5

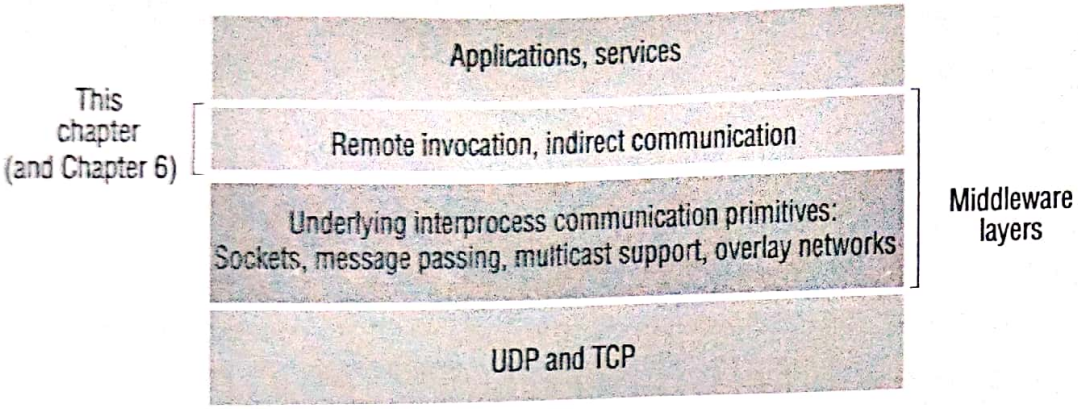
REMOTE INVOCATION

This chapter steps through the remote invocation paradigms introduced in Chapter 2 (indirect communication techniques are addressed in Chapter 6). The chapter starts by examining the most primitive service, request-reply communication, which represents relatively minor enhancements to the underlying interprocess communication primitives discussed in Chapter 4. The chapter then continues by examining the two most prominent remote invocation techniques for communication in distributed systems:

- The remote procedure call (RPC) approach extends the common programming abstraction of the procedure call to distributed environments, allowing a calling process to call a procedure in a remote node as if it is local.
- Remote method invocation (RMI) is similar to RPC but for distributed objects, with added benefits in terms of using object-oriented programming concepts in distributed systems and also extending the concept of an object reference to the global distributed environments, and allowing the use of object references as parameters in remote invocations.

The chapter also features Java RMI as a case study of the remote method invocation approach (further insight can also be gained in Chapter 8, where we look at CORBA).

Figure 5.1 Middleware layers



5.1 Introduction

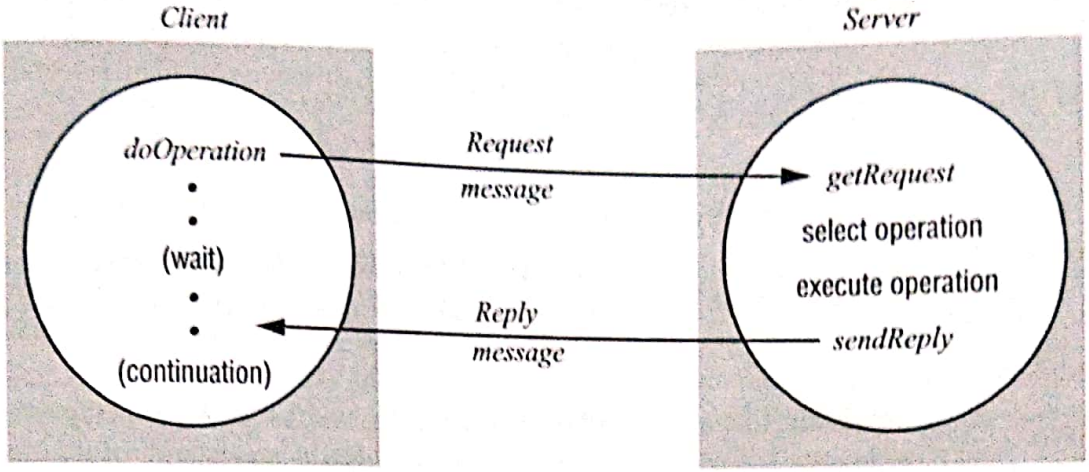
This chapter is concerned with how processes (or entities at a higher level of abstraction such as objects or services) communicate in a distributed system, examining, in particular, the remote invocation paradigms defined in Chapter 2:

- *Request-reply protocols* represent a pattern on top of message passing and support the two-way exchange of messages as encountered in client-server computing. In particular, such protocols provide relatively low-level support for requesting the execution of a remote operation, and also provide direct support for RPC and RMI, discussed below.
- The earliest and perhaps the best-known example of a more programmer-friendly model was the extension of the conventional procedure call model to distributed systems (the *remote procedure call*, or RPC, model), which allows client programs to call procedures transparently in server programs running in separate processes and generally in different computers from the client.
- In the 1990s, the object-based programming model was extended to allow objects in different processes to communicate with one another by means of *remote method invocation* (RMI). RMI is an extension of local method invocation that allows an object living in one process to invoke the methods of an object living in another process.

Note that we use the term 'RMI' to refer to remote method invocation in a generic way – this should not be confused with particular examples of remote method invocation such as Java RMI.

Returning to the diagram first introduced in Chapter 4 (and reproduced in Figure 5.1), this chapter, together with Chapter 6, continues our study of middleware concepts by focusing on the layer above interprocess communication. In particular, Sections 5.2 through 5.4 focus on the styles of communication listed above, with Section 5.5 providing a more complex case study, Java RMI.

Figure 5.2 Request-reply communication



5.2 Request-reply protocols

This form of communication is designed to support the roles and message exchanges in typical client-server interactions. In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server. It can also be reliable because the reply from the server is effectively an acknowledgement to the client. Asynchronous request-reply communication is an alternative that may be useful in situations where clients can afford to retrieve replies later – see Section 7.5.2.

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

- Acknowledgements are redundant, since requests are followed by replies.
- Establishing a connection involves two extra pairs of messages in addition to the pair required for a request and a reply.
- Flow control is redundant for the majority of invocations, which pass only small arguments and results.

The request-reply protocol • The protocol we describe here is based on a trio of communication primitives, *doOperation*, *getRequest* and *sendReply*, as shown in Figure 5.2. This request-reply protocol matches requests to replies. It may be designed to provide certain delivery guarantees. If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message. Figure 5.3 outlines the three communication primitives.

