Interprocess communication is at the heart of all distributed systems. It makes no sense to study distributed systems without carefully examining the ways that processes on different machines can exchange information. Communication in distributed systems is always based on low-level message passing as offered by the underlying network. As we explained in the previous chapter, expressing communication through message passing is harder than using primitives based on shared memory. Modern distributed systems often consist of thousands or even millions of processes scattered across an unreliable network such as the Internet. Unless the primitive communication facilities of computer networks are replaced by something else, development of large-scale distributed applications is extremely difficult.

In this chapter, we start by discussing the rules that communicating processes must adhere to, known as protocols, and concentrate on structuring those protocols in the form of layers. We then look at four widely-used models for communication: Remote Procedure Call (RPC), Remote Method Invocation (RMI), Message-Oriented Middleware (MOM), and streams.

Our first model for communication in distributed systems is the remote procedure call (RPC). An RPC aims at hiding most of the intricacies of message passing, and is ideal for client-server applications. An improvement to the RPC model comes in the form of remote method invocations (RMI), which are based on a notion of distributed objects. RPCs and RMI are discussed in separate sections.
In many distributed applications, communication does not follow the rather strict pattern of client-server interaction. In those cases, it turns out that thinking in terms of messages is more appropriate. However, the low-level communication facilities of computer networks are in many ways not suitable due to their lack of distribution transparency. An alternative is to use a high-level message-queuing model, in which communication proceeds very much the same as in electronic mail systems. Message-oriented middleware (MOM) is a subject important enough to warrant a section of its own.

With the advent of multimedia distributed systems, it became apparent that many systems were lacking support for communication of continuous media, such as audio and video. What is needed is the notion of a stream that can support the continuous flow of messages, subject to various timing constraints. Streams are discussed in the last section of this chapter.

2.1 LAYERED PROTOCOLS

Due to the absence of shared memory, all communication in distributed systems is based on exchanging (low level) messages. When process A wants to communicate with process B, it first builds a message in its own address space. Then it executes a system call that causes the operating system to send the message over the network to B. Although this basic idea sounds simple enough, in order to prevent chaos, A and B have to agree on the meaning of the bits being sent. If A sends a brilliant new novel written in French and encoded in IBM’s EBCDIC character code, and B expects the inventory of a supermarket written in English and encoded in ASCII, communication will be less than optimal.

Many different agreements are needed. How many volts should be used to signal a 0-bit, and how many volts for a 1-bit? How does the receiver know which is the last bit of the message? How can it detect if a message has been damaged or lost, and what should it do if it finds out? How long are numbers, strings, and other data items, and how are they represented? In short, agreements are needed at a variety of levels, varying from the low-level details of bit transmission to the high-level details of how information is to be expressed.

To make it easier to deal with the numerous levels and issues involved in communication, the International Standards Organization (ISO) developed a reference model that clearly identifies the various levels involved, gives them standard names, and points out which level should do which job. This model is called the Open Systems Interconnection Reference Model (Day and Zimmerman, 1985), usually abbreviated as ISO OSI or sometimes just the OSI model. It should be emphasized that the protocols that were developed as part of the OSI model were never widely used. However, the underlying model itself has proved to be quite useful for understanding computer networks. Although we do not intend to give a full description of this model and all of its implications here, a short introduction will be helpful. For more details, see (Tanenbaum, 1996).

The OSI model is designed to allow open systems to communicate. An open system is one that is prepared to communicate with any other open system by using standard rules that govern the format, contents, and meaning of the messages sent and received. These rules are formalized in what are called protocols. To allow a group of computers to communicate over a network, they must all agree on the protocols to be used. A distinction is made between two general types of protocols. With connection-oriented protocols, before exchanging data the sender and receiver first explicitly establish a connection, and possibly negotiate the protocol they will use. When they are done, they must release (terminate) the connection. The telephone is a connection-oriented communication system. With connectionless protocols, no setup in advance is needed. The sender just transmits the first message when it is ready. Dropping a letter in a mailbox is an example of connectionless communication. With computers, both connection-oriented and connectionless communication are common.

In the OSI model, communication is divided up into seven layers or layers, as shown in Fig. 2-1. Each layer deals with one specific aspect of the communication. In this way, the problem can be divided up into manageable pieces, each of which can be solved independently of the others. Each layer provides an interface to the one above it. The interface consists of a set of operations that together define the service the layer is prepared to offer its users.

When process A on machine 1 wants to communicate with process B on machine 2, it builds a message and passes the message to the application layer on its machine. This layer might be a library procedure, for example, but it could also
be implemented in some other way (e.g., inside the operating system, on an external network processor, etc.). The application layer software then adds a header to the front of the message and passes the resulting message across the layer 6/7 interface to the presentation layer. The presentation layer in turn adds its own header and passes the result down to the session layer, and so on. Some layers add not only a header to the front, but also a trailer to the end. When it hits the bottom, the physical layer actually transmits the message, which by now might look as shown in Fig. 2-2.

![Diagram of message structure](image)

Figure 2-2. A typical message as it appears on the network.

When the message arrives at machine 2, it is passed upward, with each layer stripping off and examining its own header. Finally, the message arrives at the receiver, process B, which may reply to it using the reverse path. The information in the layer n header is used for the layer n protocol.

As an example of why layered protocols are important, consider communication between two companies, Zippy Airlines and its caterer, Mushy Meals, Inc. Every month, the head of passenger service at Zippy asks her secretary to contact the sales manager's secretary at Mushy to order 100,000 boxes of rubber chicken. Traditionally, the orders have gone via the post office. However, as the postal service deteriorates, at some point the two secretaries decide to abandon it and communicate by FAX. They can do this without bothering their bosses, since their protocol deals with the physical transmission of the orders, not their contents.

Similarly, the head of passenger service can decide to drop the rubber chicken and go for Mushy's new special, prime rib of goat, without that decision affecting the secretaries. The thing to notice is that we have two layers here, the bosses and the secretaries. Each layer has its own protocol (subjects of discussion and technology) that can be changed independently of the other one. It is precisely this independence that makes layered protocols attractive. Each one can be changed as technology improves, without the other ones being affected.

In the OSI model, there are not two layers, but seven, as we saw in Fig. 2-1. The collection of protocols used in a particular system is called a protocol suite or protocol stack. It is important to distinguish a reference model from its actual protocols. As we mentioned, the OSI protocols were never popular. In contrast, protocols developed for the Internet, such as TCP and IP, are mostly used. In the following sections, we will briefly examine each of the OSI layers in turn, starting at the bottom. However, instead of giving examples of OSI protocols, where appropriate, we will point out some of the Internet protocols used in each layer.

### 2.1.1 Lower-Level Protocols

We start with discussing the three lowest layers of the OSI protocol suite. Together, these layers implement the basic functions that encompass a computer network.

#### Physical Layer

The physical layer is concerned with transmitting the 0s and 1s. How many volts to use for 0 and 1, how many bits per second can be sent, and whether transmission can take place in both directions simultaneously are key issues in the physical layer. In addition, the size and shape of the network connector (plug), as well as the number of pins and meaning of each are of concern here.

The physical layer protocol deals with standardizing the electrical, mechanical, and signaling interfaces so that when one machine sends a 0 bit it is actually received as a 0 bit and not a 1 bit. Many physical layer standards have been developed (for different media), for example, the RS-232-C standard for serial communication lines.

#### Data Link Layer

The physical layer just sends bits. As long as no errors occur, all is well. However, real communication networks are subject to errors, so some mechanism is needed to detect and correct them. This mechanism is the main task of the data link layer. What it does is to group the bits into units, sometimes called frames, and see that each frame is correctly received.

The data link layer does its work by putting a special bit pattern on the start and end of each frame to mark them, as well as computing a checksum by adding up all the bytes in the frame in a certain way. The data link layer appends the checksum to the frame. When the frame arrives, the receiver recomputes the checksum from the data and compares the result to the checksum following the frame. If they agree, the frame is considered correct and is accepted. If they disagree, the receiver asks the sender to retransmit it. Frames are assigned sequence numbers (in the header), so everyone can tell which is which.

In Fig. 2-3 we see a (slightly pathological) example of A trying to send two messages, 0 and 1, to B. At time 0, data message 0 is sent, but when it arrives, at time 1, noise on the transmission line has damaged so that the checksum is wrong.
B notices this, and at time 2 asks for a retransmission using a control message. Unfortunately, at the same time, A is sending data message 1. When B gets the request for retransmission, it resends 0. However, when B gets message 1, instead of the requested message 0, it sends control message 1 to A complaining that it wants 0, not 1. When A sees this, it shrugs its shoulders and sends message 0 for the third time.

<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>B</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>Data 0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>Data 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
<td>B gets 0, sees bad checksum</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>Control 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>A sends data message 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
<td>B complains about the checksum</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>Data 1, Control 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>Both messages arrive correctly</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
<td>B says: “I want 0, not 1”</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>Data 0, Control 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>Both messages arrive correctly</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
<td>B says: “I already retransmitted it”</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>Data 0, Control 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>Both messages arrive correctly</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
<td>B says: “No you didn’t”</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>Data 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>A retransmits data message 0 again</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>Data 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
<td>B finally gets message 0</td>
</tr>
</tbody>
</table>

**Figure 2.3.** Discussion between a receiver and a sender in the data link layer.

The point here is not so much whether the protocol of Fig. 2-3 is a great one (it is not), but rather to illustrate that in each layer there is a need for discussion between the sender and the receiver. Typical messages are: “Please retransmit message n,” “I already retransmitted it,” “No you didn’t,” “Yes I did,” “All right, have it your way, but send it again,” and so forth. This discussion takes place in the header field, where various requests and responses are defined, and parameters (such as frame numbers) can be supplied.

**Network Layer**

On a LAN, there is usually no need for the sender to locate the receiver. It just puts the message out on the network and the receiver takes it off. A wide-area network, however, consists of a large number of machines, each with some number of lines to other machines, rather like a large-scale map showing major cities and roads connecting them. For a message to get from the sender to the receiver it may have to make a number of hops, at each one choosing an outgoing line to use.

**The Question of How to Choose the Best Path is Called Routing, and Is Essentially the Primary Task of the Network Layer**

The problem is complicated by the fact that the shortest route is not always the best route. What really matters is the amount of delay on a given route, which, in turn, is related to the amount of traffic and the number of messages queued up for transmission over the various lines. The delay can thus change over the course of time. Some routing algorithms try to adapt to changing loads, whereas others are content to make decisions based on long-term averages.

At present, perhaps the most widely used network protocol is the connectionless IP (Internet Protocol), which is part of the Internet protocol suite. An IP packet (the technical term for a message in the network layer) can be sent without any setup. Each IP packet is routed to its destination independent of all others. No internal path is selected and remembered.

A connection-oriented protocol that is now gaining popularity, is the **virtual channel** in ATM networks. A virtual channel in ATM is a unidirectional connection from a source to a destination, possibly crossing several intermediate ATM switches. Instead of setting up each virtual channel separately between two hosts, a collection of virtual channels can be grouped into what is called a **virtual path**. A virtual path is comparable to a predefined route between two hosts, along which all its virtual channels are laid down. Rerouting a path implies that all the associated channels are automatically rerouted as well. More on ATM can be found in (Handel et al., 1994).

### 2.1.2 Transport Protocols

The transport layer forms the last part of what could be called a basic network protocol stack, in the sense that it implements all those services that are not provided at the interface of the network layer, but which are reasonably needed to build network applications. In other words, the transport layer turns the underlying network into something that an application developer can use.

**The Function of the Transport Layer**

Packets can be lost on the way from the sender to the receiver. Although some applications can handle their own error recovery, others prefer a reliable connection. The job of the transport layer is to provide this service. The idea is that the application layer should be able to deliver a message to the transport layer with the expectation that it will be delivered without loss.

Upon receiving a message from the application layer, the transport layer breaks it into pieces small enough for transmission, assigns each one a sequence number, and then sends them all. The discussion in the transport layer header concerns which packets have been sent, which have been received, how many more the receiver has room to accept, which should be retransmitted, and similar topics.
Reliable transport connections (which by definition are connection-oriented) can be built on top of connection-oriented or connectionless network services. In the former case all the packets will arrive in the correct sequence (if they arrive at all), but in the latter case it is possible for one packet to take a different route and arrive earlier than the packet sent before it. It is up to the transport layer software to put everything back in order to maintain the illusion that a transport connection is like a big tube—you put messages into it and they come out undamaged and in the same order in which they went in. Providing this end-to-end communication behavior is an important aspect of the transport layer.

The Internet transport protocol is called TCP (Transmission Control Protocol) and is described in detail in (Comer, 2000a). The combination TCP/IP is now used as a de facto standard for network communication. The Internet protocol suite also supports a connectionless transport protocol called UDP (Universal Datagram Protocol), which is essentially just IP with some minor additions. User programs that do not need a connection-oriented protocol normally use UDP.

The official ISO transport protocol has five variants, known as TP0 through TP4. The differences relate to error handling and the ability to send several transport connections over a single lower-level connection (specifically, X.25). The choice of which one to use depends on the properties of the underlying network layer. None of these were ever used much.

Additional transport protocols are regularly proposed. For example, to support real-time data transfer, the Real-time Transport Protocol (RTP) has been defined. RTP is a framework protocol in the sense that it specifies packet formats for real-time data without providing the actual mechanisms for guaranteeing data delivery. In addition, it specifies a protocol for monitoring and controlling data transfer of RTP packets (Schulzrinne et al., 1996).

**Client-Server TCP**

Client-server interaction in distributed systems is often done using the transport protocols of the underlying network. With the increasing popularity of the Internet, it is now common to build client-server applications and systems using TCP. The benefit of TCP compared to UDP is that it works reliably over any network. The obvious drawback is that TCP introduces considerably more overhead, especially compared to those cases in which the underlying network is highly reliable, such as in local-area systems.

When performance and reliability are at stake, an alternative solution has always been to resort to UDP, and combine it with additional error and flow control that is optimized for the specific application. The drawback of this approach is that much extra development work needs to be done, but also that a proprietary solution is introduced, which affects the openness of the system.

What makes TCP so unattractive in many cases, is that it is not tailored to support the synchronous request-reply behavior of most client-server interaction.

Under normal circumstances, when messages do not get lost, using TCP for client-server interaction proceeds as shown in Fig. 2-4(a). First, the client initiates the setup of a connection, which is performed using a three-way handshake protocol, shown as the first three messages in Fig. 2-4(a). This protocol is needed for the two sides to reach agreement on sequence numbering for packets that are to be sent across the connection (see Tanenbaum, 1996 for further details). When the connection has been set up, the client sends its request (message 4), directly followed by a packet telling the server to close the connection (message 5).

![Figure 2-4](image)

(a) Normal operation of TCP.
(b) Transactional TCP.

The server responds by immediately acknowledging that it received the client's request, piggybacked with an acknowledgement that the connection will be closed down (message 6). The server then does the requested work and sends the answer to the client (message 7), followed by a request to release the connection as well (message 8). The client need only respond with an acknowledgement to finish its communication with the server (message 9).

Clearly, much of the overhead in TCP comes from actually managing the connection. When TCP is used for client-server interaction, it is much cheaper to combine setting up a connection with immediately sending the request, and likewise to combine sending an answer with closing the connection. The resulting protocol is called TCP for Transactions, abbreviated to T/CP, and the essence of how it operates under normal conditions is shown in Fig. 2-4(b).
What happens under normal circumstances, is that a client sends a single message (shown as message 1) containing three pieces of information: a request to set up a connection, the actual service request, and a request telling the server that it can immediately tear down the connection afterward.

The server responds only after it has serviced the actual request, so that it can send the answer along with the necessary data for accepting the connection, and immediately requesting its release, shown as message 2 in Fig. 2-4(b). Again, the client need only acknowledge the final release of the connection (message 3).

The protocol has been designed as an enhancement to TCP, meaning that a T/TCP process will automatically switch to normal TCP when the other side does not have a T/TCP implementation. Stevens (1996) discusses TCP/IP at length.

### 2.1.3 Higher-Level Protocols

Above the transport layer, OSI distinguished three additional layers. In practice, only the application layer is ever used. In fact, in the Internet protocol suite, everything above the transport layer is grouped together. In the face of middleware systems, we shall see in this section that neither the OSI nor the Internet approach is really appropriate.

