4

INTERPROCESS COMMUNICATION

This chapter is concerned with the characteristics of protocols for communication between processes in a distributed system – that is, interprocess communication.

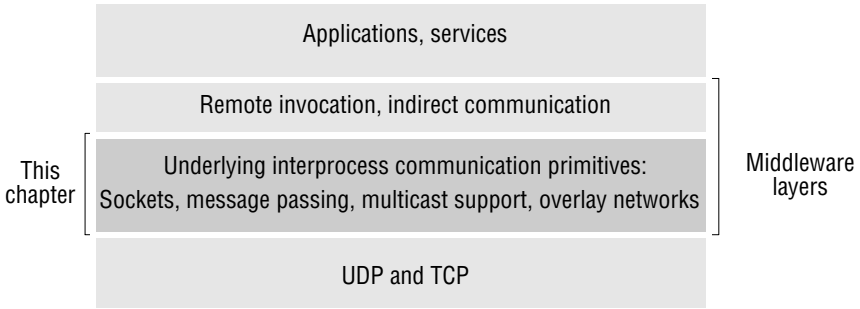
Interprocess communication in the Internet provides both datagram and stream communication. The Java APIs for these are presented, together with a discussion of their failure models. They provide alternative building blocks for communication protocols. This is complemented by a study of protocols for the representation of collections of data objects in messages and of references to remote objects. Together, these services offer support for the construction of higher-level communication services, as discussed in the following two chapters.

The interprocess communication primitives discussed above all support point-to-point communication, yet it is equally useful to be able to send a message from one sender to a group of receivers. The chapter also considers multicast communication, including IP multicast and the key concepts of reliability and ordering of messages in multicast communication.

Multicast is an important requirement for distributed applications and must be provided even if underlying support for IP multicast is not available. This is typically provided by an overlay network constructed on top of the underlying TCP/IP network. Overlay networks can also provide support for file sharing, enhanced reliability and content distribution.

The Message Passing Interface (MPI) is a standard developed to provide an API for a set of message-passing operations with synchronous and asynchronous variants.

Figure 4.1 Middleware layers



4.1 Introduction

This and the next two chapters are concerned with the communication aspects of middleware, although the principles discussed are more widely applicable. This one is concerned with the design of the components shown in the darker layer in Figure 4.1. The layer above it is discussed in Chapter 5, which examines remote invocation, and Chapter 6, which is concerned with indirect communications paradigms.

Chapter 3 discussed 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 each of these two protocols, together with a discussion of their failure models.

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. 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 at the receiving host 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.

Section 4.3 is concerned with how the objects and data structures used in application programs can be translated into a form suitable for sending 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.

Section 4.4 discusses multicast communication: a form of interprocess communication in which one process in a group of processes transmits the same message to all members of the group. After explaining IP multicast, the section discusses the need for more reliable forms of multicast.

Section 4.5 examines the increasingly important topic of overlay networks. An overlay network is a network that is built over another network to permit applications to route messages to destinations not specified by an IP address. Overlay networks can enhance TCP/IP networks by providing alternative, more specialized network services. They are important in supporting multicast communication and peer-to-peer communication.

Finally, Section 4.6 presents a case study of a significant message-passing service, MPI, developed by the high-performance computing community.

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; a case study on the latter is available on the web site for the book [www.cdk5.net/ipc]. Readers studying the programming examples in this section should consult the online Java documentation or Flanagan [2002] 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. To communicate, 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, with a further case study of message passing (MPI) offered in Section 4.6.

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 processes 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 (or thread) is blocked until the corresponding *receive* is issued. Whenever a *receive* is issued by a process (or thread), it 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 no 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, today's 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 (multicast ports are an exception, see Section 4.5.1) but can have many senders. Processes may use multiple ports 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.

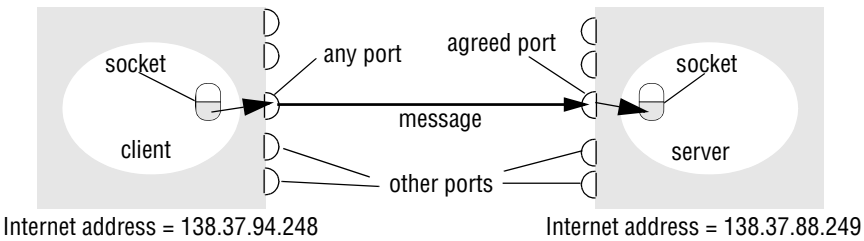
If the client uses a fixed Internet address to refer to a service, then that service must always run on the same computer for its address to remain valid. This can be avoided by using the following approach to providing location transparency:

- Client programs refer to services by name and use a name server or binder (see Section 5.4.2) to translate their names into server locations at runtime. This allows services to be relocated but not to migrate – that is, to be moved while the system is running.

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.

Figure 4.2 Sockets and ports



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 other versions of UNIX, including Linux as well as Windows 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 4.4.1.) 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 System (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 the 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.qmul.ac.uk*, use:

```
InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.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 addresses – 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. To send or receive messages a process must first create a socket bound to an Internet address of the local host and a 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 are 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, for example DNS, will decide on a size that is not excessively large but is adequate for its intended use.

Blocking: Sockets normally provide non-blocking *sends* and blocking *receives* for datagram communication (a non-blocking *receive* is an option in some implementations). 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 7. 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 forever 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 invoked a *receive* operation should wait indefinitely in situations where the sending process may have crashed or the expected message may have 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 the message 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 for UDP datagrams • 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 should not be corrupted or duplicated. The use of a checksum ensures that there is a negligible probability that any message received is corrupted. UDP datagrams 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 2.15) 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 5.2 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 Name System, which looks up DNS names in the Internet, is implemented over UDP. Voice over IP (VOIP) also runs 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:

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

The reasons for these overheads are discussed in Section 4.2.4.

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

An instance 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.

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

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
      // args give message contents and server hostname
      DatagramSocket aSocket = null;
      try {
            aSocket = new DatagramSocket();
            byte [] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request =
                  new DatagramPacket(m,  m.length(), aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
       } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
       } catch (IOException e){System.out.println("IO: " + e.getMessage());
       } finally { if(aSocket != null) aSocket.close();}
   }
}
```

DatagramSocket: This class supports sockets for sending and receiving UDP datagrams. It provides a constructor that takes a port number as its argument, for use by 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 chosen 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 its 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 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 4.4 UDP server repeatedly receives a request and sends it back to the client

```
import java.net.*;
import java.io.*;
public class UDPServer{
   public static void main(String args[]){
      DatagramSocket aSocket = null;
      try{
            aSocket = new DatagramSocket(6789);
            byte[] buffer = new byte[1000];
            while(true){
               DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(),
                      request.getLength(), request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
      } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
      } catch (IOException e) {System.out.println("IO: " + e.getMessage());
      } finally {if (aSocket != null) aSocket.close();}
   }
}
```

Figure 4.3 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) and 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 2006] 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 *accept*s 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.