The *doOperation* method is used by clients to invoke remote operations. Its arguments specify the remote server and which operation to invoke, together with additional information (arguments) required by the operation. Its result is a byte array containing the reply. It is assumed that the client calling *doOperation* marshals the

Figure 5.3 Operations of the request-reply protocol

public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)
Sends a request message to the remote server and returns the reply.
The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

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

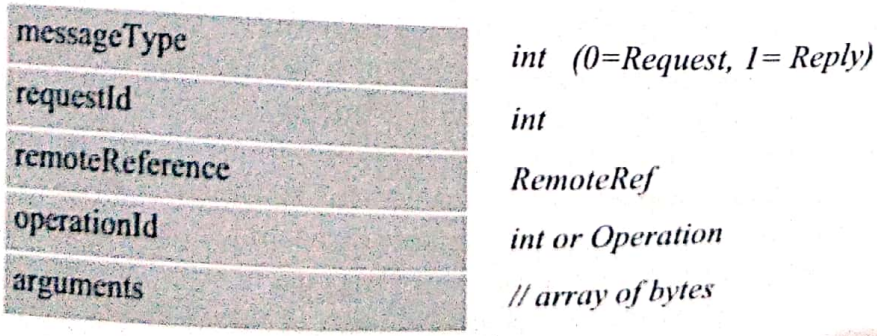
public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
Sends the reply message *reply* to the client at its Internet address and port.

arguments into an array of bytes and unmarshals the results from the array of bytes that is returned. The first argument of *doOperation* is an instance of the class *RemoteRef*, which represents references for remote servers. This class provides methods for getting the Internet address and port of the associated server. The *doOperation* method sends a request message to the server whose Internet address and port are specified in the remote reference given as an argument. After sending the request message, *doOperation* invokes *receive* to get a reply message, from which it extracts the result and returns it to the caller. The caller of *doOperation* is blocked until the server performs the requested operation and transmits a reply message to the client process.

getRequest is used by a server process to acquire service requests, as shown in Figure 5.3. When the server has invoked the specified operation, it then uses *sendReply* to send the reply message to the client. When the reply message is received by the client the original *doOperation* is unblocked and execution of the client program continues.

The information to be transmitted in a request message or a reply message is shown in Figure 5.4. The first field indicates whether the message is a *Request* or a *Reply* message. The second field, *requestId,* contains a message identifier. A *doOperation* in the client generates a *requestId* for each request message, and the server copies these IDs into the corresponding reply messages. This enables *doOperation* to check that a reply message is the result of the current request, not a delayed earlier call. The third field is a remote reference. The fourth field is an identifier for the operation to be invoked. For example, the operations in an interface might be numbered 1, 2, 3, ... , if the client and server use a common language that supports reflection, a representation of the operation itself may be put in this field.

Figure 5.4 Request-reply message structure



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

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

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

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

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

- They suffer from omission failures.
- Messages are not guaranteed to be delivered in sender order.

In addition, the protocol can suffer from the failure of processes (see Section 2.4.2). We assume that processes have crash failures. That is, when they halt, they remain halted – they do not produce Byzantine behaviour.

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

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

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

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

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

As clients can make only one request at a time, the server can interpret each request as an acknowledgement of its previous reply. Therefore the history need contain only the last reply message sent to each client. However, the volume of reply messages in a server's history may still be a problem when it has a large number of clients. This is compounded by the fact that, when a client process terminates, it does not acknowledge the last reply it has received – messages in the history are therefore normally discarded after a limited period of time.

Styles of exchange protocols • Three protocols, that produce differing behaviours in the presence of communication failures are used for implementing various types of request behaviour. They were originally identified by Spector [1982]:

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

The messages passed in these protocols are summarized in Figure 5.5. In the R protocol, a single *Request* message is sent by the client to the server. The R protocol may be used when there is no value to be returned from the remote operation and the client requires no confirmation that the operation has been executed. The client may proceed immediately after the request message is sent as there is no need to wait for a reply message. This protocol is implemented over UDP datagrams and therefore suffers from the same communication failures.

The RR protocol is useful for most client-server exchanges because it is based on the request-reply protocol. Special acknowledgement messages are not required, because a server's reply message is regarded as an acknowledgement of the client's request message. Similarly, a subsequent call from a client may be regarded as an acknowledgement of a server's reply message. As we have seen, communication

Figure 5.5 RPC exchange protocols

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

failures due to UDP datagrams being lost may be masked by the retransmission of requests with duplicate filtering and the saving of replies in a history for retransmission.

The RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply. The *Acknowledge reply* message contains the *requestId* from the reply message being acknowledged. This will enable the server to discard entries from its history. The arrival of a *requestId* in an acknowledgement message will be interpreted as acknowledging the receipt of all reply messages with lower *requestIds*, so the loss of an acknowledgement message is harmless. Although the exchange involves an additional message, it need not block the client, as the acknowledgement may be transmitted after the reply has been given to the client. However it does use processing and network resources. Exercise 5.10 suggests an optimization to the RRA protocol.

Use of TCP streams to implement the request-reply protocol • Section 4.2.3 mentioned that it is often difficult to decide on an appropriate size for the buffer in which to receive datagrams. In the request-reply protocol, this applies to the buffers used by the server to receive request messages and by the client to receive replies. The limited length of datagrams (usually 8 kilobytes) may not be regarded as adequate for use in transparent RMI or RPC systems, since the arguments or results of procedures may be of any size.

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

Howeever, if the application does not require all of the facilities offered by TCP, a more efficient, specially tailored protocol can be implemented over UDP. For example, Sun NFS does not require support for messages of unlimited size, since it

transmits fixed-size file blocks between client and server. In addition to that, its operations are designed to be idempotent, so it does not matter if operations are executed more than once in order to retransmit lost reply messages, making it unnecessary to maintain a history.

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

- as data – for example the text of an HTML page, an image or the class of an applet;
- as a program – for example, servlets [java.sun.com III], or PHP or Python programs that run on the web server.

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

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

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

Authentication: Credentials and challenges are used to support password-style authentication. On the first attempt to access a password-protected area, the server reply contains a challenge applicable to the resource. Chapter 11 explains challenges. When a client receives a challenge, it gets the user to type a name and password and submits the associated credentials with subsequent requests.

HTTP is implemented over TCP. In the original version of the protocol, each client-server interaction consisted of the following steps:

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

However, establishing and closing a connection for every request-reply exchange is expensive, overloading the server and causing too many messages to be sent over the network. Bearing in mind that browsers generally make multiple requests to the same server – for example, to get the images in a page just supplied – a later version of the protocol (HTTP 1.1, see RFC 2616 [Fielding *et al.* 1999]) uses *persistent connections* – connections that remain open over a series of request-reply exchanges between client

Figure 5.6 HTTP *Request* message

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

and server. A persistent connection can be closed by the client or server at any time by sending an indication to the other participant. Servers will close a persistent connection when it has been idle for a period of time. It is possible that a client may receive a message from the server saying that the connection is closed while it is in the middle of sending another request or requests. In such cases, the browser will resend the requests without user involvement, provided that the operations involved are idempotent. For example, the method *GET* described below is idempotent. Where non-idempotent operations are involved, the browser should consult the user as to what to do next.