#### Session and Presentation Protocols

The session layer is essentially an enhanced version of the transport layer. It provides dialog control, to keep track of which party is currently talking, and it provides synchronization facilities. The latter are useful to allow users to insert checkpoints into long transfers, so that in the event of a crash, it is necessary to go back only to the last checkpoint, rather than all the way back to the beginning. In practice, few applications are interested in the session layer and it is rarely supported. It is not even present in the Internet protocol suite.

Unlike the lower layers, which are concerned with getting the bits from the sender to the receiver reliably and efficiently, the presentation layer is concerned with the meaning of the bits. Most messages do not consist of random bit strings, but more structured information such as people’s names, addresses, amounts of money, and so on. In the presentation layer it is possible to define records containing fields like these and then have the sender notify the receiver that a message contains a particular record in a certain format. This makes it easier for machines with different internal representations to communicate.

#### Application Protocols

The OSI application layer was originally intended to contain a collection of standard network applications such as those for electronic mail, file transfer, and terminal emulation. By now, it has become the container for all applications and protocols that in one way or the other do not fit into one of the underlying layers. From the perspective of the OSI reference model, virtually all distributed systems are just applications.

What is missing in this model is a clear distinction between applications, application-specific protocols, and general-purpose protocols. For example, the Internet File Transfer Protocol (FTP) (Postel and Reynolds, 1985; Horowitz and Lunt, 1997) defines a protocol for transferring files between a client and server machine. The protocol should not be confused with the ftp program, which is an end-user application for transferring files and which also (not entirely by coincidence) happens to implement the Internet FTP.

Another example of a typical application-specific protocol is the HyperText Transfer Protocol (HTTP) (Fielding et al., 1999), which is designed to remotely manage and handle the transfer of Web pages. The protocol is implemented by applications such as Web browsers and Web servers. However, HTTP is now also used by systems that are not intrinsically tied to the Web. For example, Java’s RMI uses HTTP to request the invocation of remote objects that are protected by a firewall (Sun Microsystems, 1998).

There are also many general-purpose protocols that are useful to many applications, but which cannot be qualified as transport protocols. In many cases, such protocols fall into the category of middleware protocols, which we discuss next.

#### Middleware Protocols

Middleware is an application that logically lives in the application layer, but which contains many general-purpose protocols that warrant their own layers, independent of other, more specific applications. A distinction can be made between high-level communication protocols and protocols for establishing various middleware services.

There are numerous protocols to support a variety of middleware services. For example, as we discuss in Chap. 8, there are various ways to establish authentication, that is, provide proof of a claimed identity. Authentication protocols are not closely tied to any specific application, but instead, can be integrated into a middleware system as a general service. Likewise, authorization protocols by which authenticated users and processes are granted access only to those resources for which they have authorization, tend to have a general, application-independent nature.

As another example, we shall consider a number of distributed commit protocols in Chap. 7. Commit protocols establish that in a group of processes either all processes carry out a particular operation, or that the operation is not carried out at all. This phenomenon is also referred to as atomicity and is widely applied in transactions. As we shall see, besides transactions, other applications, like fault-tolerant ones, can also take advantage of distributed commit protocols.
As a last example, consider a distributed locking protocol by which a resource can be protected against simultaneous access by a collection of processes that are distributed across multiple machines. We shall come across a number of such protocols in Chap. 5. Again, this is an example of a protocol that can be used to implement a general middleware service, but which, at the same time, is highly independent of any specific application.

Middleware communication protocols support high-level communication services. For example, in the next two sections we shall discuss protocols that allow a process to call a procedure or invoke an object on a remote machine in a highly transparent way. Likewise, there are high-level communication services for setting and synchronizing streams for transferring real-time data, such as needed for multimedia applications. As a last example, some middleware systems offer reliable multicast services that scale to thousands of receivers spread across a wide-area network.

Some of the middleware communication protocols could equally well belong in the transport layer, but there may be specific reasons to keep them at a higher level. For example, reliable multicasting services that guarantee scalability can be implemented only if application requirements are taken into account. Consequently, a middleware system may offer different (tunable) protocols, each in turn implemented using different transport protocols, but perhaps offering a single interface.

Taking this approach to layering leads to a slightly adapted reference model for communication, as shown in Fig. 2-5. Compared to the OSI model, the session and presentation layer have been replaced by a single middleware layer that contains application-independent protocols. These protocols do not belong in the lower layers we just discussed. The original transport services may also be offered as a middleware service, without being modified. This approach is analogous to offering UDP at the transport level. Likewise, middleware communication services may include message-passing services comparable to those offered by the transport layer.

In the remainder of this chapter, we concentrate on four high-level middleware communication services: remote procedure calls, remote object invocation, message queuing services, and support for communication of continuous media through streams.

### 2.2 Remote Procedure Call

Many distributed systems have been based on explicit message exchange between processes. However, the procedures send and receive do not conceal communication, which is important to achieve transparency in distributed systems. This problem has long been known, but little was done about it until a paper by Birrell and Nelson (1984) introduced a completely different way of handling communication. Although the idea is refreshingly simple (once someone has thought of it), the implications are often subtle. In this section we will examine the concept, its implementation, its strengths, and its weaknesses.

In a nutshell, what Birrell and Nelson suggested was allowing programs to call procedures located on other machines. When a process on machine A calls a procedure on machine B, the calling process on A is suspended, and execution of the called procedure takes place on B. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing at all is visible to the programmer. This method is known as **Remote Procedure Call**, or often just **RPC**.

While the basic idea sounds simple and elegant, subtle problems exist. To start with, because the calling and called procedures run on different machines, they execute in different address spaces, which causes complications. Parameters and results also have to be passed, which can be complicated, especially if the machines are not identical. Finally, both machines can crash and each of the possible failures causes different problems. Still, most of these can be dealt with, and RPC is a widely-used technique that underlies many distributed systems.

#### 2.2.1 Basic RPC Operation

We first start with discussing conventional procedure calls, and then explain how the call itself can be split into a client and server part that are each executed on different machines.

**Conventional Procedure Call**

To understand how RPC works, it is important first to fully understand how a conventional (i.e., single machine) procedure call works. Consider a call in C like:

```c
count = read(fd, buf, nbytes);
```
where `fd` is an integer indicating a file, `buf` is an array of characters into which data are read, and `nbytes` is another integer telling how many bytes to read. If the call is made from the main program, the stack will be as shown in Fig. 2-6(a) before the call. To make the call, the caller pushes the parameters onto the stack in order, last one first, as shown in Fig. 2-6(b). (The reason that C compilers push the parameters in reverse order has to do with `printf`—by doing so, `printf` can always locate its first parameter, the format string.) After read has finished running, it puts the return value in a register, removes the return address, and transfers control back to the caller. The caller then removes the parameters from the stack, returning it to the original state.

```
Figure 2-6. (a) Parameter passing in a local procedure call: the stack before the
call to read. (b) The stack while the called procedure is active.

Several things are worth noting. For one, in C, parameters can be call-by-
value or call-by-reference. A value parameter, such as `fd` or `nbytes`, is simply
copied to the stack as shown in Fig. 2-6(b). To the called procedure, a value par-
амeter is just an initialized local variable. The called procedure may modify it,
but such changes do not affect the original value at the calling side.

A reference parameter in C is a pointer to a variable (i.e., the address of the
variable), rather than the value of the variable. In the call to read, the second par-

ermeter is a reference parameter because arrays are always passed by reference in
C. What is actually pushed onto the stack is the address of the character array. If
the called procedure uses this parameter to store something into the character
array, it does modify the array in the calling procedure. The difference between
call-by-value and call-by-reference is quite important for RPC, as we shall see.

One other parameter passing mechanism also exists, although it is not used in
C. It is called call-by-copy/restore. It consists of having the variable copied to
the stack by the caller, as in call-by-value, and then copied back after the call,
overwriting the caller’s original value. Under most conditions, this achieves ex-
actly the same effect as call-by-reference, but in some situations, such as the

same parameter being present multiple times in the parameter list, the semantics
are different. The call-by-copy/restore mechanism is not used in many languages.

The decision of which parameter passing mechanism to use is normally made
by the language designers and is a fixed property of the language. Sometimes it
depends on the data type being passed. In C, for example, integers and other
scalar types are always passed by value, whereas arrays are always passed by
reference, as we have seen. Some Ada compilers use copy/restore for in parameters,
but others use call-by-reference. The language definition permits either
choice, which makes the semantics a bit fuzzy.

**Client and Server Stubs**

The idea behind RPC is to make a remote procedure call look as much as pos-
sible like a local one. In other words, we want RPC to be transparent—the calling
procedure should not be aware that the called procedure is executing on a dif-
ferent machine or vice versa. Suppose that a program needs to read some data
from a file. The programmer puts a call to read in the code to get the data. In a
traditional (single-processor) system, the read routine is extracted from the library
by the linker and inserted into the object program. It is a short procedure, which is
generally implemented by calling an equivalent read system call. In other words,
the read procedure is a kind of interface between the user code and the local
operating system.

Even though read does a system call, it is called in the usual way, by pushing
the parameters onto the stack, as shown in Fig. 2-6(b). Thus the programmer does
not know that read is actually doing something fishy.

RPC achieves its transparency in an analogous way. When read is actually a
remote procedure (e.g., one that will run on the file server’s machine), a different
version of read, called a client stub, is put into the library. Like the original one,
it too is called using the calling sequence of Fig. 2-6(b). Also like the original
one, it too, does a call to the local operating system. Unlike the original one,
it does not ask the operating system to give it data. Instead, it packs the param-
eters into a message and requests that message to be sent to the server as illus-
trated in Fig. 2-7. Following the call to send, the client stub calls receive, block-
ing itself until the reply comes back.

When the message arrives at the server, the server’s operating system passes
it up to a server stub. A server stub is the server-side equivalent of a client stub:
it is a piece of code that transforms requests coming in over the network into local
procedure calls. Typically the server stub will have called receive and be blocked
waiting for incoming messages. The server stub unpacks the parameters from the
message and then calls the server procedure in the usual way (i.e., as in Fig. 2-6).
result to the caller in the usual way. For example, in the case of read, the server will fill the buffer, pointed to by the second parameter, with the data. This buffer will be internal to the server stub.

When the server stub gets control back after the call has completed, it packs the result (the buffer) in a message and calls send to return it to the client. After that, the server stub usually does a call to receive again, to wait for the next incoming request.

When the message gets back to the client machine, the client’s operating system sees that it is addressed to the client process (or actually the client stub, but the operating system cannot see the difference). The message is copied to the receiving buffer and the client process unblocks. The client stub inspects the message, unpacks the result, copies it to its caller, and returns in the usual way. When the caller gets control following the call to read, all it knows is that its data are available. It has no idea that the work was done remotely instead of by the local operating system.

This blissful ignorance on the part of the client is the beauty of the whole scheme. As far as it is concerned, remote services are accessed by making ordinary (i.e., local) procedure calls, not by calling send and receive. All the details of the message passing are hidden away in the two library procedures, just as the details of actually making system calls are hidden away in traditional libraries.

To summarize, a remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client’s OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.

SEC. 2.2  REMOTE PROCEDURE CALL

8. The server’s OS sends the message to the client’s OS.
9. The client’s OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

The net effect of all these steps is to convert the local call by the client procedure to the client stub, to a local call to the server procedure without either client or server being aware of the intermediate steps.

2.2.2 Parameter Passing

The function of the client stub is to take its parameters, pack them into a message, and send them to the server stub. While this sounds straightforward, it is not quite as simple as it at first appears. In this section we will look at some of the issues concerned with parameter passing in RPC systems.

Passing Value Parameters

Packaging parameters into a message is called parameter marshaling. As a very simple example, consider a remote procedure, add(i, j), that takes two integer parameters i and j and returns their arithmetic sum as a result. (As a practical matter, one would not normally make such a simple procedure remote due to the overhead, but as an example it will do.) The call to add, is shown in the left-hand portion (in the client process) in Fig. 2-8. The client stub takes its two parameters and puts them in a message as indicated. It also puts the name or number of the procedure to be called in the message because the server might support several different calls, and it has to be told which one is required.

When the message arrives at the server, the stub examines the message to see which procedure is needed and then makes the appropriate call. If the server also supports other remote procedures, the server stub might have a switch statement in it to select the procedure to be called, depending on the first field of the message. The actual call from the stub to the server looks much like the original client call, except that the parameters are variables initialized from the incoming message.

When the server has finished, the server stub gains control again. It takes the result provided by the server and packs it into a message. This message is sent back to the client stub, which unpacks it and returns the value to the client procedure.

As long as the client and server machines are identical and all the parameters and results are scalar types, such as integers, characters, and Booleans, this model works fine. However, in a large distributed system, it is common that multiple machine types are present. Each machine often has its own representation for numbers, characters, and other data items. For example, IBM mainframes use the
EBCDIC character code, whereas IBM personal computers use ASCII. As a consequence, it is not possible to pass a character parameter from an IBM PC client to an IBM mainframe server using the simple scheme of Fig. 2-8; the server will interpret the character incorrectly.

Similar problems can occur with the representation of integers (one's complement versus two's complement) and floating-point numbers. In addition, an even more annoying problem exists because some machines, such as the Intel Pentium, number their bytes from right to left, whereas others, such as the Sun SPARC, number them the other way. The Intel format is called *little endian* and the SPARC format is called *big endian*, after the politicians in *Gulliver's Travels* who went to war over which end of an egg to break (Cohen, 1981). As an example, consider a procedure with two parameters, an integer and a four-character string. Each parameter requires one 32-bit word. Fig. 2-9(a) shows what the parameter portion of a message built by a client stub on an Intel Pentium might look like. The first word contains the integer parameter, 5 in this case, and the second contains the string “JILL.”

Since messages are transferred byte for byte (actually, bit for bit) over the network, the first byte sent is the first byte to arrive. In Fig. 2-9(b) we show what the message of Fig. 2-9(a) would look like if received by a SPARC, which numbers its bytes with byte 0 at the left (high-order byte) instead of at the right (low-order byte) as do all the Intel chips. When the server stub reads the parameters at addresses 0 and 4, respectively, it will find an integer equal to 5, 83, 86, 080 (5 × 212) and a string “JILL.”

One obvious, but unfortunately incorrect, approach is to simply invert the bytes of each word after they are received, leading to Fig. 2-9(c). Now the integer is 5 and the string is “LILL.” The problem here is that integers are reversed by the different byte ordering, but strings are not. Without additional information about what is a string and what is an integer, there is no way to repair the damage.

**Passing Reference Parameters**

We now come to a difficult problem: How are pointers, or in general, references passed? The answer is: only with the greatest of difficulty, if at all. Remember that a pointer is meaningful only within the address space of the process in which it is being used. Getting back to our read example discussed earlier, if the second parameter (the address of the buffer) happens to be 1000 on the client, one cannot just pass the number 1000 to the server and expect it to work. Address 1000 on the server might be in the middle of the program text.

One solution is just to forbid pointers and reference parameters in general. However, these are so important that this solution is highly undesirable. In fact, it is not necessary either. In the read example, the client stub knows that the second parameter points to an array of characters. Suppose, for the moment, that it also knows how big the array is. One strategy then becomes apparent: copy the array into the message and send it to the server. The server stub can then call the server with a pointer to this array, even though this pointer has a different numerical value than the second parameter of read had. Changes the server makes using the pointer (e.g., storing data into it) directly affect the message buffer inside the server stub. When the server finishes, the original message can be sent back to the client stub, which then copies it back to the client. In effect, call-by-reference has been replaced by copy/restore. Although this is not always identical, it frequently is good enough.

One optimization makes this mechanism twice as efficient. If the stubs know whether the buffer is an input parameter or an output parameter to the server, one of the copies can be eliminated. If the array is input to the server (e.g., in a call to *write*) it need not be copied back. If it is output, it need not be sent over in the first place.

As a final comment, it is worth noting that although we can now handle pointers to simple arrays and structures, we still cannot handle the most general
case of a pointer to an arbitrary data structure such as a complex graph. Some systems attempt to deal with this case by actually passing the pointer to the server stub and generating special code in the server procedure for using pointers. For example, a request may be sent back to the client to provide the referenced data.

Parameter Specification and Stub Generation

From what we have explained so far, it is clear that hiding a remote procedure call requires that the caller and callee agree on the format of the messages they exchange, and that they follow the same steps when it comes to, for example, passing complex data structures. In other words, both sides in an RPC should follow the same protocol.

As a simple example, consider the procedure of Fig. 2-10(a). It has three parameters, a character, a floating-point number, and an array of five integers. Assuming a word is four bytes, the RPC protocol might prescribe that we should transmit a character in the rightmost byte of a word (leaving the next 3 bytes empty), a float as a whole word, and an array as a group of words equal to the array length, preceded by a word giving the length, as shown in Fig. 2-10(b). Thus given these rules, the client stub for fooobar knows that it must use the format of Fig. 2-10(b), and the server stub knows that incoming messages for fooobar will have the format of Fig. 2-10(b).