The pair of sockets in the 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 are 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.
- The communicating processes cannot tell whether the messages they have sent recently 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 in Section 5.2.

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, blocks until one arrives. The result of executing *accept* is an instance of *Socket* – a socket to use for communicating with the client.

Figure 4.5 TCP client makes connection to server, sends request and receives reply

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

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 the 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

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

creates a *DataInputStream* and a *DataOutputStream* from its socket's input and output streams and then waits to read a message and write the same one 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-8 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. 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. Another issue is the set of codes used to represent characters: for example, the majority of applications on systems such as UNIX 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.

One of the following methods can be used to enable any two computers to exchange binary 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.

Three alternative approaches to external data representation and marshalling are discussed (with a fourth considered in Chapter 21, when we examine Google's approach to representing structured data):

- CORBA's common data representation, which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA. It can be used by a variety of programming languages (see Chapter 8).
- 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.
- XML (Extensible Markup Language), which defines a textual fomat for representing structured data. It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services (see Chapter 9).

In the first two 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. Even in the case of XML, which is textual and therefore more accessible to hand-encoding, software for marshalling and unmarshalling is available for all commonly used platforms and programming environments. 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. Compactness is another issue that can be addressed in the design of automatically generated marshalling procedures.

In the first two approaches, the primitive data types are marshalled into a binary form. In the third approach (XML), the primitive data types are represented textually. The textual representation of a data value will generally be longer than the equivalent binary representation. The HTTP protocol, which is described in Chapter 5, is another example of the textual approach.

Another issue with regard to the design of marshalling methods is whether the marshalled data should include information concerning the type of its contents. For example, CORBA's representation includes just the values of the objects transmitted, and nothing about their types. On the other hand, both Java serialization and XML do include type information, but in different ways. Java puts all of the required type information into the serialized form, but XML documents may refer to externally defined sets of names (with types) called *namespaces*.

Although we are interested in the use of an external data representation for the arguments and results of RMIs and RPCs, it does have a more general use for representing data structures, objects or structured documents in a form suitable for transmission in messages or storing in files.

Two other techniques for external data representation are worthy of mention. Google uses an approach called *protocol buffers* to capture representations of both stored and transmitted data, which we examine in Section 20.4.1. There is also considerable interest in JSON (JavaScript Object Notation) as an approach to external data representation [www.json.org]. Protocol buffers and JSON represent a step towards more lightweight approaches to data representation (when compared, for example, to XML).

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 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 2004a]. 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 checksum to be compared. Note that each *unsigned long*, which occupies four bytes,

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	1984	*unsigned long*

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1984}

starts at an index that is a multiple of four. The figure does not distinguish between the big- and little-endian orderings. 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.cdk5.net/ipc. It was developed by Sun for use in the messages exchanged between clients and servers in Sun NFS (see Chapter 13).

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 8.3.1), 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;
        unsigned 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 the 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 and 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 the instance variables of all of the necessary classes have been

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*
1984	5 Smith	6 London	h1	*values of instance variables*

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

written 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 *writeUTF* method using the Universal Transfer Format (UTF-8), 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", 1984);

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 (1984) 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 its 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 middleware, 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 [2002].

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 Extensible Markup Language (XML)

XML is a markup language that was defined by the World Wide Web Consortium (W3C) for general use on the Web. In general, the term *markup language* refers to a textual encoding that represents both a text and details as to its structure or its appearance. Both XML and HTML were derived from SGML (Standardized Generalized Markup Language) [ISO 8879], a very complex markup language. HTML (see Section 1.6) was designed for defining the appearance of web pages. XML was designed for writing structured documents for the Web.

XML data items are tagged with 'markup' strings. The tags are used to describe the logical structure of the data and to associate attribute-value pairs with logical structures. That is, in XML, the tags relate to the structure of the text that they enclose, in contrast to HTML, in which the tags specify how a browser could display the text. For a specification of XML, see the pages on XML provided by W3C [www.w3.org VI].

XML is used to enable clients to communicate with web services and for defining the interfaces and other properties of web services. However, XML is also used in many other ways, including in archiving and retrieval systems – although an XML archive may be larger than a binary one, it has the advantage of being readable on any computer. Other examples of uses of XML include for the specification of user interfaces and the encoding of configuration files in operating systems.

XML is *extensible* in the sense that users can define their own tags, in contrast to HTML, which uses a fixed set of tags. However, if an XML document is intended to be used by more than one application, then the names of the tags must be agreed between them. For example, clients usually use SOAP messages to communicate with web services. SOAP (see Section 9.2.1) is an XML format whose tags are published for use by web services and their clients.

Some external data representations (such as CORBA CDR) do not need to be self-describing, because it is assumed that the client and server exchanging a message have prior knowledge of the order and the types of the information it contains. However, XML was intended to be used by multiple applications for different purposes. The provision of tags, together with the use of namespaces to define the meaning of the tags, has made this possible. In addition, the use of tags enables applications to select just those parts of a document it needs to process: it will not be affected by the addition of information relevant to other applications.