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

Data resources are supplied as MIME-like structures in arguments and results. Multipurpose Internet Mail Extensions (MIME), specified in RFC 2045 [Freed and Borenstein 1996], is a standard for sending multipart data containing, for example, text, images and sound in email messages. Data is prefixed with its MIME type so that the recipient will know how to handle it. A MIME type specifies a type and a subtype, for example, *text/plain*, *text/html*, *image/gif* or *image/jpeg*. Clients can also specify the MIME types that they are willing to accept.

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

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

With *GET*, all the information for the request is provided in the URL (see, for example, the query string in Section 1.6).

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

Figure 5.6 HTTP *Request* message

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

and server. A persistent connection can be closed by the client or server at any time by sending an indication to the other participant. Servers will close a persistent connection when it has been idle for a period of time. It is possible that a client may receive a message from the server saying that the connection is closed while it is in the middle of sending another request or requests. In such cases, the browser will resend the requests without user involvement, provided that the operations involved are idempotent. For example, the method *GET* described below is idempotent. Where non-idempotent operations are involved, the browser should consult the user as to what to do next.

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

Data resources are supplied as MIME-like structures in arguments and results. Multipurpose Internet Mail Extensions (MIME), specified in RFC 2045 [Freed and Borenstein 1996], is a standard for sending multipart data containing, for example, text, images and sound in email messages. Data is prefixed with its MIME type so that the recipient will know how to handle it. A MIME type specifies a type and a subtype, for example, *text/plain*, *text/html*, *image/gif* or *image/jpeg*. Clients can also specify the MIME types that they are willing to accept.

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

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

With *GET*, all the information for the request is provided in the URL (see, for example, the query string in Section 1.6).

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

POST: Specifies the URL of a resource (for example a program) that can deal with the data supplied in the body of the request. The processing carried out on the data depends on the function of the program specified in the URL. This method is used when the action may change data on the server. It is designed to deal with:

- providing a block of data to a data-handling process such as a servlet – for example, submitting a web form to buy something from a web site;
- posting a message to a mailing list or updating details of members of the list;
- extending a database with an append operation.

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

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

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

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

The operations *PUT* and *DELETE* are idempotent, but *POST* is not necessarily so because it can change the state of a resource. The others are *safe* operations in that they do not change anything.

The requests described above may be intercepted by a proxy server (see Section 2.3.1). The responses to *GET* and *HEAD* may be cached by proxy servers.

Message contents • The *Request* message specifies the name of a method, the URL of a resource, the protocol version, some headers and an optional message body. Figure 5.6 shows the contents of an HTTP *Request* message whose method is *GET*. When the URL specifies a data resource, the *GET* method does not have a message body.

Requests to proxies need the absolute URL, as shown in Figure 5.6. Requests to origin servers (the origin server is where the resource resides) specify a pathname and give the DNS name of the origin server in a *Host* header field. For example,

GET /index.html HTTP/1.1
Host: www.dcs.qmul.ac.uk

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

A *Reply* message specifies the protocol version, a status code and 'reason', some headers and an optional message body, as shown in Figure 5.7. The status code and reason provide a report on the server's success or otherwise in carrying out the request: the former is a three-digit integer for interpretation by a program, and the latter is a textual phrase that can be understood by a person. The header fields are used to pass additional information about the server or access to the resource. For example, if the request requires authentication, the status of the response indicates this and a header

Figure 5.7 HTTP *Reply* message

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

field contains a challenge. Some status returns have quite complex effects. In particular, a 303 status response tells the browser to look under a different URL, which is supplied in a header field in the reply. It is intended for use in a response from a program activated by a *POST* request when the program needs to redirect the browser to a selected resource.

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

5.3 Remote procedure call

As mentioned in Chapter 2, the concept of a remote procedure call (RPC) represents a major intellectual breakthrough in distributed computing, with the goal of making the programming of distributed systems look similar, if not identical, to conventional programming – that is, achieving a high level of distribution transparency. This unification is achieved in a very simple manner, by extending the abstraction of a procedure call to distributed environments. In particular, in RPC, procedures on remote machines can be called as if they are procedures in the local address space. The underlying RPC system then hides important aspects of distribution, including the encoding and decoding of parameters and results, the passing of messages and the preserving of the required semantics for the procedure call. This concept was first introduced by Birrell and Nelson [1984] and paved the way for many of the developments in distributed systems programming used today.

5.3.1 Design issues for RPC

Before looking at the implementation of RPC systems, we look at three issues that are important in understanding this concept:

- the style of programming promoted by RPC – programming with interfaces;
- the call semantics associated with RPC;
- the key issue of transparency and how it relates to remote procedure calls.

Programming with interfaces • Most modern programming languages provide a means of organizing a program as a set of modules that can communicate with one another. Communication between modules can be by means of procedure calls between modules

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

Interfaces in distributed systems: In a distributed program, the modules can run in separate processes. In the client-server model, in particular, each server provides a set of procedures that are available for use by clients. For example, a file server would provide procedures for reading and writing files. The term *service interface* is used to refer to the specification of the procedures offered by a server, defining the types of the arguments of each of the procedures.

There are a number of benefits to programming with interfaces in distributed systems, stemming from the important separation between interface and implementation:

- As with any form of modular programming, programmers are concerned only with the abstraction offered by the service interface and need not be aware of implementation details.
- Extrapolating to (potentially heterogeneous) distributed systems, programmers also do not need to know the programming language or underlying platform used to implement the service (an important step towards managing heterogeneity in distributed systems).
- This approach provides natural support for software evolution in that implementations can change as long as long as the interface (the external view) remains the same. More correctly, the interface can also change as long as it remains compatible with the original.

The definition of service interfaces is influenced by the distributed nature of the underlying infrastructure:

- It is not possible for a client module running in one process to access the variables in a module in another process. Therefore the service interface cannot specify direct access to variables. Note that CORBA IDL interfaces can specify attributes, which seems to break this rule. However, the attributes are not accessed directly but by means of some getter and setter procedures added automatically to the interface.
- The parameter-passing mechanisms used in local procedure calls – for example, call by value and call by reference, are not suitable when the caller and procedure are in different processes. In particular, call by reference is not supported. Rather, the specification of a procedure in the interface of a module in a distributed program describes the parameters as *input* or *output*, or sometimes both. *Input* parameters are passed to the remote server by sending the values of the arguments in the request message and then supplying them as arguments to the operation to be executed in the server. *Output* parameters are returned in the reply message and are used as the result of the call or to replace the values of the corresponding

Figure 5.8 CORBA IDL example

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

variables in the calling environment. When a parameter is used for both input and output, the value must be transmitted in both the request and reply messages.