```
foobar\'s local variables

x
y
z[5]

{ ... }
```

Figure 2-10. (a) A procedure. (b) The corresponding message.

Defining the message format is one aspect of an RPC protocol, but it is not sufficient. What we also need is the client and the server to agree on the representation of simple data structures, such as integers, characters, Booleans, etc. For example, the protocol could prescribe that integers are represented in two's complement, characters in 16-bit Unicode, and floats in the IEEE standard #754 format, with everything stored in little endian. With this additional information, messages can be unambiguously interpreted.

With the encoding rules now pinned down to the last bit, the only thing that remains to be done is that the caller and callee agree on the actual exchange of messages. For example, it may be decided to use a connection-oriented transport service such as TCP/IP. An alternative is to use an unreliable datagram service and let the client and server implement an error control scheme as part of the RPC protocol. In practice, several variants exist.

Once the RPC protocol has been completely defined, the client and server stubs need to be implemented. Fortunately, stubs for the same protocol but different procedures generally differ only in their interface to the applications. An interface consists of a collection of procedures that can be called by a client, and which are implemented by a server. An interface is generally available in the same programming language as the one in which the client or server is written (although this is strictly speaking, not necessary). To simplify matters, interfaces are often specified by means of an Interface Definition Language (IDL). An interface specified in such an IDL, is then subsequently compiled into a client stub and a server stub, along with the appropriate compile-time or run-time interfaces.

Practice shows that using an interface definition language considerably simplifies client-server applications based on RPCs. Because it is easy to fully generate client and server stubs, all RPC-based middleware systems offer an IDL to support application development. In some cases, using the IDL is even mandatory, as we shall see in later chapters.

2.2.3 Extended RPC Models

Remote procedure calls have become a de facto standard for communication in distributed systems. The popularity of the model is due to its apparent simplicity. In this section, we take a brief look at two extensions to the original RPC model that have been designed to solve some of its shortcomings.

Doors

The original RPC model assumes that the caller and callee can communicate only by means of passing messages over a network. In general, this assumption is correct. However, suppose that the client and server reside on the same machine. Normally, we would make use of the local interprocess communication (IPC) facilities that the underlying operating system offers to processes running on the same machine. For example, in UNIX such facilities include shared memory, pipes, and message queues (see Stevens, 1999 for a detailed discussion on IPC in UNIX systems).

Local IPC facilities tend to be much more efficient than networking facilities, even if the latter are used for communication between processes on the same machine. Consequently, when performance is an issue, different interprocess communication mechanisms may need to be combined depending on whether or not the processes we are dealing with are located on the same machine.
As a compromise, a few operating systems offer an equivalent of RPCs for processes that are colocated on the same machine, called doors. A door is a generic name for a procedure in the address space of a server process that can be called by processes colocated with the server. Doors were originally designed for the Spring operating system (Mitchell et al., 1994), and are described extensively in (Hamilton and Kougiouris 1993). A similar mechanism, called Lightweight RPC, was developed by Bershad et al. (1990).

Calling doors requires support from the local operating system, as shown in Fig. 2-11. In particular, the server process must first register a door before it can be called. When registering a door, an identifier for that door is returned that can be used to later give the door a symbolic name. Registration is done by a call to door_create. A registered door can be made available to other processes by simply associating a name with the identifier returned when the door was registered. For example, in Solaris, each door has a file name, which is associated with the door. The operating system then does an upcall to the server process that registered the door. An upcall results in an invocation of the door by the server. The results of invoking the door are returned to the client process through the system call door_return.

![Diagram of communication through doors](image)

Figure 2-11. The principle of using doors as IPC mechanism.

The main benefit of doors is that they allow the use of a single mechanism, namely procedure calls, for communication in a distributed system. Unfortunately, application developers still need to be aware whether a call is done local within the current process, local to a different process on the same machine, or to a remote process.

**Asynchronous RPC**

As in conventional procedure calls, when a client calls a remote procedure, the client will block until a reply is returned. This strict request-reply behavior is unnecessary when there is no result to return, and only leads to blocking the client while it could have proceeded and have done useful work just after requesting the remote procedure to be called. Examples of where there is often no need to wait for a reply include: transferring money from one account to another, adding entries into a database, starting remote services, batch processing, and so on.

To support such situations, RPC systems may provide facilities for what are called **asynchronous RPCs**, by which a client immediately continues after issuing the RPC request. With asynchronous RPCs, the server immediately sends a reply back to the client the moment the request is received, after which it calls the requested procedure. The reply acts as an acknowledgement to the client that the server is going to process the RPC. The client will continue without further blocking as soon as it has received the server’s acknowledgement. Fig. 2-12(b) shows how the server and client interact in the case of asynchronous RPCs. For comparison, Fig. 2-12(a) shows the normal request-reply behavior.

![Diagram of asynchronous RPC interaction](image)

Figure 2-12. (a) The interaction between client and server in a traditional RPC. (b) The interaction using asynchronous RPC.

Asynchronous RPCs can also be useful when a reply will be returned but the client is not prepared to wait for it and do nothing in the meantime. For example, a client may want to prefetch the network addresses of a set of hosts that it expects to contact soon. While a naming service is collecting those addresses, the client may want to do other things. In such cases, it makes sense to organize the communication between the client and server through two asynchronous RPCs, as shown in Fig. 2-13. The client first calls the server to hand over a list of host
names that should be looked up, and continues when the server has acknowledged the receipt of that list. The second call is done by the server, who calls the client to hand over the addresses it found. Combining two asynchronous RPCs is sometimes also referred to as a deferred synchronous RPC.

![Diagram](image)

Figure 2-13. A client and server interacting through two asynchronous RPCs.

It should be noted that variants of asynchronous RPCs exist in which the client continues immediately after sending the request to the server. In other words, the client does not wait for an acknowledgement of the server’s acceptance of the request. We refer to such RPCs as one-way RPCs. The problem with this approach is that if reliability is not guaranteed, the client cannot know for sure whether its request will be processed. We return to these matters in Chap. 7.

2.2.4 Example: DCE RPC

Remote procedure calls have been widely adopted as the basis of middleware and distributed systems in general. In this section, we take a closer look at one specific RPC system: the Distributed Computing Environment (DCE), which has been developed by the Open Software Foundation (OSF) now called The Open Group. DCE RPC is not as popular as some other RPC systems, notably Sun RPC. However, DCE RPC is highly representative of other RPC systems, and its specifications have been adopted in Microsoft’s base system for distributed computing. In addition, as we shall see in a later section, DCE RPC is also illustrative for understanding the relation between RPC systems and distributed objects. We start with a brief introduction to DCE, after which we consider the principal workings of DCE RPC.

Introduction to DCE

DCE is a true middleware system in that it is designed to execute as a layer of abstraction between existing (network) operating systems and distributed applications. Initially designed for UNIX, it has now been ported to all major operating systems including VMS and Windows NT, as well as desktop operating systems. The idea is that the customer can take a collection of existing machines, add the DCE software, and then be able to run distributed applications, all without disturbing existing (nondistributed) applications. Although most of the DCE package runs in user space, in some configurations a piece (part of the distributed file system) must be added to the kernel. The Open Group itself only sells source code, which vendors integrate into their systems.

The programming model underlying all of DCE is the client-server model, which was extensively discussed in the previous chapter. User processes act as clients to access remote services provided by server processes. Some of these services are part of DCE itself, but others belong to the applications and are written by the applications programmers. All communication between clients and servers takes place by means of RPCs.

There are a number of services that form part of DCE itself. The distributed file service is a worldwide file system that provides a transparent way of accessing any file in the system in the same way. It can either be built on top of the hosts’ native file systems or be used instead of them. The directory service is used to keep track of the location of all resources in the system. These resources include machines, printers, servers, data, and much more, and they may be distributed geographically over the entire world. The directory service allows a process to ask for a resource and not have to be concerned about where it is, unless the process cares. The security service allows resources of all kinds to be protected, so access can be restricted to authorized persons. Finally, the distributed time service is a service that attempts to keep clocks on the different machines globally synchronized. As we shall see in later chapters, having some notion of global time makes it much easier to ensure consistency in a distributed system.

Goals of DCE RPC

The goals of the DCE RPC system are relatively traditional. First and foremost, the RPC system makes it possible for a client to access a remote service by simply calling a local procedure. This interface makes it possible for client (i.e., application) programs to be written in a simple way, familiar to most programmers. It also makes it easy to have large volumes of existing code run in a distributed environment with few, if any, changes.

It is up to the RPC system to hide all the details from the clients, and to some extent, from the servers as well. To start with, the RPC system can automatically locate the correct server, and subsequently set up the communication between client and server software (generally called binding). It can also handle the message transport in both directions, fragmenting and reassembling them as needed (e.g., if one of the parameters is a large array). Finally, the RPC system can automatically handle data type conversions between the client and the server, even if they run on different architectures and have a different byte ordering.
COMMUNICATION

As a consequence of the RPC system’s ability to hide the details, clients and servers are highly independent of one another. A client can be written in Java and a server in C, or vice versa. A client and server can run on different hardware platforms and use different operating systems. A variety of network protocols and data representations are also supported, all without any intervention from the client or server.

Writing a Client and a Server

The DCE RPC system consists of a number of components, including languages, libraries, daemons, and utility programs, among others. Together these make it possible to write clients and servers. In this section we will describe the pieces and how they fit together. The entire process of writing and using an RPC client and server is summarized in Fig. 2-14.

![Diagram of the RPC compilation process]

In a client-server system, the glue that holds everything together is the interface definition, as specified in the Interface Definition Language, or IDL. It permits procedure declarations in a form closely resembling function prototypes in ANSI C. IDL files can also contain type definitions, constant declarations, and other information needed to correctly marshal parameters and unmarshal results. Ideally, the interface definition should also contain a formal definition of what the procedures do, but such a definition is beyond the current state of the art, so the interface definition just defines the syntax of the calls, not their semantics. At best the writer can add a few comments describing what the procedures do.

A crucial element in every IDL file is a globally unique identifier for the specified interface. The client sends this identifier in the first RPC message and the server verifies that it is correct. In this way, if a client inadvertently tries to bind to the wrong server, or even to an older version of the right server, the server will detect the error and the binding will not take place.

Interface definitions and unique identifiers are closely related in DCE. As illustrated in Fig. 2-14, the first step in writing a client/server application is usually calling the uuidgen program, asking it to generate a prototype IDL file containing an interface identifier guaranteed never to be used again in any interface generated anywhere by uuidgen. Uniqueness is ensured by encoding in it the location and time of creation. It consists of a 128-bit binary number represented in the IDL file as an ASCII string in hexadecimal.

The next step is editing the IDL file, filling in the names of the remote procedures and their parameters. It is worth noting that RPC is not totally transparent—for example, the client and server cannot share global variables—but the IDL rules make it impossible to express constructs that are not supported.

When the IDL file is complete, the IDL compiler is called to process it. The output of the IDL compiler consists of three files:

1. A header file (e.g., interface.h, in C terms).
2. The client stub.
3. The server stub.

The header file contains the unique identifier, type definitions, constant definitions, and function prototypes. It should be included (using #include) in both the client and server code. The client stub contains the actual procedures that the client program will call. These procedures are responsible for collecting and packing the parameters into the outgoing message and then calling the runtime system to send it. The client stub also handles unpacking the reply and returning values to the client. The server stub contains the procedures called by the runtime system on the server machine when an incoming message arrives. These, in turn, call the actual server procedures that do the work.

The next step is for the application writer to write the client and server code. Both of these are then compiled, as are the two stub procedures. The resulting client code and client stub object files are then linked with the runtime library to produce the executable binary for the client. Similarly, the server code and server stub are compiled and linked to produce the server’s binary. At runtime, the client and server are started so that the application is actually executed as well.
Binding a Client to a Server

To allow a client to call a server, it is necessary that the server be registered and prepared to accept incoming calls. Registration of a server makes it possible for a client to actually locate the server and bind to it. Server location is done in two steps:

1. Locate the server's machine.
2. Locate the server (i.e., the correct process) on that machine.

The second step is somewhat subtle. Basically, what it comes down to is that to communicate with a server, the client needs to know an endpoint, on the server's machine to which it can send messages. An endpoint (also commonly known as a port) is used by the server's operating system to distinguish incoming messages for different processes. In DCE, a table of (server, endpoint)-pairs is maintained on each server machine by a process called the DCE daemon. Before it becomes available for incoming requests, the server must ask the operating system for an endpoint. It then registers this endpoint with the DCE daemon. The DCE daemon records this information (including which protocols the server speaks) in the endpoint table for future use.

The server also registers with the directory service by providing it the network address of the server's machine and a name under which the server can be looked up. Binding a client to a server then proceeds as shown in Fig. 2-15.

![Figure 2-15. Client-to-server binding in DCE.](image)

Let us assume that the client wants to bind to a video server that is locally known under the name /local/multimedia/video/movies. It passes this name to the directory server, which returns the network address of the machine running the video server. The client then goes to the DCE daemon on that machine (which has a well-known endpoint), and asks it to look up the endpoint of the video server in its endpoint table. Armed with this information, the RPC can now take place. On subsequent RPCs this lookup is not needed. DCE also gives clients the ability to do more sophisticated searches for a suitable server when that is needed. Secure RPC is also an option.

Performing an RPC

The actual RPC is carried out transparently and in the usual way. The client sub marshals the parameters to the runtime library for transmission using the protocol chosen at binding time. When a message arrives at the server side, it is routed to the correct server based on the endpoint contained in the incoming message. The runtime library passes the message to the server stub, which unmarshals the parameters and calls the server. The reply goes back by the reverse route.

DCE provides several semantic options. The default is at-most-once operation, in which case no call is ever carried out more than once, even in the face of system crashes. In practice, what this means is that if a server crashes during an RPC and then recovers quickly, the client does not repeat the operation, for fear that it might already have been carried out once.

Alternatively, it is possible to mark a remote procedure as idempotent (in the IDL file), in which case it can be repeated multiple times without harm. For example, reading a specified block from a file can be tried over and over until it succeeds. When an idempotent RPC fails due to a server crash, the client can wait until the server reboots and then try again. Other semantics are also available (but rarely used), including broadcasting the RPC to all the machines on the local network. We return to RPC semantics in Chap. 7, when discussing RPC in the presence of failures.

2.3 REMOTE OBJECT INVOCATION

Object-based technology has proven its value for developing nondistributed applications. One of the most important aspects of an object is that it hides its internals from the outside world by means of a well-defined interface. This approach allows objects to be easily replaced or adapted, as long as the interface remains the same.

As RPC mechanisms gradually became the de facto standard for handling communication in distributed systems, people started to realize that the principle of RPCs could be equally well applied to objects. In this section, we expand the idea of RPCs to invocations on remote objects, and show how the approach can enhance distribution transparency when compared to RPCs. We concentrate only on relatively simple remote objects. In Chap. 10, we discuss a number of object-based distributed systems in detail, including CORBA and DCOM, both of which provide important and more enhanced object models than the ones discussed here.
2.3.1 Distributed Objects

The key feature of an object is that it encapsulates data, called the state, and the operations on those data, called the methods. Methods are made available through an interface. It is important to understand that there is no legal way a process can access or manipulate the state of an object other than by invoking methods made available to it through an object’s interface. An object may implement multiple interfaces. Likewise, given an interface definition, there may be several objects that offer an implementation for it.

This separation between interfaces and the objects implementing these interfaces is crucial for distributed systems. A strict separation allows us to place an interface at one machine, while the object itself resides on another machine. This organization, which is shown in Fig. 2-16, is commonly referred to as a distributed object.

Figure 2-16. Common organization of a remote object with client-side proxy.

When a client binds to a distributed object, an implementation of the object’s interface, called a proxy, is loaded into the client’s address space. A proxy is analogous to a client stub in RPC systems. The only thing it does is marshal method invocations into messages and unmarshal reply messages to return the result of the method invocation to the client. The actual object resides at a server machine, where it offers the same interface as it does on the client machine. Incoming invocation requests are first passed to a server stub, often referred to as a skeleton, which unmarshals them to proper method invocations at the object’s interface at the server. The server stub is also responsible for marshaling replies and forwarding reply messages to the client-side proxy.

A characteristic, but somewhat counterintuitive feature of most distributed objects is that their state is not distributed: it resides at a single machine. Only the interfaces implemented by the object are made available on other machines. Such objects are also referred to as remote objects. As we shall see in later chapters, in a general distributed object, the state itself may be physically distributed across multiple machines, but this distribution is also hidden from clients behind the object’s interfaces.

Compile-time versus Runtime Objects

Objects in distributed systems appear in many forms. The most obvious form is the one that is directly related to language-level objects such as those supported by Java, C++, or other object-oriented languages, which are referred to as compile-time objects. In this case, an object is defined as the instance of a class. A class is a description of an abstract type in terms of a module with data elements and operations on that data (Meyer, 1997).

Using compile-time objects in distributed systems often makes it much easier to build distributed applications. For example, in Java, an object can be fully defined by means of its class and the interfaces that the class implements. Compiling the class definition results in code that allows it to instantiate Java objects. The interfaces can be compiled into client-side and server-side stubs, allowing the Java objects to be invoked from a remote machine. A Java developer can mostly stay unaware of the distribution of objects: he sees only Java programming code.

The obvious drawback of compile-time objects is the dependency on a particular programming language. Therefore, an alternative way of constructing distributed objects is to do this explicitly during runtime. This approach is followed in many object-oriented distributed systems, as it is independent of the programming language in which distributed applications are written. In particular, an application may be constructed from objects written in multiple languages.

When dealing with runtime objects, how objects are actually implemented is basically left open. For example, a developer may choose to write a C library containing a number of functions that can all work on a common data file. The essence is how to let such an implementation appear to be an object whose methods can be invoked from a remote machine. A common approach is to use an object adapter, which acts as a wrapper around the implementation with the sole purpose to give it the appearance of an object. The term adapter is derived from a design pattern described in (Gamma et al., 1994), which allows an interface to be converted into something that a client expects. An example object adapter is one that dynamically binds to the C library mentioned above and opens an associated data file representing an object’s current state.

Object adapters play an important role in object-based distributed systems. To make wrapping as easy as possible, objects are solely defined in terms of the interfaces they implement. An implementation of an interface can then be registered at an adapter, which can subsequently make that interface available for (remote) invocations. The adapter will take care that invocation requests are
carried out, and thus provide an image of remote objects to its clients. We return to the organization of object servers and adapters in the next chapter.

**Persistent and Transient Objects**

Besides the distinction between language-level objects and runtime objects, there is also a distinction between persistent and transient objects. A **persistent object** is one that continues to exist even if it is currently not contained in the address space of a server process. In other words, a persistent object is not dependent on its current server. In practice, this means that the server that is currently managing the persistent object, can store the object's state on secondary storage and then exit. Later, a newly started server can read the object's state from storage into its own address space, and handle invocation requests. In contrast, a **transient object** is an object that exists only as long as the server that manages the object. As soon as that server exits, the object ceases to exist as well. There is much controversy about having persistent objects; some people believe that transient objects are enough. Rather than going into details now, we return to this question when discussing object-based distributed systems in Chap. 9.

**2.3.2 Binding a Client to an Object**

An interesting difference between traditional RPC systems and systems supporting distributed objects, is that the latter generally provides systemwide object references. Such object references can be freely passed between processes on different machines, for example as parameters to method invocations. By hiding the actual implementation of an object reference, that is, making it opaque, and perhaps even using it as the only way to reference objects, distribution transparency is enhanced compared to traditional RPCs.

When a process holds an object reference, it must first bind to the referenced object before invoking any of its methods. Binding results in a proxy being placed in the process's address space, implementing an interface containing the methods the process can invoke. In many cases, binding is done automatically. When the underlying system is given an object reference, it needs a way to locate the server that manages the actual object, and place a proxy in the client's address space.

With **implicit binding**, the client is offered a simple mechanism that allows it to directly invoke methods using only a reference to an object. For example, C++ allows overloading the unary member selection operator ("->") permitting us to introduce object references as if they were ordinary pointers as shown in Fig. 2-17(a). With implicit binding, the client is transparently bound to the object at the moment the reference is resolved to the actual object. In contrast, with **explicit binding**, the client should first call a special function to bind to the object before it can actually invoke its methods. Explicit binding generally returns a pointer to a proxy that is then become locally available, as shown in Fig. 2-17(b).

---

**SEC. 2.3 REMOTE OBJECT INVOCATION**