Figure 4.10 XML definition of the *Person* structure

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!-- a comment -->
</person >
```

XML documents, being textual, can be read by humans. In practice, most XML documents are generated and read by XML processing software, but the ability to read XML can be useful when things go wrong. In addition, the use of text makes XML independent of any particular platform. The use of a textual rather than a binary representation, together with the use of tags, makes the messages large, so they require longer processing and transmission times, as well as more space to store. A comparison of the efficiency of messages using the SOAP XML format and CORBA CDR is given in Section 9.2.4. However, files and messages can be compressed – HTTP version 1.1 allows data to be compressed, which saves bandwidth during transmission.

XML elements and attributes • Figure 4.10 shows the XML definition of the *Person* structure that was used to illustrate marshalling in CORBA CDR and Java. It shows that XML consists of tags and character data. The character data, for example *Smith* or *1984*, is the actual data. As in HTML, the structure of an XML document is defined by pairs of tags enclosed in angle brackets. In Figure 4.10, <*name*> and <*place*> are both tags. As in HTML, layout can generally be used to improve readability. Comments in XML are denoted in the same way as those in HTML.

Elements: An element in XML consists of a portion of character data surrounded by matching start and end tags. For example, one of the elements in Figure 4.10 consists of the data *Smith* contained within the <*name*> ... </*name*> tag pair. Note that the element with the <*name*> tag is enclosed in the element with the <*person id="123456789"*> ... </*person* > tag pair. The ability of an element to enclose another element allows hierarchic data to be represented – a very important aspect of XML. An empty tag has no content and is terminated with /> instead of >. For example, the empty tag <*european*/> could be included within the <*person*> ...</*person*> tag.

Attributes: A start tag may optionally include pairs of associated attribute names and values such as *id="123456789",* as shown above. The syntax is the same as for HTML, in which an attribute name is followed by an equal sign and an attribute value in quotes. Multiple attribute values are separated by spaces.

It is a matter of choice as to which items are represented as elements and which ones as attributes. An element is generally a container for data, whereas an attribute is used for labelling that data. In our example, *123456789* might be an identifier used by the application, whereas *name*, *place* and *year* might be displayed. Also, if data contains substructures or several lines, it must be defined as an element. Attributes are for simple values.

Names: The names of tags and attributes in XML generally start with a letter, but can also start with an underline or a colon. The names continue with letters, digits, hyphens, underscores, colons or full stops. Letters are case-sensitive. Names that start with *xml* are reserved.

Binary data: All of the information in XML elements must be expressed as character data. But the question is: how do we represent encrypted elements or secure hashes – both of which, as we shall see in Section 9.5 are used in XML security? The answer is that they can be represented in *base64* notation [Freed and Borenstein 1996], which uses only the alphanumeric characters together with +, / and =, which has a special meaning.

Parsing and well-formed documents • An XML document must be well formed – that is, it must conform to rules about its structure. A basic rule is that every start tag has a matching end tag. Another basic rule is that all tags are correctly nested – for example, <x>..<y>..</y>..</x> is correct, whereas <x>..<y>..</x>..</y> is not. Finally, every XML document must have a single root element that encloses all the other elements. These rules make it very simple to implement parsers for XML documents. When a parser reads an XML document that is not well formed, it will report a fatal error.

CDATA: XML parsers normally parse the contents of elements because they may contain further nested structures. If text needs to contain an angle bracket or a quote, it may be represented in a special way: for example, *&lt* represents the opening angle bracket. However, if a section should not be parsed – for example, because it contains special characters – it can be denoted as *CDATA*. For example, if a place name is to include an apostrophe, then it could be specified in either of the two following ways:

<place> King&apos Cross </place >
<place> <![CDATA [King's Cross]]></place >

XML prolog: Every XML document must have a *prolog* as its first line. The prolog must at least specify the version of XML in use (which is currently 1.0). For example:

<?XML version = "1.0" encoding = "UTF-8" standalone = "yes"?>

The prolog may specify the encoding (UTF-8 is the default and was explained in Section 4.3.2). The term *encoding* refers to the set of codes used to represent characters – ASCII being the best-known example. Note that in the XML prolog, ASCII is specified as *us-ascii*. Other possible encodings include ISO-8859-1 (or Latin-1) – an 8-bit encoding whose first 128 values are ASCII, with the rest being used to represent the characters in Western European languages – and various other 8-bit encodings for representing other alphabets, for example, Greek or Cyrillic.

An additional attribute may be used to state whether the document stands alone or is dependent on external definitions.

XML namespaces • Traditionally, namespaces provide a means for scoping names. An XML namespace is a set of names for a collection of element types and attributes that is referenced by a URL. Any other XML document can use an XML namespace by referring to its URL.

Any element that makes use of an XML namespace can specify that namespace as an attribute called *xmlns*, whose value is a URL referring to the file containing the namespace definitions. For example:

xmlns:pers = "http://www.cdk5.net/person"

Figure 4.11 Illustration of the use of a namespace in the *Person* structure

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place >
    <pers:year> 1984 </pers:year>
</person>
```

The name after *xmlns,* in this case *pers* can be used as a prefix to refer to the elements in a particular namespace, as shown in Figure 4.11. The *pers* prefix is bound to *http://www.cdk4.net/person* for the *person* element. A namespace applies within the context of the enclosing pair of start and end tags unless overridden by an enclosed namespace declaration. An XML document may be defined in terms of several different namespaces, each of which is referenced by a unique prefix.

The namespace convention allows an application to make use of multiple sets of external definitions in different namespaces without the risk of name clashes.

XML schemas • An XML schema [www.w3.org VIII] defines the elements and attributes that can appear in a document, how the elements are nested and the order and number of elements, and whether an element is empty or can include text. For each element, it defines the type and default value. Figure 4.12 gives an example of a schema that defines the data types and structure of the XML definition of the *person* structure in Figure 4.10.

The intention is that a single schema definition may be shared by many different documents. An XML document that is defined to conform to a particular schema may also be validated by means of that schema. For example, the sender of a SOAP message may use an XML schema to encode it, and the recipient will use the same XML schema to validate and decode it.

Figure 4.12 An XML schema for the *Person* structure

```
<xsd:schema  xmlns:xsd = URL of XML schema definitions >
    <xsd:element name= "person" type ="personType" />
        <xsd:complexType name="personType">
            <xsd:sequence>
                <xsd:element name = "name"  type="xs:string"/>
                <xsd:element name = "place"  type="xs:string"/>
                <xsd:element name = "year"  type="xs:positiveInteger"/>
            </xsd:sequence>
            <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
        </xsd:complexType>
</xsd:schema>
```

Document type definitions: Document type definitions (DTDs) [www.w3.org VI] were provided as a part of the XML 1.0 specification for defining the structure of XML documents and are still widely used for that purpose. The syntax of DTDs is different from the rest of XML and it is quite limited in what it can specify; for example, it cannot describe data types and its definitions are global, preventing element names from being duplicated. DTDs are not used for defining web services, although they may still be used to define documents that are transmitted by web services.