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

These constraints have a significant impact on the specification of interface definition languages, as discussed below.

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

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

Figure 5.8 shows a simple example of CORBA IDL. The *Person* structure is the same as the one used to illustrate marshalling in Section 4.3.1. The interface named *PersonList* specifies the methods available for RMI in a remote object that implements that interface. For example, the method *addPerson* specifies its argument as *in*, meaning that it is an *input* argument, and the method *getPerson* that retrieves an instance of *Person* by name specifies its second argument as *out*, meaning that it is an *output* argument.

Figure 5.9 Call semantics

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

The concept of an IDL was initially developed for RPC systems but applies equally to RMI and also web services. Our case studies include:

- Sun XDR as an example of an IDL for RPC (in Section 5.3.3);
- CORBA IDL as an example of an IDL for RMI (in Chapter 8 and also included above);
- the Web Services Description Language (WSDL), which is designed for an Internet-wide RPC supporting web services (see Section 9.3);
- and protocol buffers used at Google for storing and interchanging many kinds of structured information (see Section 21.4.1).

RPC call semantics • Request-reply protocols were discussed in Section 5.2, where we showed that *doOperation* can be implemented in different ways to provide different delivery guarantees. The main choices are:

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

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

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

Combinations of these choices lead to a variety of possible semantics for the reliability of remote invocations as seen by the invoker. Figure 5.9 shows the choices of interest, with corresponding names for the semantics that they produce. Note that for local procedure calls, the semantics are *exactly once*, meaning that every procedure is executed exactly once (except in the case of process failure). The choices of RPC invocation semantics are defined as follows.

Maybe semantics: With *maybe* semantics, the remote procedure call may be executed once or not at all. Maybe semantics arises when no fault-tolerance measures are applied and can suffer from the following types of failure:

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

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

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

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

Section 5.2 defines an *idempotent operation* as one that can be performed repeatedly with the same effect as if it had been performed exactly once. Non-idempotent operations can have the wrong effect if they are performed more than once. For example, an operation to increase a bank balance by $10 should be performed only once; if it were to be repeated, the balance would grow and grow! If the operations in a server can be designed so that all of the procedures in their service interfaces are idempotent operations, then *at-least-once* call semantics may be acceptable.

At-most-once semantics: With *at-most-once* semantics, the caller receives either a result, in which case the caller knows that the procedure was executed exactly once, or an exception informing it that no result was received, in which case the procedure will have been executed either once or not at all. *At-most-once* semantics can be achieved by using all of the fault-tolerance measures outlined in Figure 5.9. As in the previous case, the use of retries masks any omission failures of the request or result messages. This set of fault tolerance measures prevents arbitrary failures by ensuring that for each RPC a procedure is never executed more than once. Sun RPC (discussed in Section 5.3.3) provides at-least-once call semantics.

Transparency • The originators of RPC, Birrell and Nelson [1984], aimed to make remote procedure calls as much like local procedure calls as possible, with no distinction in syntax between a local and a remote procedure call. All the necessary calls to marshalling and message-passing procedures were hidden from the programmer making the call. Although request messages are retransmitted after a timeout, this is transparent to the caller to make the semantics of remote procedure calls like that of local procedure calls.

More precisely, returning to the terminology of Chapter 1, RPC strives to offer at least location and access transparency, hiding the physical location of the (potentially remote) procedure and also accessing local and remote procedures in the same way. Middleware can also offer additional levels of transparency to RPC.

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

The latency of a remote procedure call is several orders of magnitude greater than that of a local one. This suggests that programs that make use of remote calls need to be able to take this factor into account, perhaps by minimizing remote interactions. The designers of Argus [Liskov and Scheifler 1982] suggested that a caller should be able to abort a remote procedure call that is taking too long in such a way that it has no effect on the server. To allow this, the server would need to be able to restore things to how they were before the procedure was called. These issues are discussed in Chapter 16.

Remote procedure calls also require a different style of parameter passing, as discussed above. In particular, RPC does not offer call by reference.

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

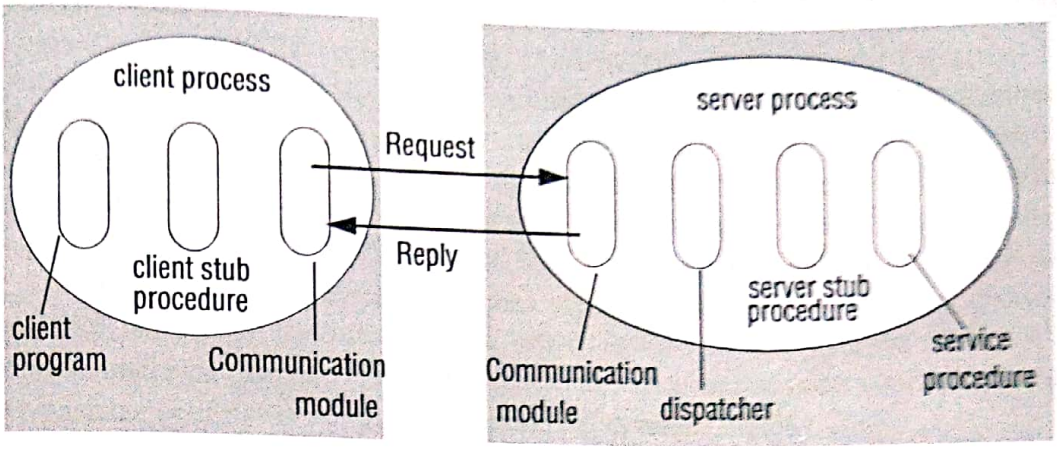
The choice as to whether RPC should be transparent is also available to the designers of IDLs. For example, in some IDLs, a remote invocation may throw an exception when the client is unable to communicate with a remote procedure. This requires that the client program handle such exceptions, allowing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

The current consensus is that remote calls should be made transparent in the sense that the syntax of a remote call is the same as that of a local invocation, but that the difference between local and remote calls should be expressed in their interfaces.

5.3.2 Implementation of RPC

The software components required to implement RPC are shown in Figure 5.10. The client that accesses a service includes one *stub procedure* for each procedure in the service interface. The stub procedure behaves like a local procedure to the client, but instead of executing the call, it marshals the procedure identifier and the arguments into a request message, which it sends via its communication module to the server. When the reply message arrives, it unmarshals the results. The server process contains a dispatcher together with one server stub procedure and one service procedure for each procedure in the service interface. The dispatcher selects one of the server stub procedures according to the procedure identifier in the request message. The server stub procedure

Figure 5.10 Role of client and server stub procedures in RPC



then unmarshals the arguments in the request message, calls the corresponding service procedure and marshals the return values for the reply message. The service procedures implement the procedures in the service interface. The client and server stub procedures and the dispatcher can be generated automatically by an interface compiler from the interface definition of the service.