```plaintext
Distr_object* obj_ref;
obj_ref = ...;
// Declare a systemwide object reference
obj_ref->doSomething();
// Initialize the reference to a distrib. obj.
// Implicitly bind and invoke a method
(a)

Distr_object* obj_ref;
Local_object* obj_ptr;
obj_ref = ...;
obj_ptr->bind(obj_ref);
obj_ptr->doSomething();
// Declare a systemwide object reference
// Declare a pointer to local objects
// Initialize the reference to a distrib. obj.
// Explicitly bind and get ptr to local proxy
// Invoke a method on the local proxy
(b)
```

Figure 2-17. (a) An example with implicit binding using only global references.
(b) An example with explicit binding using global and local references.

**Implementation of Object References**

It is clear that an object reference must contain enough information to allow a client to bind to an object. A simple object reference would include the network address of the machine where the actual object resides, along with an endpoint identifying the server that manages the object, plus an indication of which object. The latter is generally provided by the server, for example, in the form of a 16-bit number. An endpoint in this case is completely analogous to the one discussed in the DCE RPC system. In practice, it corresponds to a local port that is dynamically assigned by the server's local operating system. However, there are a number of drawbacks to this scheme.

First, if the server's machine crashes and the server is assigned a different endpoint after recovery, all object references have become invalid. This problem can be solved as is done in DCE: have a local daemon per machine listen to a well-known endpoint and keep track of the server-to-endpoint assignments in an endpoint table. When binding a client to an object, we first ask the daemon for the server's current endpoint. This approach requires that we encode a server ID into the object reference that can be used as an index into the endpoint table. The server, in turn, is always required to register itself with the local daemon.

However, encoding the network address of the server's machine into an object reference is generally not a good idea. The problem with this approach is that the server can never move to another machine without invalidating all the references to the objects it manages. An obvious solution is to expand the idea of a local daemon, maintaining an endpoint table, to a **location server** that keeps track of the machine where an object's server is currently running. An object reference would then contain the network address of the location server, along with a systemwide identifier for the server. As we shall see in Chap. 4, this solution also has a number of serious drawbacks, especially when scalability is an issue.
What we have tacitly assumed so far is that the client and server have somehow already been configured to use the same protocol stack. Not only does this mean that they use the same transport protocol, for example, TCP, but it also means that they use the same protocol for marshaling and unmarshaling parameters. They must also use the same protocol for setting up an initial connection, handle errors and flow control the same way, and so on.

We can safely drop this assumption provided we add more information in the object reference. Such information may include the identification of the protocol that is used to bind to an object and of those that are supported by the object’s server. For example, a single server may simultaneously support data coming in over a TCP connection, as well as incoming UDP datagrams. It is then the client’s responsibility to get a proxy implementation for at least one of the protocols identified in the object reference.

We can even take this approach one step further, and include an implementation handle in the object reference, which refers to a complete implementation of a proxy that the client can dynamically load when binding to the object. For example, an implementation handle could take the form of a URL pointing to an archive file, such as \texttt{ftp://ftp.clientware.org/proxies/java/proxy-v1.1a.zip}. The binding protocol would then only need to prescribe that such a file should be dynamically downloaded, unpacked, installed, and subsequently instantiated. The benefit of this approach is that the client need not worry about whether it has an implementation of a specific protocol available. In addition, it gives the object developer the freedom to design object-specific proxies. However, as we discuss in Chap. 8, we do need to take special security measures to ensure the client that it can trust the downloaded code.

### 2.3.3 Static versus Dynamic Remote Method Invocations

After a client is bound to an object, it can invoke the object’s methods through the proxy. Such a remote method invocation or simply RMI is very similar to an RPC when it comes to issues such as marshaling and parameter passing. An essential difference between an RMI and an RPC is that RMI’s generally support systemwide object references as explained above. Also, it is not necessary to have only general-purpose client-side and server-side stubs available. Instead, we can more easily accommodate object-specific stubs as we also explained.

The usual way to provide RMI support is to specify the object’s interfaces in an interface definition language, similar to the approach followed with RPCs. Alternatively, we can make use of an object-based language such as Java, that will handle stub generation automatically. This approach of using predefined interface definitions is generally referred to as static invocation. Static invocations require that the interfaces of an object are known when the client application is being developed. It also implies that if interfaces change, then the client application must be recompiled before it can make use of the new interfaces.

As an alternative, method invocations can also be done in a more dynamic fashion. In particular, it is sometimes convenient to be able to compose a method invocation at runtime, also referred to as a dynamic invocation. The essential difference with static invocation is that an application selects at runtime which method it will invoke at a remote object. Dynamic invocation generally takes a form such as

\[ \text{invoke}(\text{object}, \text{method}, \text{input\_parameters}, \text{output\_parameters}); \]

where \text{object} identifies the distributed object, \text{method} is a parameter specifying exactly which method should be invoked, \text{input\_parameters} is a data structure that holds the values of that method’s input parameters, and \text{output\_parameters} refers to a data structure where output values can be stored.

For example, consider appending an integer \text{int} to a file \text{object} for which the object provides the method append. In this case, a static invocation would take the form

\[ \text{object.append}({\text{int}}) \]

whereas the dynamic invocation would look something like

\[ \text{invoke}(\text{object}, \text{id}(\text{append}), \text{int}) \]

where the operation \text{id(append)} returns an identifier for the method append.

To illustrate the usefulness of dynamic invocations, consider an object browser that is used to examine sets of objects. Assume that the browser supports remote object invocations. Such a browser is capable of binding to a distributed object and subsequently presenting the object’s interface to its user. The user could then be asked to choose a method and provide values for its parameters, after which the browser can do the actual invocation. Typically, such an object browser should be developed to support any possible interface. Such an approach requires that interfaces can be inspected at runtime, and that method invocations can be dynamically constructed.

Another application of dynamic invocations is a batch processing service to which invocation requests can be handed along with a time when the invocation should be done. The service can be implemented by a queue of invocation requests, ordered by the time that invocations are to be done. The main loop of the service would simply wait until the next invocation is scheduled, remove the request from the queue, and call invoke as given above.

### 2.3.4 Parameter Passing

Because most RMI systems support systemwide object references, passing parameters in method invocations is generally less restricted than in the case of RPCs. However, there are some subtleties that can make RMI’s trickier than one would initially expect, as we briefly discuss in the following pages.
Let us first consider the situation that there are only distributed objects. In other words, all objects in the system can be accessed from remote machines. In that case, we can consistently use object references as parameters in method invocations. References are passed by value, and thus copied from one machine to the other. When a process is given an object reference as the result of a method invocation, it can simply bind to the object referred to when needed later.

Unfortunately, using only distributed objects can be highly inefficient, especially when objects are small, such as integers and Booleans. Each invocation by a client that is not colocated in the same server as the object, generates a request between different address spaces or, even worse, between different machines. Therefore, references to remote objects and those to local objects are often treated differently.

When invoking a method with an object reference as parameter, that reference is copied and passed as a value parameter only when it refers to a remote object. In this case, the object is literally passed by reference. However, when the reference refers to a local object, that is an object in the same address space as the client, the referred object is copied as a whole and passed along with the invocation. In other words, the object is passed by value.

These two situations are illustrated in Fig. 2.18, which shows a client program running on machine A, and a server program on machine C. The client has a reference to a local object O1 that it uses as a parameter when calling the server program on machine C. In addition, it holds a reference to a remote object O2 residing at machine B, which is also used as a parameter. When calling the server, a copy of O1 is passed to the server on machine C, along with only a copy of the reference to O2.

![Figure 2.18](image.png)

**Figure 2.18.** The situation when passing an object by reference or by value.

Note that whether we are dealing with a reference to a local object or to a remote object can be highly transparent, such as in Java. In Java, the distinction is visible only because local objects are essentially of a different data type than remote objects. Otherwise, both types of references are treated very much the same (see also Wollrath et al., 1996). On the other hand, when using conventional programming languages such as C, a reference to a local object can be as simple as a pointer, which can never be used to refer to a remote object.

The side effect of invoking a method with an object reference as parameter is that we may be copying an object. Hiding this aspect is unacceptable, so that we are consequently forced to make an explicit distinction between local and distributed objects. Clearly, this distinction not only violates distribution transparency, but also makes it harder to write distributed applications.

### 2.3.5 Example 1: DCE Remote Objects

DCE is an example of a distributed system that may have been in the right place, but somewhat at the wrong time. Being one of the first distributed systems constructed as middleware on top of existing operating systems, DCE had to go through a relatively long period of acceptance to prove itself. Unfortunately, the acceptance phase was paralleled by the introduction of remote objects as the panacea for building distributed systems. Being a traditional RPC-based system, DCE was given a rough time as it was considered to be already out-of-date before it had established a reasonable installed base. No objects, no deal, and DCE did not have objects.

The DCE people long argued with advocates of object technology that they did support objects. For example, they claimed that RPC systems are inherently object-based because all implementation and distribution aspects are hidden behind interfaces. However, their arguments did not sell, so DCE was forced to adopt object technology more explicitly. In this section, we take a look at how DCE supports distributed objects. DCE objects are interesting as they form a direct refinement of the RPC-based client-server model, and thus forming a shift from remote procedure calls to remote method invocations.

#### The DCE Distributed-Object Model

Distributed objects have been added to DCE in the form of extensions to their Interface Definition Language (IDL), along with C++ language bindings. In other words, distributed objects in DCE are specified in IDL, and implemented in C++. Distributed objects take the form of remote objects, of which the actual implementation resides at a server. A server is responsible for creating C++ objects locally and making methods available to remote client machines. There is no other way to create distributed objects.

Two types of distributed objects are supported. A **distributed dynamic object** is an object that a server creates locally on behalf of a client, and which, in
principle, is accessible only to that client. To create an object, a client will have to
issue a request at the server. Therefore, each class of dynamic objects has an asso-
ciated create procedure that can be called using a standard RPC. After creating a
dynamic object, the DCE runtime system administrates the new object, and asso-
ciates it with the client on whose behalf it was created.

In contrast to dynamic objects, distributed named objects are not intended to
be associated with only a single client but are created by a server to have it shared
by several clients. Named objects are registered with a directory service so that a
client can look up the object and subsequently bind to it. Registration yields that a
unique identifier for that object is stored, along with information on how to con-
tact the object’s server. The difference between dynamic and named objects is
shown in Fig. 2-19.

![Diagram of dynamic and named objects in DCE]

**Figure 2-19.** (a) Distributed dynamic objects in DCE. (b) Distributed named
objects.

**DCE Remote Object Invocation**

As might be expected, each remote object invocation in DCE is done by
means of an RPC. When a client invokes a method of an object, it passes the
object identifier, the identifier of the interface that contains the method, an identi-
fication of the method itself, and parameters to the server. The server maintains an
object table from which it can derive which object is to be invoked if given the
object identifier and interface identifier. It can then properly dispatch the
requested method with its parameters.

Because a server may have thousands of objects to serve, DCE offers the pos-
sibility to place objects in secondary storage instead of keeping all objects active
in main memory. When an invocation request comes in for which no object can be
found in the server’s object table, the runtime system can alternatively invoke a
server-specific lookup function to first retrieve the object from secondary storage
and place it into the server’s address space. After the object is placed into main
memory, the invocation can take place.

Distributed objects in DCE have one problem that is inherent to their strong
RPC background: there is no mechanism for transparent object references. At
best, a client can use a binding handle associated with a named object. A binding
handle contains an identification of an interface of the object, the transport proto-
col used for communicating with the object’s server, and the server’s host address
and endpoint. A binding handle can be turned into a string and as such passed
between different processes.

Lacking a proper systemwide object reference mechanism makes parameter
passing in DCE harder than in many other object-based systems. An application
developer now has to devise a proprietary solution for passing objects in RPCs. In
practice, this means that objects need to be explicitly marshaled to be passed by
value, for which object-specific marshaling routines need to be developed.

As an alternative, a developer can use delegation by which a special stub is
generated from an object’s interface specification. The stub acts as a wrapper for
the actual object and contains only those methods that need to be called by a
remote process. The stub can then be linked into any other process that wants to
use the object. The benefit of this approach becomes clear when realizing that
DCE does allow remote references to stubs to be passed as parameters in RPCs.
Consequently, it becomes possible to refer to objects through the entire system by
means of stub references.

Further information on DCE object programming can be found in (Viveney,
1998) and (OSF, 1997).

### 2.3.6 Example 2: Java RMI

In DCE, distributed objects have essentially been added as a refinement of
remote procedure calls. Instead of identifying a remote procedure in a server, a
client identifies a remote procedure in a server’s object. The lack of a proper sys-
temwide object reference mechanism emphasizes that we are indeed dealing with
a mere refinement of RPCs.

Let us now take a look at distributed objects from a completely different per-
spective. In Java, distributed objects have been integrated into the language. An
important goal was to keep as much of the semantics of nondistributed objects as
possible. In other words, the Java language developers have aimed for a high
degree of distribution transparency. However, as we shall see, Java’s developers
have also decided to make distribution apparent where a high degree of tran-
sparency was simply too inefficient, difficult, or impossible to realize.
The Java Distributed-Object Model

Java also adopts remote objects as the only form of distributed objects. Recall that a remote object is a distributed object whose state always resides on a single machine, but whose interfaces can be made available to remote processes. Interfaces are implemented in the usual way by means of a proxy, which offers exactly the same interfaces as the remote object. A proxy itself appears as a local object in the client’s address space.

There are only a few, but subtle and important differences between remote objects and local objects. First, cloning local or remote objects is different. Cloning a local object O results in a new object of the same type as O, and with exactly the same state. Cloning thus returns an exact copy of the object that is cloned. These semantics are hard to apply to a remote object. If we were to make an exact copy of a remote object, we would not only have to clone the actual object at its server, but also the proxy at each client that is currently bound to the remote object. Cloning a remote object is therefore an operation that can be executed only by the server. It results in making an exact copy of the actual object in the server’s address space. Proxies of the actual object are thus not cloned. If a client at a remote machine wants access to the cloned object at the server, it will first have to bind to that object again.

A more important difference between local and remote objects in Java is the semantics of blocking on an object. Java allows each object to be constructed as a monitor, by declaring a method to be synchronized. If two processes simultaneously call a synchronized method, only one of the processes will proceed while the other will be blocked. In this way, we can ensure that access to an object’s internal data is completely serialized. As in monitors, a process can also be blocked inside an object, waiting for some condition to become true, as was also explained in Chap. 1.

Logically, blocking in a remote object is simple. Suppose that client A calls a synchronized method of a remote object. To make access to remote objects look always exactly the same as to local objects, it would be necessary to block A in the client-side stub that implements the object’s interface and to which A has direct access. Likewise, another client on a different machine would need to be blocked locally as well before its request can be sent to the server. The consequence is that we need to synchronize different clients at different machines. As we discuss in Chap. 5, distributed synchronization can be fairly complex.

An alternative approach would be to allow blocking only at the server. In principle, this works fine, but problems arise when a client crashes while its invocation is being handled by the server. As we shall see in Chap. 7, we may require relatively sophisticated protocols to handle this situation, and which may significantly affect the overall performance of remote method invocations.