APIs for accessing XML • XML parsers and generators are available for most commonly used programming languages. For example, there is Java software for writing out Java objects as XML (marshalling) and for creating Java objects from such structures (unmarshalling). Similar software is available in Python for Python data types and objects.

4.3.4 Remote object references

This section applies only to languages such as Java and CORBA that support the distributed object model. It is not relevant to XML.

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. Here, we 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 host 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 format such as that shown in Figure 4.13. 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 the address of the remote object. In other words, invocation messages are sent to the Internet

Figure 4.13 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

address in the remote reference and to the process on that computer using the given port number.

To allow remote objects to be relocated into a different process on a different computer, the remote object reference should not be used as the address of the remote object. Section 8.3.3 discusses a form of remote object reference that allows objects to be activated in different servers throughout its lifetime.

The peer-to-peer overlay systems described in Chapter 10 use a form of remote object reference that is completely independent of location. Messages are routed to resources by means of a distributed routing algorithm.

The last field of the remote object reference shown in Figure 4.13 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 as the 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 5.4.2.

4.4 Multicast communication

The pairwise exchange of messages is not the best model for communication from one process to a group of other processes, which may be necessary, 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 multicast protocol provides no guarantees about message delivery or ordering.

Multicast messages provide a useful infrastructure for constructing distributed systems with the following characteristics:

1. *Fault tolerance based on replicated services*: A replicated service consists of a group of servers. Client requests are multicast to all the members of the group, each of which performs an identical operation. Even when some of the members fail, clients can still be served.

2. *Discovering services in spontaneous networking*: Section 1.3.2 defines service discovery in the context of spontaneous networking. Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.

3. *Better performance through replicated data*: Data are replicated to increase the performance of a service – in some cases replicas of the data are placed in users' computers. Each time the data changes, the new value is multicast to the processes managing the replicas.

4. *Propagation of event notifications*: Multicast to a group may be used to notify processes when something happens. For example, in Facebook, when someone changes their status, all their friends receive notifications. Similarly, publish-subscribe protocols may make use of group multicast to disseminate events to subscribers (see Chapter 6).

In this section introduce IP multicast and then review the needs of the above uses of group communication to see which of them can be satisfied by IP multicast. For those that cannot, we propose some further properties for group communication protocols in addition to those provided by IP multicast.

4.4.1 IP multicast – An implementation of multicast communication

This section discusses IP multicast and presents Java's API to it via the *MulticastSocket* class.

IP multicast • *IP multicast* is built on top of the Internet Protocol (IP). Note that IP packets are addressed to computers – ports belong to the TCP and UDP levels. IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group. The sender is unaware of the identities of the individual recipients and of the size of the group. A *multicast group* is specified by a Class D Internet address (see Figure 3.15) – that is, an address whose first 4 bits are 1110 in IPv4.

Being a member of a multicast group allows a computer to receive IP packets sent to the group. The membership of multicast groups is dynamic, allowing computers to join or leave at any time and to join an arbitrary number of groups. It is possible to send datagrams to a multicast group without being a member.

At the application programming level, IP multicast is available only via UDP. An application program performs multicasts by sending UDP datagrams with multicast addresses and ordinary port numbers. It can join a multicast group by making its socket join the group, enabling it to receive messages to the group. At the IP level, a computer belongs to a multicast group when one or more of its processes has sockets that belong to that group. When a multicast message arrives at a computer, copies are forwarded to all of the local sockets that have joined the specified multicast address and are bound to the specified port number. The following details are specific to IPv4:

Multicast routers: IP packets can be multicast both on a local network and on the wider Internet. Local multicasts use the multicast capability of the local network, for example, of an Ethernet. Internet multicasts make use of multicast routers, which forward single datagrams to routers on other networks, where they are again multicast to local members. To limit the distance of propagation of a multicast datagram, the sender can specify the number of routers it is allowed to pass – called the *time to live*, or TTL for short. To understand how routers know which other routers have members of a multicast group, see Comer [2007].

Multicast address allocation: As discussed in Chapter 3, Class D addresses (that is, addresses in the range 224.0.0.0 to 239.255.255.255) are reserved for multicast traffic and managed globally by the Internet Assigned Numbers Authority (IANA). The management of this address space is reviewed annually, with current practice documented in RPC 3171 [Albanna *et al.* 2001]. This document defines a partitioning of this address space into a number of blocks, including:

- Local Network Control Block (224.0.0.0 to 224.0.0.225), for multicast traffic within a given local network.
- Internet Control Block (224.0.1.0 to 224.0.1.225).
- Ad Hoc Control Block (224.0.2.0 to 224.0.255.0), for traffic that does not fit any other block.
- Administratively Scoped Block (239.0.0.0 to 239.255.255.255), which is used to implement a scoping mechanism for multicast traffic (to constrain propagation).

Multicast addresses may be permanent or temporary. Permanent groups exist even when there are no members – their addresses are assigned by IANA and span the various blocks mentioned above. For example, 224.0.1.1 in the Internet block is reserved for the Network Time Protocol (NTP), as discussed in Chapter 14, and the range 224.0.6.000 to 224.0.6.127 in the ad hoc block is reserved for the ISIS project (see Chapters 6 and 18). Addresses are reserved for a variety of purposes, from specific Internet protocols to given organizations that make heavy use of multicast traffic, including multimedia broadcasters and financial institutions. A full list of reserved addresses can be seen on the IANA web site [www.iana.org II].

The remainder of the multicast addresses are available for use by temporary groups, which must be created before use and cease to exist when all the members have left. When a temporary group is created, it requires a free multicast address to avoid accidental participation in an existing group. The IP multicast protocol does not directly address this issue. If used locally, relatively simple solutions are possible – for example setting the TTL to a small value, making collisions with other groups unlikely. However, programs using IP multicast throughout the Internet require a more sophisticated solution to this problem. RFC 2908 [Thaler *et al.* 2000] describes a multicast address allocation architecture (MALLOC) for Internet-wide applications, that allocates unique addresses for a given period of time and in a given scope. As such, the proposal is intrinsically bound with the scoping mechanisms mentioned above. A client-server solution is adopted whereby clients request a multicast address from a multicast address allocation server (MAAS), which must then communicate across domains to ensure allocations are unique for the given lifetime and scope.