RPC is generally implemented over a request-reply protocol like the ones discussed in Section 5.2. The contents of request and reply messages are the same as those illustrated for request-reply protocols in Figure 5.4. RPC may be implemented to have one of the choices of invocation semantics discussed in Section 5.3.1 – *at-least-once* or *at-most-once* is generally chosen. To achieve this, the communication module will implement the desired design choices in terms of retransmission of requests, dealing with duplicates and retransmission of results, as shown in Figure 5.9.

5.3.3 Case study: Sun RPC

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

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

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

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

Client and server programs • Further material on Sun RPC is available at www.cdk5.net/rmi. It includes example client and server programs corresponding to the interface defined in Figure 5.11.

5.4 Remote method invocation

Remote method invocation (RMI) is closely related to RPC but extended into the world of distributed objects. In RMI, a calling object can invoke a method in a potentially remote object. As with RPC, the underlying details are generally hidden from the user. The commonalities between RMI and RPC are as follows:

- They both support programming with interfaces, with the resultant benefits that stem from this approach (see Section 5.3.1).
- They are both typically constructed on top of request-reply protocols and can offer a range of call semantics such as *at-least-once* and *at-most-once*.
- They both offer a similar level of transparency – that is, local and remote calls employ the same syntax but remote interfaces typically expose the distributed nature of the underlying call, for example by supporting remote exceptions.

The following differences lead to added expressiveness when it comes to the programming of complex distributed applications and services.

- The programmer is able to use the full expressive power of object-oriented programming in the development of distributed systems software, including the use of objects, classes and inheritance, and can also employ related object-oriented design methodologies and associated tools.
- Building on the concept of object identity in object-oriented systems, all objects in an RMI-based system have unique object references (whether they are local or remote), such object references can also be passed as parameters, thus offering significantly richer parameter-passing semantics than in RPC.

The issue of parameter passing is particularly important in distributed systems. RMI allows the programmer to pass parameters not only by value, as input or output parameters, but also by object reference. Passing references is particularly attractive if the underlying parameter is large or complex. The remote end, on receiving an object reference, can then access this object using remote method invocation, instead of having to transmit the object value across the network.

The rest of this section examines the concept of remote method invocation in more detail, looking initially at the key issues surrounding distributed object models before looking at implementation issues surrounding RMI, including distributed garbage collection.

5.4.1 Design issues for RMI

As mentioned above, RMI shares the same design issues as RPC in terms of programming with interfaces, call semantics and level of transparency. The reader is encouraged to refer back to Section 5.3.1 for discussion of these items.

The key added design issue relates to the object model and, in particular, achieving the transition from objects to distributed objects. We first describe the conventional, single-image object model and then describe the distributed object model.

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

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

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

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

1. The state of the receiver may be changed.
2. A new object may be instantiated, for example, by using a constructor in Java or C++.
3. Further invocations on methods in other objects may take place.

As an invocation can lead to further invocations of methods in other objects, an action is a chain of related method invocations, each of which eventually returns.

Exceptions: Programs can encounter many sorts of errors and unexpected conditions of varying seriousness. During the execution of a method, many different problems may be discovered: for example, inconsistent values in the object's variables, or failures in

attempts to read or write to files or network sockets. When programmers need to insert tests in their code to deal with all possible unusual or erroneous cases, this detracts from the clarity of the normal case. Exceptions provide a clean way to deal with error conditions without complicating the code. In addition, each method heading explicitly lists as exceptions the error conditions it might encounter, allowing users of the method to deal with them. A block of code may be defined to *throw* an exception whenever particular unexpected conditions or errors arise. This means that control passes to another block of code that *catches* the exception. Control does not return to the place where the exception was thrown.

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

Distributed objects • The state of an object consists of the values of its instance variables. In the object-based paradigm the state of a program is partitioned into separate parts, each of which is associated with an object. Since object-based programs are logically partitioned, the physical distribution of objects into different processes or computers in a distributed system is a natural extension (the issue of placement is discussed in Section 2.3.1).

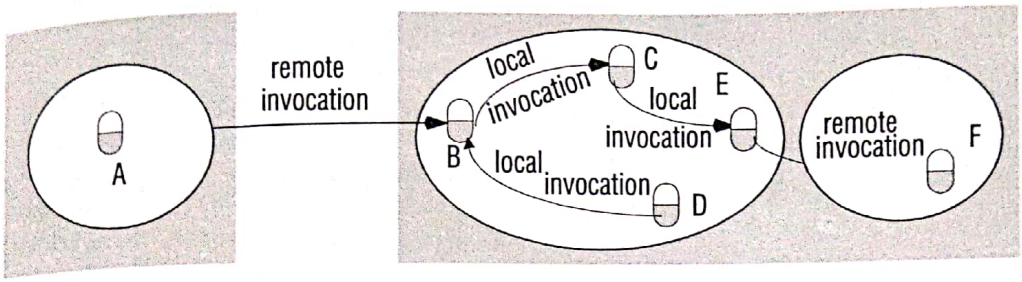
Distributed object systems may adopt the client-server architecture. In this case, objects are managed by servers and their clients invoke their methods using remote method invocation. In RMI, the client's request to invoke a method of an object is sent in a message to the server managing the object.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

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

Having client and server objects in different processes enforces *encapsulation*. That is, the state of an object can be accessed only by the methods of the object, which means that it is not possible for unauthorized methods to act on the state. For example, the possibility of concurrent RMIs from objects in different computers implies that an object may be accessed concurrently. Therefore the possibility of conflicting accesses arises. However, the fact that the data of an object is accessed only by its own methods allows objects to provide methods for protecting themselves against incorrect accesses. For example, they may use synchronization primitives such as condition variables to protect access to their instance variables.

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

Figure 5.12 Remote and local method invocations



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

The distributed object model • This section discusses extensions to the object model to make it applicable to distributed objects. Each process contains a collection of objects, some of which can receive both local and remote invocations, whereas the other objects can receive only local invocations, as shown in Figure 5.12. Method invocations between objects in different processes, whether in the same computer or not, are known as remote method invocations. Method invocations between objects in the same process are local method invocations.

We refer to objects that can receive remote invocations as *remote objects*. In Figure 5.12, the objects B and F are remote objects. All objects can receive local invocations, although they can receive them only from other objects that hold references to them. For example, object C must have a reference to object E so that it can invoke one of its methods. The following two fundamental concepts are at the heart of the distributed object model:

Remote object references: Other objects can invoke the methods of a remote object if they have access to its *remote object reference*. For example, a remote object reference for B in Figure 5.12 must be available to A.