Therefore, the designers of Java RMI have chosen to restrict blocking on remote objects only to the proxies (Woolrath et al., 1996). In practice, this means that remote objects cannot be protected against simultaneous access from processes operating on different proxies by using synchronized methods. Instead, explicit distributed locking techniques will have to be used.

Java Remote Object Invocation

As the distinction between local and remote objects is hardly visible at the language level, Java can also hide most of the differences during a remote method invocation. For example, any primitive or object type can be passed as a parameter to an RMI, provided that the type can be marshaled. In Java terminology, this means that it must be serializable. Although, in principle, most objects can be serialized, serialization is not always allowed or possible. Typically, platform-dependent objects such as file descriptors and sockets, cannot be serialized.

The only distinction made between local and remote objects during an RMI is that local objects are passed by value (including large objects such as arrays), whereas remote objects are passed by reference. In other words, a local object is first copied after which the copy is used as parameter value. For a remote object, a reference to the object is passed as parameter instead of a copy of the object, as was also shown in Fig. 2.18.

In Java RMI, a reference to a remote object is essentially implemented as we explained in Sec. 2.3.2. Such a reference consists of the network address and endpoint of the server, as well as a local identifier for the actual object in the server’s address space. That local identifier is used only by the server. As we also explained, a reference to a remote object also needs to encode the protocol stack that is used by a client and the server to communicate. To understand how such a stack is encoded in the case of Java RMI, it is important to realize that each object in Java is an instance of a class. A class, in turn, contains an implementation of one or more interfaces.

In essence, a remote object is built from two different classes. One class contains an implementation of server-side code, which we call the server class. This class contains an implementation of that part of the remote object that will be running on a server. In other words, it contains the description of the object’s state, as well as an implementation of the methods that operate on that state. The server-side stub, that is, the skeleton, is generated from the interface specifications of the object.

The other class contains an implementation of the client-side code, which we call the client class. This class contains an implementation of a proxy. Like the skeleton, this class is also generated from the object’s interface specification. In its simplest form, the only thing a proxy does is to convert each method call into a message that is sent to the server-side implementation of the remote object, and convert a reply message into the result if a method call. For each call, it sets up a connection with the server, which is subsequently torn down when the call is finished. For this purpose, the proxy needs the server’s network address and endpoint
as mentioned above. This information, along with the local identifier of the object at the server, is always stored as part of the state of a proxy.

Consequently, a proxy has all the information it needs to let a client invoke methods of the remote object. In Java, proxies are serializable. In other words, it is possible to marshal a proxy and send it as a series of bytes to another process, where it can be unmarshaled and used to invoke methods on the remote object. In other words, a proxy can be used as a reference to a remote object.

This approach is consistent with Java's way of integrating local and distributed objects. Recall that in an RMI, a local object is passed by making a copy of it, while a remote object is passed by means of a system-wide object reference. A proxy is treated as nothing else but a local object. Consequently, it is possible to pass a serializable proxy as a parameter in an RMI. The side effect is that such a proxy can be used as a reference to the remote object.

In principle, when marshaling a proxy, its complete implementation, that is, all its state and code, is converted to a series of bytes. Marshaling the code like this is not very efficient and may lead to very large references. Therefore, when marshaling a proxy in Java, what actually happens is that an implementation handle is generated, specifying precisely which classes are needed to construct the proxy. Possibly, some of these classes need to be downloaded from a remote site. The implementation handle replaces the marshaled code as part of a remote-object reference. In effect, references to remote objects in Java are in the order of a few hundred bytes.

This approach to referencing remote objects is highly flexible and is one of the distinguishing features of Java RMI (Waldo, 1998). In particular, it allows for the object-specific solutions. For example, consider a remote object whose state changes only once in a while. We can turn such an object into a truly distributed object by copying the entire state to a client at binding time. Each time the client invokes a method, it operates on the local copy. To ensure consistency, each invocation also checks whether the state at the server has changed, in which case the local copy is refreshed. Likewise, methods that modify the state are forwarded to the server. The developer of the remote object will now have to implement only the necessary client-side code, and have it dynamically downloaded when the client binds to the object.

Being able to pass proxies as parameters works only because each process is executing the same Java virtual machine. In other words, each process is running in the same execution environment. A marshaled proxy is simply unmarshaled at the receiving side, after which its code can be executed. In contrast, in DCE for example, passing stubs is out of the question, as different processes may be running in execution environments that differ with respect to language, operating system, and hardware. Instead, a DCE process first needs to (dynamically) link in a locally available stub that has been previously compiled specifically for the process's execution environment. By passing a reference to a stub as parameter in an RPC, it is possible to refer to objects across process boundaries.

2.4 MESSAGE-ORIENTED COMMUNICATION

Remote procedure calls and remote object invocations contribute to hiding communication in distributed systems, that is, they enhance access transparency. Unfortunately, neither mechanism is always appropriate. In particular, when it cannot be assumed that the receiving side is executing at the time a request is issued, alternative communication services are needed. Likewise, the inherent synchronous nature of RPCs and RMI's, by which a client is blocked until its request has been processed, sometimes needs to be replaced by something else.

That something else is messaging. In this section we concentrate on message-oriented communication in distributed systems by first taking a closer look at what exactly synchronous behavior is and what its implications are. Then, we discuss messaging systems that assume that parties are executing at the time of communication. Finally, we will examine message-queuing systems that allow processes to exchange information, even if the other party is not executing at the time communication is initiated.

2.4.1 Persistence and Synchronicity in Communication

To understand the various alternatives in message-oriented communication, we assume the communication system is organized as a computer network as shown in Fig. 2-20. Applications are always executed on hosts, where each host offers an interface to the communication system through which messages can be submitted for transmission. The hosts are connected through a network of communication servers, which are responsible for passing (and routing) messages between hosts. Without loss of generality, we may assume that each host is connected to exactly one communication server. In Chap. 1, we assumed that buffers could be placed only at hosts. In a more general setting, we need to consider that buffers can also be placed in the communication servers of underlying network.

To give an example, consider an electronic mail system based on this design. A host runs a user agent: the application by which a user can compose, send, receive, and read messages. Each host is connected to exactly one mail server, corresponding to what we have called a communication server. The interface at the user's host allows the user agent to send messages to a specific destination. When the user agent submits a message for transmission at its host, the host generally forwards the message first to its local mail server where it is temporarily stored in an output buffer.

A mail server removes a message from its output buffer and looks up the destination. Looking up the destination returns the (transport-level) address of the mail server to which the message should be sent. The mail server subsequently sets up a connection and passes the message to the target mail server. The latter stores the message in an input buffer for the designated receiver, also called the
transportation, as well as the means by which letters are sorted, have changed over the past hundred years, this principle of sorting, storing, and forwarding post is still the same.

Figure 2-21. Persistent communication of letters back in the days of the Pony Express.

In contrast, with transient communication, a message is stored by the communication system only as long as the sending and receiving application are executing. More precisely, in terms of Fig. 2-20, we have the situation that if a communication server cannot deliver a message to the next server, or the receiver, the message will simply be discarded. Typically, all transport-level communication services offer only transient communication. In this case, a communication server corresponds to a traditional store-and-forward router. If a router cannot deliver a message to the next router, or the destination host, it will simply drop the message.

Besides being persistent or transient, communication can also be asynchronous or synchronous. The characteristic feature of asynchronous communication, is that a sender continues immediately after it has submitted its message for transmission. This means that the message is stored in a local buffer at the sending host, or otherwise at the first communication server. With synchronous communication, the sender is blocked until its message is stored in a local buffer at the receiving host, or actually delivered to the receiver. The strongest form of synchronous communication is when the sender is blocked until the receiver has processed the message.

There are several combinations of these types of communication that occur in practice. In the case of persistent asynchronous communication, a message is either persistently stored in a buffer at the local host, or at the first communication server. Electronic mail systems typically offer this form of communication. With persistent synchronous communication, messages can be persistently stored only at the receiving host. A sender is blocked until its message is stored in the receiver’s buffer. Note that it is not necessary that the receiving application is
executing to store the message at its local host. A weaker form of persistent synchronous communication is when the sender is blocked until its message is stored at the communication server connected to the receiving host.

Transient asynchronous communication is typically offered by transport-level datagram services such as UDP. When a sending application submits a message for transmission, the message is temporarily stored in a local buffer at the sending host, after which the sender immediately continues. In parallel, the communication system routes the message to the destination where it expects it can deliver it to the receiver, possibly by storing it in a local buffer. If the receiver is not executing at the time the message reaches the receiver's host, transmission fails. One-way RPCs are another example of transient asynchronous communication.

Transient synchronous communication comes in different forms. In the weakest form, based on message receipt, the sender is blocked until the message is stored in a local buffer at the receiving host. The sender receives an acknowledgment and continues. In delivery-based transient synchronous communication, the sender is blocked until the message is delivered to the receiver for further processing. We came across this form of synchronous behavior when discussing asynchronous RPCs. In an asynchronous RPC, the client synchronizes with the server by waiting until its request has been accepted for further processing. The strongest form is response-based transient synchronous communication, in which the sender blocks until it receives a reply message from the other side, as in the request-reply behavior in client-server interaction. RPCs and RMIs adhere to this scheme.

These different combinations of persistence and synchronicity in communication are summarized in Fig. 2-22. Another, but similar classification, is discussed in (Tai and Rouvellou, 2000).

Until recently, many distributed systems supported only response-based transient synchronous communication, either through remote procedure calls or remote object invocations. Recognizing that this form of communication is not always appropriate, facilities for weaker forms of transient synchronous communication were added, such as asynchronous RPCs or deferred synchronous operations, as shown in Fig. 2-13.

A fundamentally different approach is followed in message-passing systems, which take transient asynchronous communication as their starting point, and possibly add facilities for synchronous communication. However, in all cases where message passing was used, communication was still assumed to be transient. In other words, only those communication facilities were provided that were suitable for simultaneously executing processes. Having only such facilities is in many cases not appropriate, especially when taking geographical scalability into account.

The need for persistent communication services became apparent when developers of middleware needed to integrate applications in large-scale and widely dispersed interconnected networks. Such networks are often distributed across different departments and administrative domains, whose parts may not always be immediately accessible. For example, access can be restricted due to network or process failures. To solve such problems, proprietary solutions for persistent communication have been developed, but such solutions are not very satisfactory in light of interoperability and portability, as can easily be imagined.

![Figure 2-22. Six different forms of communication: (a) persistent asynchronous communication, (b) persistent synchronous communication, (c) transient asyn-chronous communication, (d) request-based transient synchronous communication, (e) delivery-based transient synchronous communication at message delivery, and (f) response-based transient synchronous communication.](image-url)
Another drawback of transient communication is that whenever a failure occurs, that failure has to be immediately masked and a recovery procedure has to be initiated. It would be unacceptable to postpone a repair as this would violate failure transparency. However, with persistent communication, applications are developed to handle long delays between sending a request and receiving an answer. Consequently, we can resort to simpler, but perhaps slower solutions for failure masking and recovery.

It should be clear that having only transient or persistent communication is not sufficient in many cases. Likewise, having only synchronous or asynchronous communication facilities will not do either. There is simply need for all types, depending on what the distributed system is being used for. So far, we have mainly concentrated on transient synchronous communication by RPCs and RMIs. The other forms of communication are generally offered through message-oriented communication systems, which are discussed in the following sections. We make a distinction between transient and persistent communication.

### 2.4.2 Message-Oriented Transient Communication

Many distributed systems and applications are built directly on top of the simple message-oriented model offered by the transport layer. To better understand and appreciate the message-oriented systems as part of middleware solutions, we first discuss messaging through transport-level sockets.

**Berkeley Sockets**

Special attention has been paid to standardizing the interface of the transport layer to allow programmers to make use of its entire suite of (messaging) protocols through a simple set of primitives. Also, standard interfaces make it easier to port an application to a different machine.

As an example, we briefly discuss the sockets interface as introduced in Berkeley UNIX. Another important interface is XTI, which stands for the X/Open Transport Interface, formerly called the Transport Layer Interface (TLI), and developed by AT&T. Sockets and XTI are very similar in their model of network programming, but differ in their set of primitives.

Conceptually, a socket is a communication endpoint to which an application can write data that are to be sent out over the underlying network, and from which incoming data can be read. A socket forms an abstraction over the actual communication endpoint that is used by the local operating system for a specific transport protocol. In the following text, we concentrate on the socket primitives for TCP, which are shown in Fig. 2-23.

Servers generally execute the first four primitives, normally in the order given. When calling the socket primitive, the caller creates a new communication endpoint for a specific transport protocol. Internally, creating a communication endpoint means that the local operating system reserves resources to accommodate sending and receiving messages for the specified protocol.

The bind primitive associates a local address with the newly created socket. For example, a server should bind the IP address of its machine together with a (possibly well-known) port number to a socket. Binding tells the operating system that the server wants to receive messages only on the specified address and port.

The listen primitive is called only in the case of connection-oriented communication. It is a nonblocking call that allows the local operating system to reserve enough buffers for a specified maximum number of connections that the caller is willing to accept.

A call to accept blocks the caller until a connection request arrives. When such a request arrives, the local operating system creates a new socket with the same properties as the original one, and returns it to the caller. This approach will allow the server to, for example, fork off a process that will subsequently handle the actual communication through the new connection. The server, in the meantime, can go back and wait for another connection request on the original socket.

Let us now look at the client side. Here, too, a socket must first be created using the socket primitive, but explicitly binding the socket to a local address is not necessary, since the operating system can dynamically allocate a port when the connection is set up. The connect primitive requires that the caller specifies the transport-level address to which a connection request is to be sent. The client is blocked until a connection has been set up successfully, after which both sides can start exchanging information through the write and read primitives, which establish the sending and receiving of data, respectively. Finally, closing a connection is symmetric when using sockets, and is established by having both the client and server call the close primitive. The general pattern followed by a client and server for connection-oriented communication using sockets is shown in Fig. 2-24. Many details concerning network programming using sockets and other interfaces in a UNIX environment can be found in (Stevens, 1998).
The Message-Passing Interface (MPI)

With the advent of high-performance multicomputers, developers have been looking for message-oriented primitives that would allow them to easily write highly efficient applications. This means that the primitives should be at a convenient level of abstraction (to ease application development), and that their implementation incurs minimal overhead. Sockets were deemed insufficient for two reasons. First, they were at the wrong level of abstraction by supporting only simple send and receive primitives. Second, sockets had been designed to communicate across networks using general-purpose protocol stacks such as TCP/IP. They were not considered suitable for the proprietary protocols developed for high-speed interconnection networks, such as those used in COWs or MPPs (which we discussed in Sec. 1.3). Those protocols required an interface that could handle more advanced features, such as different forms of buffering and synchronization.