Failure model for multicast datagrams • Datagrams multicast over IP multicast have the same failure characteristics as UDP datagrams – that is, they suffer from omission failures. The effect on a multicast is that messages are not guaranteed to be delivered to any particular group member in the face of even a single omission failure. That is, some but not all of the members of the group may receive it. This can be called *unreliable* multicast, because it does not guarantee that a message will be delivered to any member of a group. Reliable multicast is discussed in Chapter 15.

Figure 4.14 Multicast peer joins a group and sends and receives datagrams

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
    public static void main(String args[]){
      // args give message contents & destination multicast group (e.g. "228.5.6.7")
       MulticastSocket s =null;
       try {
          InetAddress group = InetAddress.getByName(args[1]);
          s = new MulticastSocket(6789);
          s.joinGroup(group);
          byte [] m = args[0].getBytes();
          DatagramPacket messageOut =
                new DatagramPacket(m, m.length, group, 6789);
          s.send(messageOut);
          byte[] buffer = new byte[1000];
          for(int i=0; i< 3; i++) {    // get messages from others in group
                DatagramPacket messageIn =
                     new DatagramPacket(buffer, buffer.length);
                s.receive(messageIn);
                System.out.println("Received:" + new String(messageIn.getData()));
          }
          s.leaveGroup(group);
       } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
       } catch (IOException e){System.out.println("IO: " + e.getMessage());
       } finally { if(s != null) s.close();}
    }
}
```

Java API to IP multicast • The Java API provides a datagram interface to IP multicast through the class *MulticastSocket*, which is a subclass of *DatagramSocket* with the additional capability of being able to join multicast groups. The class *MulticastSocket* provides two alternative constructors, allowing sockets to be created to use either a specified local port (6789, in Figure 4.14) or any free local port. A process can join a multicast group with a given multicast address by invoking the *joinGroup* method of its multicast socket. Effectively, the socket joins a multicast group at a given port and it will receive datagrams sent by processes on other computers to that group at that port. A process can leave a specified group by invoking the *leaveGroup* method of its multicast socket.

In the example in Figure 4.14, the arguments to the *main* method specify a message to be multicast and the multicast address of a group (for example, *"228.5.6.7").* After joining that multicast group, the process makes an instance of *DatagramPacket* containing the message and sends it through its multicast socket to the multicast group address at port 6789. After that, it attempts to receive three multicast messages from its

peers via its socket, which also belongs to the group on the same port. When several instances of this program are run simultaneously on different computers, all of them join the same group, and each of them should receive its own message and the messages from those that joined after it.

The Java API allows the TTL to be set for a multicast socket by means of the *setTimeToLive* method. The default is 1, allowing the multicast to propagate only on the local network.

An application implemented over IP multicast may use more than one port. For example, the MultiTalk [mbone] application, which allows groups of users to hold text-based conversations, has one port for sending and receiving data and another for exchanging control data.

4.4.2 Reliability and ordering of multicast

The previous section stated the failure model for IP multicast, which suffers from omission failures. A datagram sent from one multicast router to another may be lost, thus preventing all recipients beyond that router from receiving the message. Also, when a multicast on a local area network uses the multicasting capabilities of the network to allow a single datagram to arrive at multiple recipients, any one of those recipients may drop the message because its buffer is full.

Another factor is that any process may fail. If a multicast router fails, the group members beyond that router will not receive the multicast message, although local members may do so.

Ordering is another issue. IP packets sent over an internetwork do not necessarily arrive in the order in which they were sent, with the possible effect that some group members receive datagrams from a single sender in a different order from other group members. In addition, messages sent by two different processes will not necessarily arrive in the same order at all the members of the group.

Some examples of the effects of reliability and ordering • We now consider the effect of the failure semantics of IP multicast on the four examples of the use of replication in the introduction to Section 4.4.

1. *Fault tolerance based on replicated services*: Consider a replicated service that consists of the members of a group of servers that start in the same initial state and always perform the same operations in the same order, so as to remain consistent with one another. This application of multicast requires that either all of the replicas or none of them should receive each request to perform an operation – if one of them misses a request, it will become inconsistent with the others. In most cases, this service would require that all members receive request messages in the same order as one another.

3. *Discovering services in spontaneous networking*: One way for a process to discover services in spontaneous networking is to multicast requests at periodic intervals, and for the available services to listen for those multicasts and respond. An occasional lost request is not an issue when discovering services. In fact, Jini uses IP multicast in its protocol for discovering services. This is described in Section 19.2.1.

3. *Better performance through replicated data*: Consider the case where the replicated data itself, rather than operations on the data, are distributed by means of multicast messages. The effect of lost messages and inconsistent ordering would depend on the method of replication and the importance of all replicas being totally up-to-date.
4. *Propagation of event notifications*: The particular application determines the qualities required of multicast. For example, the Jini lookup services use IP multicast to announce their existence (see Section 19.2.1).

These examples suggest that some applications require a multicast protocol that is more reliable than IP multicast. In particular, there is a need for *reliable multicast*, in which any message transmitted is either received by all members of a group or by none of them. The examples also suggest that some applications have strong requirements for ordering, the strictest of which is called *totally ordered multicast*, in which all of the messages transmitted to a group reach all of the members in the same order.