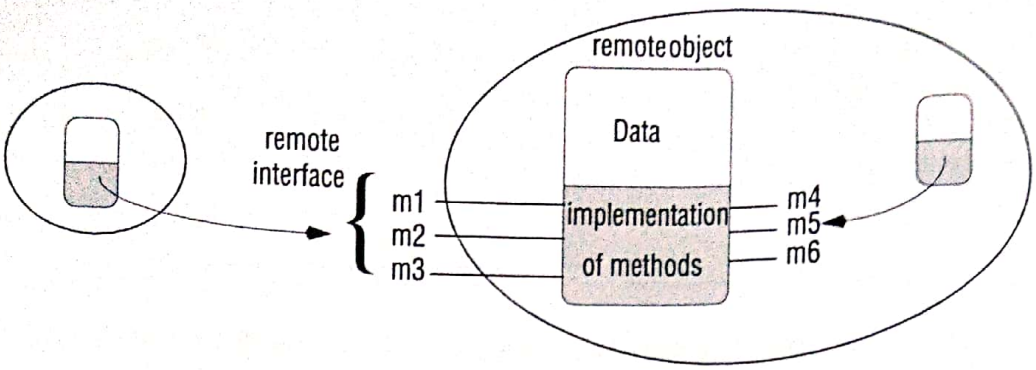
Remote interfaces: Every remote object has a *remote interface* that specifies which of its methods can be invoked remotely. For example, the objects B and F in Figure 5.12 must have remote interfaces.

We look at remote object references, remote interfaces and other aspects of the distributed object model next.

Remote object references: The notion of object reference is extended to allow any object that can receive an RMI to have a remote object reference. A remote object reference is an identifier that can be used throughout a distributed system to refer to a particular unique remote object. Its representation, which is generally different from that of a local object reference is discussed in Section 4.3.4. Remote object references are analogous to local ones in that:

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

Figure 5.13 A remote object and its remote interface



Remote interfaces: The class of a remote object implements the methods of its remote interface, for example as public instance methods in Java. Objects in other processes can invoke only the methods that belong to its remote interface, as shown in Figure 5.13. Local objects can invoke the methods in the remote interface as well as other methods implemented by a remote object. Note that remote interfaces, like all interfaces, do not have constructors.

The CORBA system provides an interface definition language (IDL), which is used for defining remote interfaces. See Figure 5.8 for an example of a remote interface defined in CORBA IDL. The classes of remote objects and the client programs may be implemented in any language for which an IDL compiler is available, such as C++, Java or Python. CORBA clients need not use the same language as the remote object in order to invoke its methods remotely.

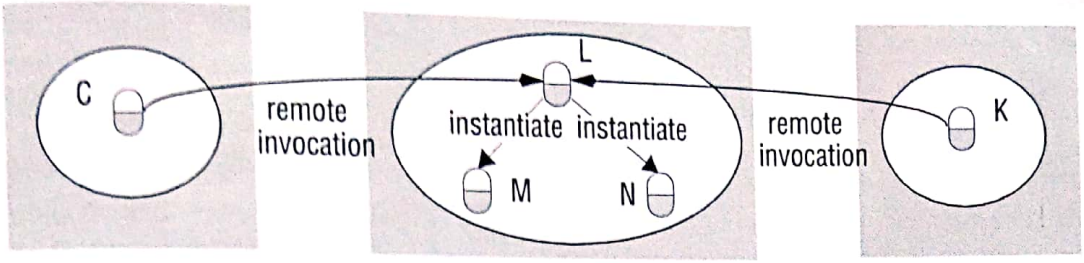
In Java RMI, remote interfaces are defined in the same way as any other Java interface. They acquire their ability to be remote interfaces by extending an interface named *Remote*. Both CORBA IDL (Chapter 8) and Java support multiple inheritance of interfaces. That is, an interface is allowed to extend one or more other interfaces.

Actions in a distributed object system • As in the non-distributed case, an action is initiated by a method invocation, which may result in further invocations on methods in other objects. But in the distributed case, the objects involved in a chain of related invocations may be located in different processes or different computers. When an invocation crosses the boundary of a process or computer, RMI is used, and the remote reference of the object must be available to the invoker. In Figure 5.12, object A needs to hold a remote object reference to object B. Remote object references may be obtained as the results of remote method invocations. For example, object A in Figure 5.12 might obtain a remote reference to object F from object B.

When an action leads to the instantiation of a new object, that object will normally live within the process where instantiation is requested – for example, where the constructor was used. If the newly instantiated object has a remote interface, it will be a remote object with a remote object reference.

Distributed applications may provide remote objects with methods for instantiating objects that can be accessed by RMI, thus effectively providing the effect of remote instantiation of objects. For example, if the object L in Figure 5.14 contains a method for creating remote objects, then the remote invocations from C and K could lead to the instantiation of the objects M and N, respectively.

Figure 5.14 Instantiation of remote objects



Garbage collection in a distributed-object system: If a language, for example Java, supports garbage collection, then any associated RMI system should allow garbage collection of remote objects. Distributed garbage collection is generally achieved by cooperation between the existing local garbage collector and an added module that carries out a form of distributed garbage collection, usually based on reference counting. Section 5.4.3 describes such a scheme in detail. If garbage collection is not available, then remote objects that are no longer required should be deleted.

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

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

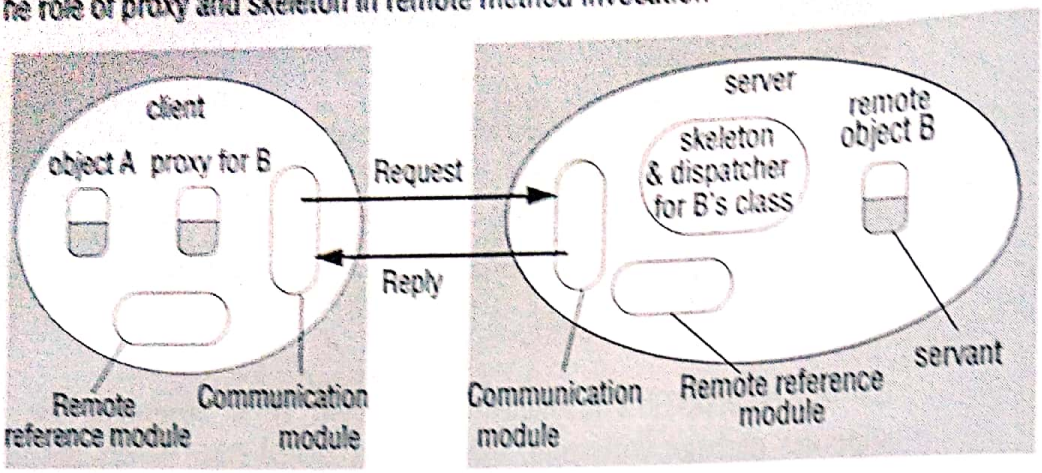
5.4.2 Implementation of RMI

Several separate objects and modules are involved in achieving a remote method invocation. These are shown in Figure 5.15, in which an application-level object A invokes a method in a remote application-level object B for which it holds a remote object reference. This section discusses the roles of each of the components shown in that figure, dealing first with the communication and remote reference modules and then with the RMI software that runs over them.