The result was that most interconnection networks and high-performance multicomputers were shipped with proprietary communication libraries. These libraries offered a wealth of high-level and generally efficient communication primitives. Of course, all libraries were mutually incompatible, so that application developers now had a portability problem.

The need to be hardware independent eventually led to the definition of a standard for message passing, simply called the Message-Passing Interface or MPI. MPI is designed for parallel applications and is such as tailored to transient communication. It makes use of the underlying network, and as such has no concept such as the communication servers shown in Fig. 2-20. Also, it assumes that serious failures such as process crashes or network partitions are fatal and do not require automatic recovery.

MPI assumes communication takes place within a known group of processes. Each group is assigned an identifier. Each process within a group is also assigned a (local) identifier. A (groupID, processID) pair therefore uniquely identifies the source or destination of a message, and is used instead of a transport-level address. There may be several, possibly overlapping groups of processes involved in a computation and that are all executing at the same time.

---

SEC. 2.4 MESSAGE-ORIENTED COMMUNICATION

At the core of MPI are messaging primitives to support most of the forms of transient communication shown in Fig. 2-22(c)-(f), and of which the most intuitive ones are summarized in Fig. 2-25.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_send</td>
<td>Append outgoing message to a local send buffer</td>
</tr>
<tr>
<td>MPI_sendwait</td>
<td>Send a message and wait until copied to local or remote buffer</td>
</tr>
<tr>
<td>MPI_ssend</td>
<td>Send a message and wait until receipt starts</td>
</tr>
<tr>
<td>MPI_ssendrecv</td>
<td>Send a message and wait for reply</td>
</tr>
<tr>
<td>MPI_isend</td>
<td>Pass reference to outgoing message, and continue</td>
</tr>
<tr>
<td>MPI_isendrecv</td>
<td>Pass reference to outgoing message, and wait until receipt starts</td>
</tr>
<tr>
<td>MPI_recv</td>
<td>Receive a message; block if there is none</td>
</tr>
<tr>
<td>MPI_readcheck</td>
<td>Check if there is an incoming message, but do not block</td>
</tr>
</tbody>
</table>

In essence, only the synchronous communication shown in Fig. 2-22(d) is not supported. In other words, MPI does not support having the receiver synchronize with the receiver at the point where a message has been transmitted across the network.

Transient asynchronous communication (Fig. 2-22(c)) is supported by means of the MPI_isend primitive. The sender submits a message for transmission, which is generally first copied to a local buffer in the MPI runtime system. When the message has been copied, the sender continues. The local MPI runtime system will remove the message from its local buffer and take care of transmission as soon as a receiver has called a receive primitive.

There is also blocking send operation, called MPI_send, of which the semantics are implementation dependent. The primitive MPI_send may either block the caller until the specified message has been copied to the MPI runtime system at the sender's side, or until the receiver has initiated a receive operation. The first case corresponds to asynchronous communication as shown in Fig. 2-22(d), whereas the second case is that shown in Fig. 2-22(e).

Synchronous communication by which the sender blocks until its request is accepted for further processing, as shown in Fig. 2-22(e), is available through the MPI_ssend primitive.

Finally, the strongest form of synchronous communication, as shown in Fig. 2-22(f), is also supported. When a sender calls MPI_sendrecv, it sends a request to the receiver and blocks until the latter returns a reply. Basically, this primitive corresponds to a normal RPC.

Both MPI_send and MPI_ssend have a variant that avoids the need to copy a message from a user's buffer to a buffer internal to the local MPI runtime system,
and which both correspond to a form of asynchronous communication. With MPI\_i\_send, a sender passes a pointer to the message after which the MPI runtime system takes care of communication. The sender immediately continues. To prevent overwriting the message before communication completes, MPI offers primitives to check for completion, or even to block if required. As with MPI\_send, whether the message has actually been transferred to the receiver or that it has merely been copied by the local MPI runtime system to an internal buffer is left unspecified.

Likewise, with MPI\_i\_send, a sender also passes only a pointer to the MPI runtime system. When the runtime system indicates it has processed the message, the sender is then guaranteed that the receiver has accepted the message and is now working on it.

The operation MPI\_i\_recv is called to receive a message; it blocks the caller until a message arrives. There is also an asynchronous variant, called MPI\_i\_irecv, by which a receiver indicates that is prepared to accept a message. The receiver can check whether or not a message has indeed arrived, or even block until one does.

The semantics of MPI communication primitives are not always straightforward, and different primitives can sometimes be interchanged without affecting the correctness of a program. The official reason why so many different forms of communication are supported is that it gives implementers of MPI systems enough possibilities for optimizing performance. Cynics might say the committee could not make up its collective mind, so it threw in everything. MPI has been designed for high-performance parallel applications, which makes it easier to understand its diversity in different communication primitives.

More on MPI can be found in \cite{Gropp1998a} The complete reference in which the over 100 functions in MPI are explained in detail, can be found in \cite{Snir1998} and \cite{Gropp1998b}.

### 2.4.3 Message-Oriented Persistent Communication

We now come to an important class of message-oriented middleware services, generally known as message-queuing systems, or just Message-Oriented Middleware (MOM). Message-queuing systems provide extensive support for persistent asynchronous communication. The essence of these systems is that they offer intermediate-term storage capacity for messages, without requiring either the sender or receiver to be active during message transmission. An important difference with Berkeley sockets and MPI, is that message-queuing systems are typically targeted to support message transfers that are allowed to take minutes instead of seconds or milliseconds. We first explain a general approach to message-queuing systems, and conclude this section by comparing them to more traditional systems, notably the Internet e-mail systems.

### SEC. 2.4 MESSAGE-ORIENTED COMMUNICATION

#### Message-Queuing Model

The basic idea behind a message-queuing system is that applications communicate by inserting messages in specific queues. These messages are forwarded over a series of communication servers and are eventually delivered to the destination, even if it was down when the message was sent. In practice, most communication servers are directly connected to each other. In other words, a message is generally transferred directly to a destination server. In principle, each application has its own private queue to which other applications can send messages. A queue can be read only by its associated application, but it is also possible for multiple applications to share a single queue.

An important aspect of message-queuing systems is that a sender is generally given only the guarantees that its message will eventually be inserted in the recipient’s queue. No guarantees are given about when, or even if the message will actually be read, which is completely determined by the behavior of the recipient.

These semantics permit loosely-coupled communication. There is thus no need for the receiver to be executing when a message is being sent to its queue. Likewise, there is no need for the sender to be executing when its message is picked up by the receiver. The sender and receiver can execute completely independently of each other. In fact, once a message has been deposited in a queue, it will remain there until it is removed, irrespective of whether its sender or receiver is executing. This gives us four combinations with respect to the execution mode of the sender and receiver, as shown in Fig. 2-26.

In Fig. 2-26(a), both the sender and receiver are executing during the entire transmission of a message. In Fig. 2-26(b), only the sender is executing, while the receiver is passive, that is, in a state in which message delivery is not possible. Nevertheless, the sender and receiver can still send messages. The combination of a passive sender and an executing receiver is shown in Fig. 2-26(c). In this case, the receiver can read messages that were sent to it, but it is not necessary that their respective senders are executing as well. Finally, in Fig. 2-26(d), we see the situation that the system is storing (and possibly transmitting) messages even while sender and receiver are passive.

Messages can, in principle, contain any data. The only important aspect is that messages are properly addressed. In practice, addressing is done by providing a systemwide unique name of the destination queue. In some cases, message size may be limited, although it is also possible that the underlying system takes care of fragmenting and assembling large messages in a way that is completely transparent to applications. An effect of this approach is that the basic interface offered to applications can be extremely simple, as shown in Fig. 2-27.

The put primitive is called by a sender to pass a message to the underlying system that is to be appended to the specified queue. As we explained, this is a nonblocking call. The get primitive is a blocking call by which an authorized
process can remove the longest pending message in the specified queue. The process is blocked only if the queue is empty. Variations on this call allow searching for a specific message in the queue, for example, using a priority, or a matching pattern. The nonblocking variant is given by the poll primitive. If the queue is empty, or if a specific message could not be found, the calling process simply continues.

Finally, most queueing systems also allow a process to install a handler as a callback function, which is automatically invoked whenever a message is put into the queue. Callbacks can also be used to automatically start a process that will fetch messages from the queue if no process is currently executing. This approach is often implemented by means of a daemon on the receiver’s side that continuously monitors the queue for incoming messages and handles accordingly.

Queues are managed by queue managers. Normally, a queue manager interacts directly with the application that is sending or receiving a message. However, there are also special queue managers that operate as routers, or relays: they forward incoming messages to other queue managers. In this way, a message-queueing system may gradually grow into a complete, application-level, overlay network, on top of an existing computer network. This approach is similar to the construction of the early MBone over the Internet, in which ordinary user processes were configured as multicast routers. These days, many routers already support multicasting, often rendering overlay multicasting less useful.

![Figure 2-27. Basic interface to a queue in a message-queueing system.](image)

![Figure 2-28. The relationship between queue-level addressing and network-level addressing.](image)
Relays can be convenient for a number of reasons. For example, in many message-queuing systems, there is no general naming service available that can dynamically maintain queue-to-destination mappings. Instead, the topology of the queuing network is static, and each queue manager needs a copy of the queue-to-destination mapping. It is needless to say that in large-scale queuing systems, this approach can easily lead to network-management problems.

One solution is to use a few routers that know about the network topology. When a sender $A$ puts a message for destination $B$ in its local queue, that message is first transferred to the nearest router, say $R_1$, as shown in Fig. 2.29. At that point, the router knows what to do with the message and forwards it in the direction of $B$. For example, $R_1$ may derive from $B$’s name that the message needs to be forwarded to router $R_2$. In this way, only the routers need to be updated when queues are added or removed, while every other queue manager has to know only where the nearest router is.

![Diagram of message-queuing system with routers](image)

Figure 2.29. The general organization of a message-queuing system with routers.

Relays can thus generally help build scalable message-queuing systems. However, as queuing networks grow, it is clear that the manual configuration of networks will soon become unmanageable. The only solution is to adopt dynamic routing schemes as is done for computer networks. In that respect, it is somewhat surprising that such solutions are not yet integrated into some of the popular message-queuing systems.

Another reason why relays are used is that they allow for secondary processing of messages. For example, messages may need to be logged for reasons of security or fault tolerance. A special form of relay that we discuss in the next section is one that acts as a gateway, transforming messages into a format that can be understood by the receiver.

Finally, relays can be used for multicasting purposes. In that case, an incoming message is simply put into each send queue.

**Message Brokers**

An important application area of message-queuing systems is integrating existing and new applications into a single, coherent distributed information system. Integration requires that applications can understand the messages they receive. In practice, this requires the sender to have its outgoing messages in the same format as that of the receiver.

The problem with this approach is that each time an application is added to the system that requires a separate message format, each potential receiver will have to be adjusted in order to produce that format.

An alternative is to agree on a common message format, as is done with traditional network protocols. Unfortunately, this approach will generally not work for message-queuing systems. The problem is the level of abstraction at which these systems operate. A common message format makes sense only if the collection of processes that use it are all in relatively close communication and the collection of applications that use it is highly diverse (which it often is), then the best common format may well be no more than a sequence of bytes.

Although a few common message formats for specific application domains have been defined, the general approach is to learn to live with different formats, and try to provide the means to make conversions as simple as possible. In message-queuing systems, conversions are handled by special nodes in a queuing network known as **message brokers**. A message broker acts as an application-level gateway in a message-queuing system. Its main purpose is to convert incoming messages to a format that can be understood by the destination application. Note that a message-queuing system, a message broker is just another application, as shown in Fig. 2.30. In other words, a message broker is generally not considered to be an integral part of the queuing system.

A message broker can be as simple as a reformatter for messages. For example, assume an incoming message contains a table from a database, in which records are separated by a special `end-of-record` delimiter and fields within a record have a known, fixed length. If the destination application expects a different delimiter between records, and also expects that fields have variable lengths, a message broker can be used to convert messages to the format expected by the destination.

In a more advanced setting, a message broker may act as an application-level gateway, such as one that handles the conversion between X.400 and Internet...
e-mail messages. In such cases, it cannot be guaranteed that all information contained in the incoming message can actually be transformed into something appropriate for the outgoing message. In other words, it may be necessary to accept a certain loss of information during transformation (see also Houttuin, 1993; and Alvestrand, 1998).

At the heart of a message broker lies a database of rules that specify how a message in format $X$ is to be converted to a message in format $Y$. The problem is defining the rules. Most message broker products come with sophisticated rule development tools, but the bottom line is still that rules are to be manually entered into the database. Rules can be formulated in a special conversion language, but many message broker products also allow program conversions using normal programming languages. Setting up a message broker is thus generally a highly laborious task.

### A Note on Message-Queuing Systems

Considering what we have said about message-queuing systems, it would appear that they have long existed in the form of implementations for e-mail services. E-mail systems are generally implemented through a collection of mail servers that store and forward messages on behalf of the users on hosts directly connected to the server. Routing is generally left out, as e-mail systems can make direct use of the underlying transport services. For example, in the mail protocol for the Internet, SMTP (Postel, 1982), a message is transferred by setting up a direct TCP connection to the destination mail server.

What makes e-mail systems special compared to message-queuing systems is that they are primarily aimed at providing direct support for end users. This explains, for example, why a number of groupware applications are based directly on an e-mail system (Khoshafian and Buckiewicz 1995). In addition, e-mail systems may have very specific requirements such as automatic message filtering, support for advanced messaging databases (e.g., to easily retrieve previously stored messages), and so on.

General message-queuing systems are not aimed at supporting only end users. An important issue is that they are set up to enable persistent communication between processes, regardless of whether a process is running a user application, handling access to a database, performing computations, and so on. This approach leads to a different set of requirements for message-queuing systems than pure e-mail systems. For example, e-mail systems generally need not provide guaranteed message delivery, message priorities, logging facilities, efficient multicasting, load balancing, fault tolerance, and so on.

General-purpose message-queuing systems, therefore, have a wide range of applications, including e-mail, workflow, groupware, and batch processing. However, perhaps the most important application area is the integration of a (possibly widely dispersed) collection of databases, or database applications, into a multidatabase (information) system. See also (Ozsu and Valduriez 1999) and (Shehd and Lane 1990). For example, a query expanding several databases may need to be split into subqueries that are forwarded to individual databases. Message-queuing systems assist by providing the basic means to package each subquery into a message and routing it to the appropriate database. Other communication facilities we have discussed in this chapter are far less appropriate.

### 2.4.4 Example: IBM MQSeries

To help understand how message-queuing systems work in practice, let us take a look at one specific system, namely the MQSeries from IBM (Gilman and Schreiber, 1996). This system is gaining popularity in the relatively traditional area of IBM mainframes, which are being used to access and manipulate large-scale databases. An important application area for MQSeries is finance.

#### Overview

The basic architecture of an MQSeries queuing network is quite straightforward, and is shown in Fig. 2.31. All queues are managed by *queue managers*. A queue manager is responsible for removing messages from its send queues, and forwarding those to other queue managers. Likewise, a queue manager is responsible for handling incoming messages by picking them up from the underlying network and subsequently storing each message in the appropriate input queue.

Queue managers are pairwise connected through *message channels*, which are an abstraction of transport-level connections. A message channel is a unidirectional, reliable connection between a sending and a receiving queue manager,