Chapter 15 will define and show how to implement reliable multicast and various useful ordering guarantees, including totally ordered multicast.

4.5 Network virtualization: Overlay networks

The strength of the Internet communication protocols is that they provide, through their API (Section 4.2), a very effective set of building blocks for the construction of distributed software. However, a growing range of different classes of application (including, for example, peer-to-peer file sharing and Skype) coexist in the Internet. It would be impractical to attempt to alter the Internet protocols to suit each of the many applications running over them – what might enhance one of them could be detrimental to another. In addition, the IP transport service is implemented over a large and ever-increasing number of network technologies. These two factors have led to the interest in network virtualization.

Network virtualization [Petersen *et al*. 2005] is concerned with the construction of many different virtual networks over an existing network such as the Internet. Each virtual network can be designed to support a particular distributed application. For example, one virtual network might support multimedia streaming, as in BBC iPlayer, BoxeeTV [boxee.tv] or Hulu [hulu.com], and coexist with another that supports a multiplayer online game, both running over the same underlying network. This suggests an answer to the dilemma raised by Salzer's end-to-end argument (see Section 2.3.3): an application-specific virtual network can be built above an existing network and optimized for that particular application, without changing the characteristics of the underlying network.

Chapter 3 showed that computer networks have addressing schemes, protocols and routing algorithms; similarly, each virtual network has its own particular addressing scheme, protocols and routing algorithms, but redefined to meet the needs of particular application classes.

4.5.1 Overlay networks

An *overlay network* is a virtual network consisting of nodes and virtual links, which sits on top of an underlying network (such as an IP network) and offers something that is not otherwise provided:

- a service that is tailored towards the needs of a class of application or a particular higher-level service – for example, multimedia content distribution;
- more efficient operation in a given networked environment – for example routing in an ad hoc network;
- an additional feature – for example, multicast or secure communication.

This leads to a wide variety of types of overlay as captured by Figure 4.15. Overlay networks have the following advantages:

- They enable new network services to be defined without requiring changes to the underlying network, a crucial point given the level of standardization in this area and the difficulties of amending underlying router functionality.
- They encourage experimentation with network services and the customization of services to particular classes of application.
- Multiple overlays can be defined and can coexist, with the end result being a more open and extensible network architecture.

The disadvantages are that overlays introduce an extra level of indirection (and hence may incur a performance penalty) and they add to the complexity of network services when compared, for example, to the relatively simple architecture of TCP/IP networks.

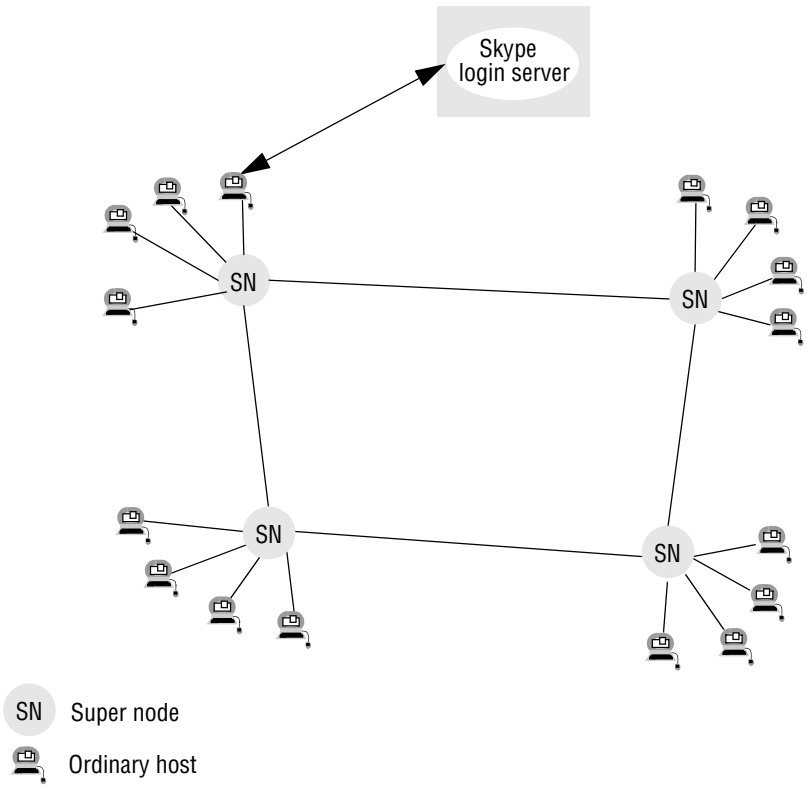
Overlays can be related to the familiar concept of layers (as introduced in Chapters 2 and 3). Overlays are layers, but layers that exist outside the standard architecture (such as the TCP/IP stack) and exploit the resultant degrees of freedom. In particular, overlay developers are free to redefine the core elements of a network as mentioned above, including the mode of addressing, the protocols employed and the approach to routing, often introducing radically different approaches more tailored towards particular application classes of operating environments. For example, distributed hash tables introduce a style of addressing based on a keyspace and also build a topology in such a way that a node in the topology either owns the key or has a link to a node that is closer to the owner (a style of routing known as *key-based routing*). The topology is most commonly in the form of a ring.

We exemplify the successful use of an overlay network by discussing Skype. Further examples of overlays will be given throughout the book. For example, Chapter 10 presents details of the protocols and structures adopted by peer-to-peer file sharing, along with further information on distributed hash tables. Chapter 19 considers both wireless ad hoc networks and disruption-tolerant networks in the context of mobile and ubiquitous computing and Chapter 20 examines overlay support for multimedia streaming.

Figure 4.15 Types of overlay

Motivation	*Type*	*Description*
Tailored for application needs	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard hash table but in a networked environment).
	Peer-to-peer file sharing	Overlay structures that focus on constructing tailored addressing and routing mechanisms to support the cooperative discovery and use (for example, download) of files.
	Content distribution networks	Overlays that subsume a range of replication, caching and placement strategies to provide improved performance in terms of content delivery to web users; used for web acceleration and to offer the required real-time performance for video streaming [www.kontiki.com].
Tailored for network style	Wireless ad hoc networks	Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically supported by flooding.
	Disruption-tolerant networks	Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.
Offering additional features	Multicast	One of the earliest uses of overlay networks in the Internet, providing access to multicast services where multicast routers are not available; builds on the work by Van Jacobsen, Deering and Casner with their implementation of the MBone (or Multicast Backbone) [mbone].
	Resilience	Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [nms.csail.mit.edu].
	Security	Overlay networks that offer enhanced security over the underling IP network, including virtual private networks, for example, as discussed in Section 3.4.8.

Figure 4.16 Skype overlay architecture



4.5.2 Skype: An example of an overlay network

Skype is a peer-to-peer application offering Voice over IP (VoIP). It also includes instant messaging, video conferencing and interfaces to the standard telephony service through SkypeIn and SkypeOut. The software was developed by Kazaa in 2003 and hence shares many of the characteristics of the Kazaa peer-to-peer file-sharing application [Leibowitz *et al.* 2003]. It is widely deployed, with an estimated 370 million users as of the start of 2009.

Skype is an excellent case study of the use of overlay networks in real-world (and large-scale) systems, indicating how advanced functionality can be provided in an application-specific manner and without modification of the core architecture of the Internet. Skype is a virtual network in that it establishes connections between people (Skype subscribers who are currently active). No IP address or port is required to establish a call. The architecture of the virtual network supporting Skype is not widely publicized but researchers have studied Skype through a variety of methods, including traffic analysis, and its principles are now in the public domain. Much of the detail of the description that follows is taken from the paper by Baset and Schulzrinne [2006], which contains a detailed study of the behaviour of Skype.