We then explore the following related topics: the generation of proxies, the binding of names to their remote object references, the activation and passivation of objects and the location of objects from their remote object references.

Communication module • The two cooperating communication modules carry out the request-reply protocol, which transmits *request* and *reply* messages between the client and server. The contents of *request* and *reply* messages are shown in Figure 5.4. The communication module uses only the first three items, which specify the message type, its *requestId* and the remote reference of the object to be invoked. The *operationId* and

Figure 5.15 The role of proxy and skeleton in remote method invocation



all the marshalling and unmarshalling are the concern of the RMI software, discussed below. The communication modules are together responsible for providing a specified invocation semantics, for example *at-most-once*.

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

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

- An entry for all the remote objects held by the process. For example, in Figure 5.15 the remote object B will be recorded in the table at the server.
- An entry for each local proxy. For example, in Figure 5.15 the proxy for B will be recorded in the table at the client.

The role of a proxy is discussed in the subsection on RMI software. The actions of the remote reference module are as follows:

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

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

Servants • A *servant* is an instance of a class that provides the body of a remote object. It is the servant that eventually handles the remote requests passed on by the corresponding skeleton. Servants live within a server process. They are created when remote objects are instantiated and remain in use until they are no longer needed, finally being garbage collected or deleted.

The RMI software • This consists of a layer of software between the application-level objects and the communication and remote reference modules. The roles of the middleware objects shown in Figure 5.15 are as follows:

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

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

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

Remote object references are marshalled in the form shown in Figure 4.13, which includes information about the remote interface of the remote object – for example, the name of the remote interface or the class of the remote object. This information enables the proxy class to be determined so that a new proxy may be created when it is needed. For example, the proxy class name may be generated by appending *_proxy* to the name of the remote interface.

Generation of the classes for proxies, dispatchers and skeletons • The classes for the proxy, dispatcher and skeleton used in RMI are generated automatically by an interface compiler. For example, in the Orbix implementation of CORBA, interfaces of remote objects are defined in CORBA IDL, and the interface compiler can be used to generate the classes for proxies, dispatchers and skeletons in C++ or in Java [www.iona.com]. For Java RMI, the set of methods offered by a remote object is defined as a Java interface

that is implemented within the class of the remote object. The Java RMI compiler generates the proxy, dispatcher and skeleton classes from the class of the remote object.

Dynamic invocation: An alternative to proxies • The proxy just described is static, in the sense that its class is generated from an interface definition and then compiled into the client code. Sometimes this is not practical, though. Suppose that a client program receives a remote reference to an object whose remote interface was not available at compile time. In this case it needs another way to invoke the remote object. *Dynamic invocation* gives the client access to a generic representation of a remote invocation like the *doOperation* method used in Exercise 5.18, which is available as part of the infrastructure for RMI (see Section 5.4.1). The client will supply the remote object reference, the name of the method and the arguments to *doOperation* and then wait to receive the results.

Note that although the remote object reference includes information about the interface of the remote object, such as its name, this is not enough – the names of the methods and the types of the arguments are required for making a dynamic invocation. CORBA provides this information via a component called the Interface Repository, which is described in Chapter 8.

The dynamic invocation interface is not as convenient to use as a proxy, but it is useful in applications where some of the interfaces of the remote objects cannot be predicted at design time. An example of such an application is the shared whiteboard that we use to illustrate Java RMI (Section 5.5), CORBA (Chapter 8) and web services (Section 9.2.3). To summarize: the shared whiteboard application displays many different types of shapes, such as circles, rectangles and lines, but it should also be able to display new shapes that were not predicted when the client was compiled. A client that uses dynamic invocation is able to address this challenge. We shall see in Section 5.5 that the dynamic downloading of classes to clients is an alternative to dynamic invocation. This is available in Java RMI – a single-language system.

Dynamic skeletons: It is clear, from the above example, that it can also arise that a server will need to host remote objects whose interfaces were not known at compile time. For example, a client may supply a new type of shape to the shared whiteboard server for it to store. A server with dynamic skeletons would be able to deal with this situation. We defer describing dynamic skeletons until the chapter on CORBA (Chapter 8). However, as we shall see in Section 5.5, Java RMI addresses this problem by using a generic dispatcher and the dynamic downloading of classes to the server.

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

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

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

The binder • Client programs generally require a means of obtaining a remote object reference for at least one of the remote objects held by a server. For example, in Figure 5.12, object A would require a remote object reference for object B. A *binder* in a distributed system is a separate service that maintains a table containing mappings from textual names to remote object references. It is used by servers to register their remote objects by name and by clients to look them up. Chapter 8 contains a discussion of the CORBA Naming Service. The Java binder, RMIregistry, is discussed briefly in the case study on Java RMI in Section 5.5.

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

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

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

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

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

An *activator* is responsible for:

- registering passive objects that are available for activation, which involves recording the names of servers against the URLs or file names of the corresponding passive objects;

- starting named server processes and activating remote objects in them;
- keeping track of the locations of the servers for remote objects that it has already activated.

Java RMI provides the ability to make some remote objects *activatable* [java.sun.com IX]. When an activatable object is invoked, if that object is not currently active, the object is made active from its marshalled state and then passed the invocation. It uses one activator on each server computer.

The CORBA case study in Chapter 8 describes the implementation repository – a weak form of activator that starts services containing objects in an initial state.

Persistent object stores • An object that is guaranteed to live between activations of processes is called a *persistent object*. Persistent objects are generally managed by persistent object stores, which store their state in a marshalled form on disk. Examples include the CORBA persistent state service (see Chapter 8), Java Data Objects [java.sun.com VIII] and Persistent Java [Jordan 1996; java.sun.com IV].

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

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

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

- The persistent object store maintains some persistent roots, and any object that is reachable from a persistent root is defined to be persistent. This approach is used by Persistent Java, Java Data Objects and PerDiS [Ferreira *et al.* 2000]. They make use of a garbage collector to dispose of objects that are no longer reachable from the persistent roots.
- The persistent object store provides some classes on which persistence is based – persistent objects belong to their subclasses. For example, in Arjuna [Parrington *et al.* 1995], persistent objects are based on C++ classes that provide transactions and recovery. Unwanted objects must be deleted explicitly.