through which queued messages are transported. For example, an Internet-based message channel is implemented as a TCP connection. Each of the two ends of a message channel is managed by a message channel agent (MCA). A sending MCA is basically doing nothing else than checking send queues for a message, wrapping it into a transport-level packet, and sending it along the connection to its associated receiving MCA. Likewise, the basic task of a receiving MCA is listening for an incoming packet, unwrapping it, and subsequently storing the unwrapped message into the appropriate queue.

![Diagram of IBM's MQSeries message-queueing system](image)

Figure 2-31. General organization of IBM's MQSeries message-queueing system.

Queue managers can be linked into the same process as the application for which it manages the queues. In that case, the queues are hidden from the application behind a standard interface, but effectively can be directly manipulated by the application. An alternative organization is one in which queue managers and applications run on separate machines. In that case, the application is offered the same interface as when the queue manager is collocated on the same machine. However, the interface is implemented as a proxy that communicates with the queue manager using traditional RPC-based synchronous communication. In this way, MQSeries basically retains the model that only queues local to an application can be accessed.

**Channels**

An important component of MQSeries is formed by the message channels. Each message channel has exactly one associated send queue from which it fetches the messages it should transfer to the other end. Transfer along the channel can take place only if both its sending and receiving MCA are up and running.

**SEC. 2.4 MESSAGE-ORIENTED COMMUNICATION**

Apart from starting both MCAs manually, there are several alternative ways to start a channel, some of which we discuss next.

One alternative is to have an application directly start its end of a channel by activating the sending or receiving MCA. However, from a transparency point of view, this is not a very attractive alternative. A better approach to start a sending MCA is to configure the channel’s send queue to set off a trigger when a message is first put into the queue. That trigger is associated with a handler to start the sending MCA so that it can remove messages from the send queue.

Another alternative is to start an MCA over the network. In particular, if one side of a channel is already active, it can send a control message requesting the other MCA to be started. Such a control message is sent to a daemon listening to a well-known address on the same machine as where the other MCA is to be started.

Channels are stopped automatically after a specified time has expired in which no more messages were dropped into the send queue.

Each MCA has a set of associated attributes that determine the overall behavior of a channel. Some of the attributes are listed in Fig. 2-32. Attribute values of the sending and receiving MCA should be compatible and perhaps negotiated first before a channel can be set up. For example, both MCAs should obviously support the same transport protocol. An example of a nonnegotiable attribute is whether or not messages are to be delivered in the same order as they are put into the send queue. If one MCA wants FIFO delivery, the other must comply. An example of a negotiable attribute value is the maximum message length, which will simply be chosen as the minimum value specified by either MCA.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport type</td>
<td>Determines the transport protocol to be used</td>
</tr>
<tr>
<td>FIFO delivery</td>
<td>Indicates that messages are to be delivered in the order they are sent</td>
</tr>
<tr>
<td>Message length</td>
<td>Maximum length of a single message</td>
</tr>
<tr>
<td>Setup retry count</td>
<td>Specifies maximum number of retries to start up the remote MCA</td>
</tr>
<tr>
<td>Delivery retries</td>
<td>Maximum times MCA will try to put received message into queue</td>
</tr>
</tbody>
</table>

Figure 2-32. Some attributes associated with message channel agents.

**Message Transfer**

To transfer a message from one queue manager to another (possibly remote) queue manager, it is necessary that each message carries its destination address, for which a transmission header is used. An address in MQSeries consists of two parts. The first part consists of the name of the queue manager to which the message is to be delivered. The second part is the name of the destination queue, storing under that manager to which the message is to be appended.
Besides the destination address, it is also necessary to specify the route that a message should follow. Route specification is done by providing the name of the local send queue to which a message is to be appended. Thus it is not necessary to provide the full route in a message. Recall that each message channel has exactly one send queue. By telling to which send queue a message is to be appended, we effectively specify to which neighboring queue manager a message is to be forwarded.

In most cases, routes are explicitly stored inside a queue manager in a routing table. An entry in a routing table is a pair (destQM, sendQ), where destQM is the name of the destination queue manager, and sendQ is the name of the local send queue to which message for that queue manager should be appended. (A routing table entry is called an alias in MQSeries.)

It is possible that a message needs to be transferred across multiple queue managers before reaching its destination. Whenever such an intermediate queue manager receives the message, it simply extracts the name of the destination queue manager from the message header, and does a routing-table look-up to find the local send queue to which the message should be appended.

It is important to realize that each queue manager has a systemwide unique name that is effectively used as an identifier for that queue manager. The problem with using these names is that replacing a queue manager, or changing its name, will affect all applications that send messages to it. Problems can be alleviated by using a local alias for queue manager names. An alias defined within a queue manager M1 is another name for a queue manager M2, but which is available only to applications interfacing to M1. An alias allows the use of the same (logical) name for a queue, even if the queue manager of that queue changes. Changing the name of a queue manager requires that we change its alias in all queue managers. However, applications can be left unaffected.

The principle of using routing tables and aliases is shown in Fig. 2-33. For example, an application linked to queue manager QMA can refer to a remote queue manager using the local alias LA1. The queue manager will first look up the actual destination in the alias table to find it is queue manager QMC. The route to QMC should be appended to the outgoing queue SQ1, which is used to transfer messages to queue manager QMB. The latter will use its routing table to forward the message to QMC.

Following this approach of routing and aliasing, leads to a relatively simple programming interface for applications, called the Message Queue Interface (MQI). The most important primitives of MQI are summarized in Fig. 2-34.

To put messages into a queue, an application calls the MQOpen primitive, specifying a destination queue in a specific queue manager. The queue manager can be named using the locally available alias. Whether the destination queue is actually remote or not is completely transparent to the application. MQOpen should also be called if the application wants to get messages from its local queue.

Figure 2-33. The general organization of an MQSeries queuing network using routing tables and aliases.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQOpen</td>
<td>Open a (possibly remote) queue</td>
</tr>
<tr>
<td>MQClose</td>
<td>Close a queue</td>
</tr>
<tr>
<td>MQPut</td>
<td>Put a message into an opened queue</td>
</tr>
<tr>
<td>MQGet</td>
<td>Get a message from a (local) queue</td>
</tr>
</tbody>
</table>

Figure 2-34. Primitives available in IBM MQSeries MQI.

Only local queues can be opened for reading incoming messages. When an application is finished with accessing a queue, it should close it by calling MQClose.

Messages can be written to, or read from, a queue using MQPut and MQGet, respectively. In principle, messages are removed from a queue on a priority basis. Messages with the same priority are removed on a first-in, first-out basis, that is, the longest pending message is removed first. It is also possible to request for specific messages. Finally, MQSeries provides facilities to signal applications when messages have arrived, thus avoiding that an application will continuously have to poll a message queue for incoming messages.

2.5 STREAM-ORIENTED COMMUNICATION

Communication as discussed so far has concentrated on exchanging more-or-less independent and complete units of information. Examples include a request for invoking a procedure or method, the reply to such a request, and messages...
exchanged between applications as in message-queueing systems. The characteristic feature of this type of communication is that it does not matter at what particular point in time communication takes place. Although a system may perform too slow or too fast, timing has no effect on correctness.

There are also forms of communication in which timing plays a crucial role. Consider, for example, an audio stream built up as a sequence of 16-bit samples, each representing the amplitude of the sound wave as is done through Pulse Code Modulation (PCM). Also assume that the audio stream represents CD quality, meaning that the original sound wave has been sampled at a frequency of 44100 Hz. To reproduce the original sound, it is essential that the samples in the audio stream are played out in the order they appear in the stream, but also at intervals of exactly 1/44100 sec. Playing out at a different rate will produce an incorrect version of the original sound.

The question we address in this section is which facilities a distributed system should offer to exchange time-dependent information such as audio and video streams. Various network protocols that deal with stream-oriented communication are discussed in (Halsall, 2001). Steinmetz and Nahrstedt (1995) provide an overall introduction to multimedia issues, of which stream-oriented communication is part of.

### 2.5.1 Support for Continuous Media

Support for the exchange of time-dependent information is often formulated as support for continuous media. A medium refers to the means by which information is conveyed. These means include storage and transmission media, presentation media such as a monitor, and so on. An important type of medium is the way that information is represented. In other words, how is information encoded in a computer system? Different representations are used for different types of information. For example, text is generally encoded as ASCII or Unicode. Images can be represented in different formats such as GIF or JPEG. Audio streams can be encoded in a computer system by, for example, taking 16-bit samples using PCM.

In continuous (representation) media, the temporal relationships between different data items are fundamental to correctly interpreting the data. Typical examples of discrete media include representations of text and still images, but also object code or executable files.

### Data Stream

To capture the exchange of time-dependent information, distributed systems generally provide support for **data streams**. A data stream is nothing but a sequence of data units. Data streams can be applied to discrete as well as continuous media. For example, UNIX pipes or TCP/IP connections are typical examples of (byte-oriented) discrete data streams. Playing an audio file typically requires setting up a continuous data stream between the file and the audio device.

Timing is crucial to continuous data streams. To capture timing aspects, a distinction is often made between different transmission modes. In **asynchronous transmission mode**, the data items in a stream are transmitted one after the other, but there are no further timing constraints on when transmission of items should take place. This is typically the case for discrete data streams. For example, a file can be transferred as a data stream, but it is mostly irrelevant exactly when the transfer of each item completes.

In **synchronous transmission mode**, there is a maximum end-to-end delay defmed for each unit in a data stream. Whether a data unit is transferred much faster than the maximum tolerated delay is not important. For example, a sensor may sample temperature at a certain rate and pass it through a network to an operator. In that case, it may be important that the end-to-end propagation time through the network is guaranteed to be lower than the time interval between taking samples, but it cannot do any harm if samples are propagated much faster than strictly necessary.

Finally, in **isochronous transmission mode**, it is necessary that data units are transferred on time. This means that data transfer is subject to a maximum and minimum end-to-end delay, also referred to as bounded (delay) jitter. Isochronous transmission mode is particularly interesting for distributed multimedia systems, as it plays a crucial role in representing audio and video. In this chapter, we consider only continuous data streams using isochronous transmission, which we will refer to simply as streams.

Streams can be simple or complex. A **simple stream** consists of only a single sequence of data, whereas a **complex stream** consists of several related simple streams, called **substreams**. The relation between the substreams in a complex stream is often also time dependent. For example, stereo audio can be transmitted by means of a complex stream consisting of two substreams, each used for a single audio channel. It is important, however, that those two substreams are continuously synchronized. In other words, data units from each stream are to be communicated pairwise to ensure the effect of stereo. Another example of a complex stream is one for transmitting a movie. Such a stream could consist of a single...
video stream, along with two streams for transmitting the sound of the movie in stereo. A fourth stream might contain subtitles for the deaf, or a translation into a different language than the audio. Again, synchronization of the substreams is important. If synchronization fails, reproduction of the movie fails. We return to stream synchronization below.

A stream can often be considered as a virtual connection between a source and a sink. The source or sink could be a process, but could also be a device. For example, when sending data across a network, we may have a sending process reading an audio file from disk, and sending it, byte by byte, through a network. The sink may be a process fetching the bytes as they come in, and passing them to the local audio device. This situation is shown in Fig. 2-35(a). On the other hand, in multimedia distributed systems, support may be provided for setting up a direct connection between source and sink. For example, the video stream produced by a camera may possibly be directly forwarded to a display device, as shown in Fig. 2-35(b).

![Figure 2-35. (a) Setting up a stream between two processes across a network. (b) Setting up a stream directly between two devices.](image)

Another issue is whether there is just a single source or sink, or if multiparty communication can be set up. The most common situation in multiparty communication is attaching multiple sinks to a stream. In other words, the data stream is multicast to several receivers, as shown in Fig. 2-36.

![Figure 2-36. An example of multicasting a stream to several receivers.](image)

The main problem with multicast streaming is when the receivers have different requirements with respect to the quality of the stream. Consider, for example, a source transmitting a high-quality stereo-sound movie. This may require a complex stream consisting of a video substream in which images are transmitted at 50 Hz and two audio substreams each having CD-quality audio. Even when using advanced compression techniques, the complex stream may require a bandwidth of more than $30 \times 10^6$ bits/sec (Steinmetz and Nahorst, 1995). Not every receiver may be capable of processing so much data. Therefore, the stream should be configured with filters (Yeadon et al., 1996) that adjust the quality of an incoming stream differently for outgoing streams, as also shown in Fig. 2-36. We return to controlling the quality of a stream below.

### 2.5.2 Streams and Quality of Service

Time-dependent (and other nonfunctional) requirements are generally expressed as **Quality of Service (QoS)** requirements. These requirements describe what is needed from the underlying distributed system and network to ensure that, for example, the temporal relationships in a stream can be preserved. QoS for continuous data streams mainly concerns timeliness, volume, and reliability. In this section we take a closer look at QoS and its relation to setting up a stream.

**Specifying QoS**

Expressing QoS requirements can be done in several ways. One approach is to provide a precise **flow specification** containing bandwidth requirements, transmission rates, delays, etc. An example of such a flow specification is given in Fig. 2-37, developed by Partridge (1992).

In Partridge's model, the characteristics of the stream are formulated in terms of a **token bucket algorithm**, which specifies how the stream will shape its network traffic. The principle of a token bucket is shown in Fig. 2-38. The basic idea is that tokens are generated at a constant rate. A token represents a fixed number
of bytes that an application is allowed to pass to the network. Tokens are buffered in a bucket, which has a limited capacity. When the bucket is full, tokens will simply be dropped. Each time the application wants to pass a data unit of size \( N \) to the network, it will have to remove enough tokens from the bucket that jointly represent at least \( N \) bytes. So, for example, if each token represents \( k \) bytes, the application will have to remove at least \( N/k \) tokens from the bucket.

The effect of a token bucket algorithm is that data is passed to the network at a relatively constant rate, determined by the rate of generating tokens. However, it also allows for some burstiness as an application is allowed to provide a complete bucket worth of tokens to the network in a single operation. To avoid extreme bursts, the data stream may also limit its rate to a specified maximum. In a flow specification, the application promises that it will offer data units to the communication system according to the output of the token bucket algorithm.

In addition to specifying the temporal relations between data units, a flow specification also consists of service requirements. Loss sensitivity in combination with a loss interval specify what a maximum acceptable loss rate would be (e.g., 1 byte per minute). Burst loss sensitivity specifies how many consecutive data units may be lost.

The minimum delay noticed specifies how long the network can delay the delivery of a data unit before the receiver notices the delay. Related to this measure is the maximum delay variation, which specifies the maximum tolerated jitter. Jitter specifications are especially important for video and audio.

Finally, the quality of guarantee is a number indicating how serious the service requirements should be taken. Basically, a low number tells that if the communication system cannot provide the required services, no real harm is done. A high number, on the other hand, indicates that if no firm guarantees can be given, then the system should not establish a stream because the client will go for nothing less than that required.

A problem with flow specifications such as these is that an application may simply not know its own requirements. In particular, forcing a user to specify quality in terms of token bucket parameters, loss sensitivity, etc., would soon put a service provider out of business. A different approach is therefore to classify a stream, and subsequently provide reasonable defaults for detailed flow specifications. For example, a user may need to specify only whether a stream is for audio or video. For an audio stream, a choice may be available between high, medium, or low quality. A similar classification can be offered for video streams.

As also argued in (Partridge, 1994), classification is not very different from detailed flow specifications. The difference lies merely in the number of parameters that need to be specified, and the number of distinct values that each parameter can have.

### Setting up a Stream

Once a data stream has been described, for example, in the form of a flow specification, a distributed system is in a position to allocate resources to set up a stream that satisfies the QoS requirements. Resources in the context of stream management mainly concern bandwidth, buffers, and processing capacity. Bandwidth reservation takes place by ensuring that data units are properly scheduled for transmission, for example, by assigning scheduling priorities. By allocating buffers in routers and operating systems, data units can be queued for further processing. Finally, it is essential that data units are processed in time, requiring that associated tasks, such as schedulers, encoders and decoders, filters, and the like, are properly scheduled for CPU time.

