPC case study

RFC 1831 [Srinivasan 1995a] describes Sun RPC which was designed for client-server communication in the Sun NFS network file system. Sun RPC is sometimes called ONC (Open Network Computing) RPC. It is supplied as a part of the various Sun and other UNIX operating systems and is also available with other NFS installations. Implementors have the choice of using remote procedure calls over either UDP or TCP. When Sun RPC is used with UDP, the length of request and reply messages is restricted in length – theoretically to 64 kilobytes, but more often in practice to 8 or 9 kilobytes. It uses at-least-once call semantics. Broadcast RPC is an option.

Figure 5.8 Files interface in Sun XDR

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
     int length;
     char buffer[MAX];
};
struct writeargs {
     FileIdentifier f;
     FilePointer position;
     Data data;
};
struct readargs {
     FileIdentifier f;
     FilePointer position;
     Length length;
};

program FILEREADWRITE {
 version VERSION {
     void WRITE(writeargs)=1;                                        1
     Data READ(readargs)=2;                                          2
     }=2;
} = 9999;
```

The Sun RPC system provides an interface language called XDR and an interface compiler called *rpcgen* which is intended for use with the C programming language.

Interface definition language ◊ The Sun XDR language, which was originally designed for specifying external data representations, was extended to become an interface definition language. It may be used to define a service interface for Sun RPC by specifying a set of procedure definitions together with supporting type definitions. The notation is rather primitive in comparison with that used by CORBA IDL or Java. In particular:

- Most languages allow interface names to be specified, but Sun RPC does not – instead of this, a program number and a version number are supplied. The program numbers can be obtained from a central authority to allow every program to have its own unique number. The version number changes when a procedure signature changes. Both program and version number are passed in the request message, so that client and server can check that they are using the same version.

- A procedure definition specifies a procedure signature and a procedure number. The procedure number is used as a procedure identifier in request messages. (It would be possible for the interface compiler to generate procedure identifiers.)
- Only a single input parameter is allowed. Therefore, procedures requiring multiple parameters must include them as components of a single structure.
- The output parameters of a procedure are returned via a single result.
- The procedure signature consists of the result type, the name of the procedure and the type of the input parameter. The type of both the result and the input parameter may specify either a single value or a structure containing several values.

For example, see the XDR definition in Figure 5.8 of an interface with a pair of procedures for writing and reading files. The program number is 9999 and the version number is 2. The *READ* procedure (line 2) takes as input parameter a structure with three components specifying a file identifier, a position in the file and the number of bytes required. Its result is a structure containing the number of bytes returned and the file data. The *WRITE* procedure (line 1) has no result. The *WRITE* and *READ* procedures are given numbers 1 and 2. The number zero is reserved for a null procedure, which is generated automatically and is intended to be used to test whether a server is available.

The interface definition language provides a notation for defining constants, typedefs, structures, enumerated types, unions and programs. Typedefs, structures and enumerated types use the C language syntax. The interface compiler *rpcgen* can be used to generate the following from an interface definition:

- client stub procedures;
- server *main* procedure, dispatcher and server stub procedures;
- XDR marshalling and unmarshalling procedures for use by the dispatcher and client and server stub procedures.

Binding ◊ Sun RPC runs a local binding service called the *port mapper* at a well-known port number on each computer. Each instance of a port mapper records the program number, version number and port number in use by each service running locally. When a server starts up it registers its program number, version number and port number with the local port mapper. When a client starts up, it finds out the server's port by making a remote request to the port mapper at the server's host, specifying the program number and version number.

When a service has multiple instances running on different computers, the instances may use different port numbers for receiving client requests. If a client needs to multicast a request to all the instances of a service that are using different port numbers, it cannot use a direct broadcast message for this purpose. The solution is that clients make multicast remote procedure calls by broadcasting them to all the port mappers, specifying the program and version number. Each port mapper forwards all such calls to the appropriate local service program, if there is one.

Authentication ◊ Sun RPC request and reply messages provide additional fields enabling authentication information to be passed between client and server. The request message contains the credentials of the user running the client program. For example, in the UNIX style of authentication the credentials include the *uid* and *gid* of the user. Access control mechanisms can be built on top of the authentication information which is made available to the server procedures via a second argument. The server program is responsible for enforcing access control by deciding whether to execute each procedure call according to the authentication information. For example, if the server is an NFS file server, it can check whether the user has sufficient rights to carry out a requested file operation.

Several different authentication protocols can be supported. These include:

- none;
- UNIX style as described above;
- a style in which a shared key is established for signing the RPC messages; or
- Kerberos style of authentication (see Chapter 7).

A field in the RPC header indicates which style is being used.

A more generic approach to security is described in RFC 2203 [Eisler *et al.* 1997]. It provides for secrecy and integrity of RPC messages as well as authentication. It allows client and server to negotiate a security context in which either no security is applied or in the case that security is required, message integrity or message privacy or both may be applied.

Client and server programs ◊ Further material on Sun RPC is available at www.cdk3.net/RMI. It includes example client and server programs corresponding to the interface defined in Figure 5.8.

5.4 Events and notifications

The idea behind the use of events is that one object can react to a change occurring in another object. Notifications of events are essentially asynchronous and determined by their receivers. In particular, in interactive applications, the actions that the user performs on objects, for example by manipulating a button with the mouse or entering text in a text box via the keyboard, are seen as *events* that cause changes in the objects that maintain the state of the application. The objects that are responsible for displaying a view of the current state are *notified* whenever the state changes.

Distributed event-based systems extend the local event model by allowing multiple objects at different locations to be notified of events taking place at an object. They use the *publish-subscribe* paradigm, in which an object that generates events *publishes* the type of events that it will make available for observation by other objects. Objects that want to receive notifications from an object that has published its events *subscribe* to the types of events that are of interest to them. Different event *types* may, for example, refer to the different methods executed by the object of interest. Objects that represent events are called *notifications*. Notifications may be stored, sent in messages, queried and applied in a variety of orders to different things. When a publisher experiences an event, subscribers that expressed an interest in that type of event will receive notifications. Subscribing to a particular type of event is also called *registering interest* in that type of event.