Skype architecture • Skype is based on a peer-to-peer infrastructure consisting of ordinary users' machines (referred to as hosts) and super nodes – super nodes are ordinary Skype hosts that happen to have sufficient capabilities to carry out their enhanced role. Super nodes are selected on demand based a range of criteria including bandwidth available, reachability (the machine must have a global IP address and not be hidden behind a NAT-enabled router, for example) and availability (based on the length of time that Skype has been running continuously on that node). This overall structure is captured in Figure 4.16.

User connection • Skype users are authenticated via a well-known login server. They then make contact with a selected super node. To achieve this, each client maintains a cache of super node identities (that is, IP address and port number pairs). At first login this cache is filled with the addresses of around seven super nodes, and over time the client builds and maintains a much larger set (perhaps several hundred).

Search for users • The main goal of super nodes is to perform the efficient search of the global index of users, which is distributed across the super nodes. The search is orchestrated by the client's chosen super node and involves an expanding search of other super nodes until the specified user is found. On average, eight super nodes are contacted. A user search typically takes between three and four seconds to complete for hosts that have a global IP address (and slightly longer, five to six seconds, if behind a NAT-enabled router). From experiments, it appears that intermediary nodes involved in the search cache the results to improve performance.

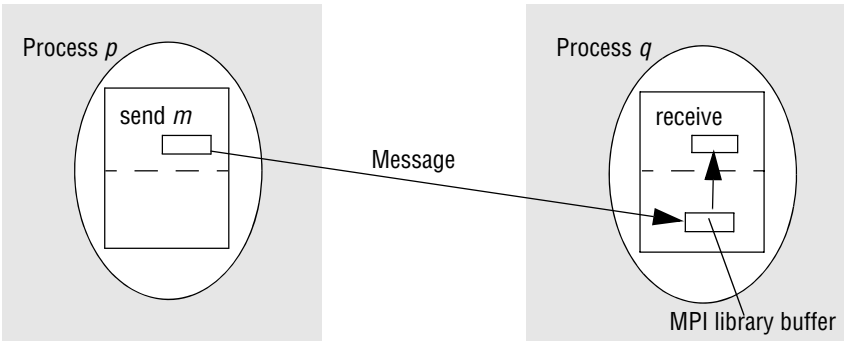
Voice connection • Once the required user is discovered, Skype establishes a voice connection between the two parties using TCP for signalling call requests and terminations and either UDP or TCP for the streaming audio. UDP is preferred but TCP, along with the use of an intermediary node, is used in certain circumstances to circumvent firewalls (see Baset and Schulzrinne [2006] for details). The software used for encoding and decoding audio plays a key part in providing the excellent call quality normally attained using Skype, and the associated algorithms are carefully tailored to operate in Internet environments at 32 kbps and above.

4.6 Case study: MPI

Message passing was introduced in Section 4.2.1, which outlines the basic principles of exchanging messages between two processes using *send* and *receive* operations. The synchronous variant of message passing is realised by blocking *send* and *receive* calls, whereas the asynchronous variant requires a non-blocking form of *send*. The end result is a paradigm for distributed programming that is lightweight, efficient and in many ways minimal.

This style of distributed programming is attractive in classes of system where performance is paramount, most notably in high-performance computing. In this section, we present a case study of the Message Passing Interface standard, developed by the high performance computing community. MPI was first introduced in 1994 by the MPI Forum [www.mpi-forum.org] as a reaction against the wide variety of proprietary approaches that were in use for message passing in this field. The standard

Figure 4.17 An overview of point-to-point communication in MPI



has also been strongly influential in Grid computing (discussed in Chapter 9), for example through the development of GridMPI [www.gridmpi.org]. The goal of the MPI Forum was to retain the inherent simplicity, practicality and efficiency of the message-passing approach but enhance this with portability through presenting a standardized interface independent of the operating system or programming language-specific socket interface. MPI was also designed to be flexible, and the result is a comprehensive specification of message passing in all its variants (with over 115 operations). Applications use the MPI interface via a message-passing library available for a variety of operating systems and programming languages, including C++ and Fortran.

The underlying architectural model for MPI is relatively simple and captured in Figure 4.17. This is similar to the model introduced in Section 4.2.1, but with the added dimension of explicitly having MPI library buffers in both the sender and the receiver, managed by the MPI library and used to hold data in transit. Note that this figure shows one pathway from the sender to the receiver via the receiver's MPI library buffer (other options, for example using the sender's MPI library buffer, will become apparent below).

To provide a flavour of this complexity, let us examine a number of the variants of *send* summarized in Figure 4.18. This is a refinement of the view of message passing as presented in Section 4.2.1, offering more choice and control and effectively separating the semantics of synchronous/asynchronous and blocking/non-blocking message passing.

We start by examining the four blocking operations presented in the associated column of Figure 4.18. The key to understanding this set of operations is to appreciate that blocking is interpreted as 'blocked until it is safe to return', in the sense that application data has been copied into the MPI system and hence is in transit or delivered and therefore the application buffer can be reused (for example, for the next *send* operation). This then enables various interpretations of 'being safe to return'. The *MPI_Send* operation is a generic operation that simply requires that this level of safety is provided (in practice, this is often implemented using *MPI_Ssend*). *MPI_Ssend* is exactly the same as synchronous (and blocking) message passing as introduced in Section 4.2.1, with safety interpreted as delivered, whereas *MPI_Bsend* has weaker

Figure 4.18 Selected send operations in MPI

Send operations	*Blocking*	*Non-blocking*
Generic	*MPI_Send*: the sender blocks until it is safe to return – that is, until the message is in transit or delivered and the sender's application buffer can therefore be reused.	*MPI_Isend*: the call returns immediately and the programmer is given a communication request handle, which can then be used to check the progress of the call via *MPI_Wait* or *MPI_Test.*
Synchronous	*MPI_Ssend*: the sender and receiver synchronize and the call only returns when the message has been delivered at the receiving end.	*MPI_Issend*: as with *MPI_Isend*, but with *MPI_Wait* and *MPI_Test* indicating whether the message has been delivered at the receive end.
Buffered	*MPI_Bsend*: the sender explicitly allocates an MPI buffer library (using a separate *MPI_Buffer_attach* call) and the call returns when the data is successfully copied into this buffer.	*MPI_Ibsend*: as with *MPI_Isend* but with *MPI_Wait* and *MPI_Test* indicating whether the message has been copied into the sender's MPI buffer and hence is in transit.