One of the problems that needs to be solved is that the parameters in which the characteristics and QoS requirements of a data stream are specified, need not directly correspond to analogous resource parameters. For example, specifying that the network should guarantee that no more than \( k \) consecutive data units may be lost, may have to be translated into statically allocating buffers along a path of routers from the source to the destination. Such allocations can, in fact, be calculated by using other stream characteristics, and may result in a firmly or statistically guaranteed service from the network.

Unfortunately, there is currently no single best model for (1) specifying QoS parameters, (2) generically describing resources in any communication system,
and (3) translating QoS parameters to resource usage. The lack of such a model is the reason why expressing and establishing quality of service is often so difficult, and why different systems use different and incompatible approaches.

To make matters more concrete, and considering that QoS in distributed systems is mainly dependent on the services the underlying network can offer, let us take a brief look at a specific QoS protocol for reserving resources for continuous streams. The Resource ReSerVation Protocol (RSVP) is a transport-level control protocol for enabling resource reservations in network routers (Zhang et al., 1993; Braden et al., 1997).

Senders in RSVP provide a flow specification characterizing the data stream in terms of bandwidth, delay, jitter, and so on, very similar to the flow specification shown in Fig. 2-37. This specification is handed over to an RSVP process that is located on the same machine as the sender, as shown in Fig. 2-39. The RSVP process does not interpret the flow specification. In fact, the only thing it does when receiving a flow specification from a sender is store it locally. RSVP is a receiver-initiated QoS protocol. In other words, receivers are required to send reservation requests along the path to the sender. By storing a flow specification, RSVP can prevent more resources from being reserved than are strictly necessary.

![Figure 2.39](image-url)  
*Figure 2.39. The basic organization of RSVP for resource reservation in a distributed system.*

A sender in RSVP first sets up a path to potential receivers and provides the flow specification of the data stream to each intermediate node. When a receiver is ready to accept incoming data units, it first places a reservation request along its upstream path to the sender. The format of such a request is essentially the same as the original flow specification, but parameter values may be set to reflect a lower QoS than what the sender specified would be necessary for the most demanding receivers.

When an RSVP process receives a reservation request, it passes the request to the admission control module to check whether enough resources are available. The request is also passed to the policy control module to check whether the receiver has permission to make the reservation. If these two tests succeed, resources can be reserved.

Resource reservation is highly dependent on the data link layer. In fact, to make RSVP work, the RSVP process will have to translate the QoS parameters of its flow specifications into things that make sense to the data link layer. A simple example is translating a request for lots of bandwidth to assigning the highest priority to each frame carrying data of the stream. Based on the initial flow specification (which indicates the maximum data rate the sender will ever generate), and the available bandwidth at the data link layer, such a translation may be sufficient to meet a receiver's QoS requirements.

Another approach is needed when the data link layer provides its own set of parameters for specifying QoS requirements, as in ATM networks. In ATM networks, data is transmitted in units called cells, which consist of a 48-byte payload field and a 5-byte header. ATM will allow an RSVP process to specify the maximum cell rate, the long-term average cell rate, the minimum acceptable cell rate, and the maximum acceptable jitter between cells. There are other QoS parameters as well. In this case, it is the task of the RSVP process to translate the stream-oriented flow specifications into ATM-specific parameter values. The ATM layer will subsequently take care that the QoS requirements are indeed met.

### 2.5.3 Stream Synchronization

An important issue in multimedia systems is that different streams, possibly in the form of a complex stream, are mutually synchronized. Synchronization of streams deals with maintaining temporal relations between streams. Two types of synchronization occur.

The simplest form of synchronization is that between a discrete data stream and a continuous data stream. Consider, for example, a slide show on the Web that has been enhanced with audio. Each slide is transferred from the server to the client in the form of a discrete data stream. At the same time, the client should play out a specific (part of an) audio stream that matches the current slide that is also fetched from the server. In this case, the audio stream is to be synchronized with the presentation of slides.

A more demanding type of synchronization is that between continuous data streams. A daily example is playing a movie in which the video stream needs to be synchronized with the audio, commonly referred to as lip synchronization. Another example of synchronization is playing a stereo audio stream consisting of

SEC. 2.5 STREAM-ORIENTED COMMUNICATION 127
two substreams, one for each channel. Proper play out requires that the two substreams are tightly synchronized: a difference of more than 20 μsec can distort the stereo effect.

Synchronization takes place at the level of the data units of which a stream is made up. In other words, we can synchronize two streams only between data units. The choice of what exactly a data unit is depends very much on the level of abstraction at which a data stream is viewed. To make things concrete, consider again a CD-quality (single-channel) audio stream. At the finest granularity, such a stream appears as a sequence of 16-bit samples. With a sampling frequency of 44100 Hz, synchronization with other audio streams could, in theory, take place approximately every 23 msec. For high-quality stereo effects, it turns out that synchronization at this level is indeed necessary.

However, when we consider synchronization between an audio stream and a video stream for lip synchronization, a much coarser granularity can be taken. As we explained, video frames need to be displayed at a rate of 25 Hz or more. Taking the widely used NTSC standard of 30 Hz, we could group audio samples into logical units that last as long as a video frame is displayed (33 msec). With an audio sampling frequency of 44100 Hz, an audio data unit can thus be as large as 16 bytes (assuming each sample is 16 bits). In practice, larger units lasting 40 or even 80 msec can be tolerated (Steinmetz, 1996).

Synchronization Mechanisms

We now come to the issue how synchronization is actually achieved. Two issues need to be distinguished: (1) the basic mechanisms for synchronizing two streams, and (2) the distribution of those mechanisms in a networked environment.

Synchronization mechanisms can be viewed at different levels of abstraction. At the lowest level, synchronization is done explicitly by operating on the data units of simple streams. This principle is shown in Fig. 2-40. In essence, there is a process that simply executes read and write operations on several simple streams, ensuring that those operations adhere to specific timing and synchronization constraints.

For example, consider a movie that is presented as two input streams. The video stream contains uncompressed low-quality images of 320×240 pixels, each encoded by a single byte, leading to video data units of 76,800 bytes each. Assume that images are to be displayed at 30 Hz, or one image every 33 msec. Assume that audio is to be played back at 44.1 kHz, or 11.760 bytes (assuming each sample is 16 bits). In practice, larger units lasting 40 or even 80 msec can be tolerated (Steinmetz, 1996).

The principle of explicit synchronization on the level data units available. A better approach is to offer an application an interface that allows it to more easily control streams and devices. Returning to our example, assume that the video display has a control interface that allows it to specify the rate at which images should be displayed. In addition, the interface offers the facility to register a user-defined handler that is called each time a new image has arrived. An analogous interface is offered by the audio device. With these control interfaces, an application developer can write a simple monitor program consisting of two handlers, one for each stream, that jointly check if the video and audio streams are sufficiently synchronized, and if necessary, adjust the rate at which video or audio units are presented.

This last example is illustrated in Fig. 2-41, and is typical for many multimedia middleware systems. In effect, multimedia middleware offers a collection of interfaces for controlling audio and video streams, including interfaces for controlling devices such as monitors, cameras, microphones, etc. Each device and stream has its own high-level interfaces, including interfaces for notifying an application when some event occurred. The latter are subsequently used to write handlers for synchronizing streams. Examples of such interfaces are given in (Blair and Stefani, 1998).

The distribution of synchronization mechanisms is another issue that needs to be looked at. First, the receiving side of a complex stream consisting of substreams that require synchronization, needs to know exactly what to do. In other words, it must have a complete synchronization specification, locally available. Common practice is to provide this information implicitly by multiplexing the different streams into a single stream containing all data units, including those for synchronization.

This latter approach to synchronization is followed for MPEG streams. The MPEG (Motion Picture Experts Group) standards form a collection of algorithms for compressing video and audio. Several MPEG standards exist. MPEG-2, for example, was originally designed for compressing broadcast quality video into 4 to 6 Mbps. In MPEG-2, an unlimited number of continuous and discrete streams...
One the most widely used abstractions is the Remote Procedure Call (RPC). The essence of an RPC is that a service is implemented by means of a procedure, of which the body is executed at a server. The client is offered only the signature of the procedure, that is, the procedure’s name along with its parameters. When the client calls the procedure, the client-side implementation, called a stub, takes care of wrapping the parameter values into a message and sending that to the server. The latter calls the actual procedure and returns the result, again in a message. The client’s stub extracts the result values from the return message and passes it back to the calling client application.

RPCs are aimed at achieving access transparency. However, they offer relatively poor support for passing references. In this sense, remote objects offer better transparency. A Remote Method Invocation (RMI) is essentially an RPC, but now specific for a remote object. The main difference is that RMI allow system-wide object references to be passed as parameters.

RPCs and RMI offer synchronous communication facilities, by which a client is blocked until the server has sent a reply. Although variations of either mechanism exist by which this strict synchronous model is relaxed, it turns out that general-purpose, high-level message-oriented models are often more convenient.

In message-oriented models, the issues are whether or not communication is persistent, and whether or not communication is synchronous. The essence of persistent communication is that a message that is submitted for transmission, is stored by the communication system as long as it takes to deliver it. In other words, neither the sender nor the receiver needs to be up and running for message transmission to take place. In persistent communication, no storage facilities are offered, so that the receiver must be prepared to accept the message when it is sent.

In asynchronous communication, the sender is allowed to continue immediately after the message has been submitted for transmission, possibly before it has even been sent. In asynchronous communication, the sender is blocked at least until a message has been received. Alternatively, the sender may be blocked until message delivery has taken place or even until the receiver has responded as with RPCs.

Message-oriented middleware models generally offer persistent asynchronous communication, and are used where RPCs and RMI are not appropriate. They are primarily used to assist the integration of (widely dispersed) collections of databases into large-scale information systems. Other applications include e-mail and workflow.

A completely different form of communication is that of streaming, in which the issue is whether or not two successive messages have a temporal relationship. In continuous data streams, a maximum end-to-end delay is specified for each message. In addition, it is also required that messages are sent subject to a minimum end-to-end delay. Typical examples of such continuous data streams are
video and audio streams. Exactly what the temporal relations are, or what is expected from the underlying communication subsystem in terms of quality of service is often difficult to specify, and to implement. A complicating factor is the role of jitter. Even if the average performance is acceptable, substantial variations in delivery time may lead to unacceptable performance.

PROBLEMS

1. In many layered protocols, each layer has its own header. Surely it would be more efficient to have a single header at the front of each message with all the control in it than all these separate headers. Why is this not done?

2. Why are transport-level communication services often inappropriate for building distributed applications?

3. A reliable multicast service allows a sender to reliably pass messages to a collection of receivers. Does such a service belong to a middleware layer, or should it be part of a lower-level layer?

4. Consider a procedure incr with two integer parameters. The procedure adds one to each parameter. Now suppose that it is called with the same variable twice, for example, as incr(i, i). If i is initially 0, what value will it have afterward if call-by-reference is used? How about if copy/restore is used?

5. C has a construction called a union, in which a field of a record (called a struct in C) can hold any one of several alternatives. At run time, there is no sure-fire way to tell which one is in there. Does this feature of C have any implications for remote procedure call? Explain your answer.

6. One way to handle parameter conversion in RPC systems is to have each machine send parameters in its native representation, with the other one doing the translation, if need be. The native system could be indicated by a code in the first byte. However, since locating the first byte in the first word is precisely the problem, can this actually work?

7. Assume a client calls an asynchronous RPC to a server, and subsequently waits until the server returns a result using another asynchronous RPC. Is this approach the same as letting the client execute a normal RPC? What if we replace the asynchronous RPCs with one-way RPCs?

8. Instead of letting a server register itself with a daemon as is done in DCE, we could also choose to always assign it the same endpoint. That endpoint can then be used in references to objects in the server’s address space. What is the main drawback of this scheme?

9. Give an example implementation of an object reference that allows a client to bind to a transient remote object.

10. Java and other languages support exceptions, which are raised when an error occurs. How would you implement exceptions in RPCs and RMIs?

11. Would it be useful to also make a distinction between static and dynamic RPCs?

12. Some implementations of distributed-object middleware systems are entirely based on dynamic method invocations. Even static invocations are compiled to dynamic ones. What is the benefit of this approach?

13. Describe how connectionless communication between a client and a server proceeds when using sockets.

14. Explain the difference between the primitives mpi_.bsend and mpi_.isend in MPI.

15. Suppose that you could make use of only transient asynchronous communication primitives, including only an asynchronous receive primitive. How would you implement primitives for transient synchronous communication?

16. Now suppose you could make use of only transient synchronous communication primitives. How would you implement primitives for transient asynchronous communication?

17. Does it make sense to implement persistent asynchronous communication by means of RPCs?

18. In the text we stated that in order to automatically start a process to fetch messages from an input queue, a daemon is often used that monitors the input queue. Give an alternative implementation that does not make use of a daemon.

19. Routing tables in IBM MQSeries, and in many other message-queuing systems, are configured manually. Describe a simple way to do this automatically.

20. How would you incorporate persistent asynchronous communication into a model of communication based on RMIs to remote objects?

21. With persistent communication, a receiver generally has its own local buffer where messages can be stored when the receiver is not executing. To create such a buffer, we may need to specify its size. Give an argument why this is preferable, as well as one against specification of the size.

22. Explain why transient synchronous communication has inherent scalability problems, and how these could be solved.

23. Give an example where multicasting is also useful for discrete data streams.

24. How could you guarantee a maximum end-to-end delay when a collection of computers is organized in a (logical or physical) ring?

25. How could you guarantee a minimum end-to-end delay when a collection of computers is organized in a (logical or physical) ring?

26. Imagine we have a token bucket specification where the maximum data unit size is 1000 bytes, the token bucket rate is 10 million bytes/sec, the token bucket size is 1 million bytes, and the maximum transmission rate is 50 million bytes/sec. How long can a burst of maximum speed last?
27. For this exercise you are to implement a simple client-server system using RPC. The server offers one procedure, next, which takes an integer as input and returns its successor as output. Write a stub procedure called next for use on the client side. Its job is to send the parameter to the server using UDP and wait for the response, timing out if the response takes too long. The server procedure should listen on a known port, accept requests, carry them out, and send back the results.

In the preceding chapter, we concentrated on communication in distributed systems. Communication takes place between processes, and in this chapter, we take a closer look at how the different types of processes play a crucial role in distributed systems. The concept of a process originates from the field of operating systems where it is generally defined as a program in execution. From an operating-system perspective, the management and scheduling of processes are perhaps the most important issues to deal with. However, when it comes to distributed systems, other issues turn out to be equally or more important.

For example, to efficiently organize client-server systems, it is often convenient to make use of multithreading techniques. As we discuss in the first section, a main contribution of threads in distributed systems is that they allow clients and servers to be constructed such that communication and local processing can overlap, resulting in a high level of performance.

As we argued in Chap. 1, client-server organizations are important in distributed systems. In this chapter, we take a closer look at typical organizations of both clients and servers. We also pay attention to general design issues for servers. In addition, we consider general-purpose object servers, which form the basic means for implementing distributed objects.

An important issue, especially in wide-area distributed systems, is moving processes between different machines. Process migration or more specifically, node migration, can help in achieving scalability, but can also help to dynamically...