Ready	*MPI_Rsend*: the call returns when the sender's application buffer can be reused (as with *MPI_Send*), but the programmer is also indicating to the library that the receiver is ready to receive the message, resulting in potential optimization of the underlying implementation.	*MPI_Irsend*: the effect is as with *MPI_Isend*, but as with *MPI_Rsend*, the programmer is indicating to the underlying implementation that the receiver is guaranteed to be ready to receive (resulting in the same optimizations),

semantics in that the message is considered safe when it has been has been copied into the preallocated MPI library buffer and is still in transit. *MPI_Rsend* is a rather curious operation in which the programmer specifies that they know that the receiver is ready to receive the message. If this is known, the underlying implementation can be optimized in that there is no need to check if there is a buffer available to receive the message, avoiding a handshake. This is clearly a rather dangerous operation that will fail if the assumption about being ready is invalid. From the figure, it is possible to observe the elegant symmetry for non-blocking send operations, this time defined over the semantics of the associated *MPI_Wait* and *MPI_Test* operations (note also the consistent naming convention across all the operations).

The standard also supports both blocking and non-blocking receive (*MPI_recv* and *MPI_Irecv*, respectively), and the variants of send and receive can be paired in any combination, offering the programmer rich control over the semantics of message passing. In addition, the standard defines rich patterns of multiway communication (referred to as collective communication) including, for example, scatter (one to many) and gather (many to one) operations.

4.7 Summary

The first section of this chapter showed that the Internet transmission protocols provide two alternative building blocks from which application protocols may be constructed. There is an interesting trade-off between the two protocols: UDP provides a simple message-passing facility that suffers from omission failures but carries no built-in performance penalties, on the other hand, in good conditions TCP guarantees message delivery, but at the expense of additional messages and higher latency and storage costs.

The second section showed three alternative styles of marshalling. CORBA and its predecessors choose to marshal data for use by recipients that have prior knowledge of the types of its components. In contrast, when Java serializes data, it includes full information about the types of its contents, allowing the recipient to reconstruct it purely from the content. XML, like Java, includes full type information. Another big difference is that CORBA requires a specification of the types of data items to be marshalled (in IDL) in order to generate the marshalling and unmarshalling methods, whereas Java uses reflection in order to serialize objects and deserialize their serial form. But a variety of different means are used for generating XML, depending on the context. For example, many programming languages, including Java, provide processors for translating between XML and language-level objects.

Multicast messages are used in communication between the members of a group of processes. IP multicast provides a multicast service for both local area networks and the Internet. This form of multicast has the same failure semantics as UDP datagrams, but in spite of suffering from omission failures it is a useful tool for many applications of multicast. Some other applications have stronger requirements – in particular, that multicast delivery should be atomic; that is, it should have all-or-nothing delivery. Further requirements on multicast are related to the ordering of messages, the strongest of which requires that all members of a group receive all of the messages in the same order.

Multicast can also be supported by overlay networks in cases where, for example, IP multicast is not supported. More generally, overlay networks offer a service of virtualization of the network architecture, allowing specialist network services to be created on top of underlying networking infrastructure, (for example, UDP or TCP). Overlay networks partially address the problems associated with Saltzer's end-to-end argument by allowing the generation of more application-specific network abstractions.

The chapter concluded with a case study of the MPI specification, developed by the high-performance computing community and featuring flexible support for message passing together with additional support for multiway message passing.

EXERCISES

4.1 In synchronous communication, how do the *send* and *receive* operations work? *page 164*

4.2 What is socket abstraction? Name the main protocols used in interprocess communication. *page 165*

4.3 The programs in Figure 4.3 and Figure 4.4 are available at www.cdk5.net/ipc. Use them to make a test kit to determine the conditions in which datagrams are sometimes dropped. Hint: the client program should be able to vary the number of messages sent and their size; the server should detect when a message from a particular client is missed. *page 166*

4.4 Use the program in Figure 4.3 to make a client program that repeatedly reads a line of input from the user, sends it to the server in a UDP datagram message, then receives a message from the server. The client sets a timeout on its socket so that it can inform the user when the server does not reply. Test this client program with the server program in Figure 4.4. *page 166*

4.5 The programs in Figure 4.5 and Figure 4.6 are available at www.cdk5.net/ipc. Modify them so that the client repeatedly takes a line of user's input and writes it to the stream and the server reads repeatedly from the stream, printing out the result of each read. Make a comparison between sending data in UDP datagram messages and over a stream. *page 169*

4.6 Write one server and one client program. Test whether more than one client program can simultaneously interact with the server program. *page 169*

4.7 Describe the three alternative approaches to external data representation and marshalling. *page 175*

4.8 Sun XDR aligns each primitive value on a 4-byte boundary, whereas CORBA CDR aligns a primitive value of size *n* on an *n*-byte boundary. Discuss the trade-offs in choosing the sizes occupied by primitive values. *page 176*

4.9 What are the different ways in which CORBA CDR can represent constructed types? *page 176*

4.10 Write an algorithm in pseudo-code to describe the serialization procedure described in Section 4.3.2. The algorithm should show when handles are defined or substituted for classes and instances. Describe the serialized form that your algorithm would produce when serializing an instance of the following class, *Couple*:

```
class Couple implements Serializable{
        private Person one;
        private Person two;
        public Couple(Person a, Person b) {
            one = a;
            two = b;
        }
}
```

page 178