Some persistent object stores, such as PerDiS and Khazana [Carter *et al.* 1998], allow objects to be activated in multiple caches local to users, instead of in servers. In this case, a cache consistency protocol is required. Further details on consistency models can be found on the companion web site, in the chapter from the fourth edition on distributed shared memory [www.cdk5.net/dsm].

Object location • Section 4.3.4 describes a form of remote object reference that contains the Internet address and port number of the process that created the remote object as a way of guaranteeing uniqueness. This form of remote object reference can also be used as an address for a remote object, so long as that object remains in the same process for the rest of its life. But some remote objects will exist in a series of different processes, possibly on different computers, throughout their lifetime. In this case, a remote object reference cannot act as an address. Clients making invocations require both a remote object reference and an address to which to send invocations.

A *location service* helps clients to locate remote objects from their remote object references. It uses a database that maps remote object references to their probable current locations – the locations are probable because an object may have migrated again since it was last heard of. For example, the Clouds system [Dasgupta *et al.* 1991] and the Emerald system [Jul *et al.* 1988] used a cache/broadcast scheme in which a member of a location service on each computer holds a small cache of remote object reference-to-location mappings. If a remote object reference is in the cache, that address is tried for the invocation and will fail if the object has moved. To locate an object that has moved or whose location is not in the cache, the system broadcasts a request. This scheme may be enhanced by the use of forward location pointers, which contain hints as to the new location of an object. Another example is the resolution service required for resolving the URN of a resource into its current URL, mentioned in Section 9.1.

5.4.3 Distributed garbage collection

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

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

- Each server process maintains a set of the names of the processes that hold remote object references for each of its remote objects; for example, *B.holders* is the set of client processes (virtual machines) that have proxies for object *B*. (In Figure 5.15, this set will include the client process illustrated.) This set can be held in an additional column in the remote object table.
- When a client *C* first receives a remote reference to a particular remote object, *B*, it makes an *addRef(B)* invocation to the server of that remote object and then creates a proxy; the server adds *C* to *B.holders*.

11

SECURITY

There is a pervasive need for measures to guarantee the privacy, integrity and availability of resources in distributed systems. Security attacks take the forms of eavesdropping, masquerading, tampering and denial of service. Designers of secure distributed systems must cope with exposed service interfaces and insecure networks in an environment where attackers are likely to have knowledge of the algorithms used and to deploy computing resources.

Cryptography provides the basis for the authentication of messages as well as their secrecy and integrity; carefully designed security protocols are required to exploit it. The selection of cryptographic algorithms and the management of keys are critical to the effectiveness, performance and usability of security mechanisms. Public-key cryptography makes it easy to distribute cryptographic keys but its performance is inadequate for the encryption of bulk data. Secret-key cryptography is more suitable for bulk encryption tasks. Hybrid protocols such as Transport Layer Security (TLS) establish a secure channel using public-key cryptography and then use it to exchange secret keys for use in subsequent data exchanges.

Digital information can be signed, producing digital certificates. Certificates enable trust to be established among users and organizations.

The chapter concludes with case studies on the approaches to security system design and the security mechanisms deployed in Kerberos, TLS/SSL and 802.11 WiFi.

11.1 Introduction

In Section 2.4.3 we introduced a simple model for examining the security requirements in distributed systems. We concluded that the need for security mechanisms in distributed systems arises from the desire to share resources. (Resources that are not shared can generally be protected by isolating them from external access.) If we regard shared resources as objects, then the requirement is to protect any processes that encapsulate shared objects and any communication channels that are used to interact with them against all conceivable forms of attack. The model introduced in Section 2.4.3 provides a good starting point for the identification of security requirements. It can be summarized as follows:

- Processes encapsulate resources (both programming language–level objects and system-defined resources) and allow clients to access them through interfaces. Principals (users or other processes) are authorized to operate on resources. Resources must be protected against unauthorized access (Figure 2.17).
- Processes interact through a network that is shared by many users. Enemies (attackers) can access the network. They can copy or attempt to read any message transmitted through the network and they can inject arbitrary messages, addressed to any destination and purporting to come from any source, into the network (Figure 2.18).

The need to protect the integrity and privacy of information and other resources belonging to individuals and organizations is pervasive in both the physical and the digital world. It arises from the desire to share resources. In the physical world, organizations adopt *security policies* that provide for the sharing of resources within specified limits. For example, a company may permit entry to its buildings only to its employees and accredited visitors. A security policy for documents may specify groups of employees who can access classes of documents, or it may be defined for individual documents and users.

Security policies are enforced with the help of *security mechanisms*. For example, access to a building may be controlled by a reception clerk, who issues badges to accredited visitors, and enforced by a security guard or by electronic door locks. Access to paper documents is usually controlled by concealment and restricted distribution. In the electronic world, the distinction between security policies and mechanisms is equally important; without it, it would be difficult to determine whether a particular system was secure. Security policies are independent of the technology used, just as the provision of a lock on a door does not ensure the security of a building unless there is a policy for its use (for example, that the door will be locked whenever nobody is guarding the entrance). The security mechanisms that we describe here do not in themselves ensure the security of a system. In Section 11.1.2, we outline the requirements for security in various simple electronic commerce scenarios, illustrating the need for policies in that context.

The provision of mechanisms for the protection of data and other resources in distributed systems while allowing interactions between computers that are permitted by security policies is the concern of this chapter. The mechanisms that we shall describe are designed to enforce security policies against the most determined attacks.

The role of cryptography • Digital cryptography provides the basis for most computer security mechanisms, but it is important to note that computer security and cryptography are distinct subjects. Cryptography is the art of encoding information in a format that only the intended recipients can decode. Cryptography can also be employed to provide proof of the authenticity of information, in a manner analogous to the use of signatures in conventional transactions.

Cryptography has a long and fascinating history. The military need for secure communication and the corresponding need of an enemy to intercept and decrypt it led to the investment of much intellectual effort by some of the best mathematical brains of their time. Readers interested in exploring this history will find absorbing reading in books on the topic by David Kahn [Kahn 1967, 1983, 1991] and Simon Singh [Singh 1999]. Whitfield Diffie, one of the inventors of public-key cryptography, has written with firsthand knowledge on the recent history and politics of cryptography [Diffie 1988, Diffie and Landau 1998].

It is only in recent times that cryptography has emerged from the wraps previously placed on it by the political and military establishments that used to control its development and use. It is now the subject of open research by a large and active community, with the results presented in many books, journals and conferences. The publication of Schneier's book *Applied Cryptography* [Schneier 1996] was a milestone in the opening up of knowledge in the field. It was the first book to publish many important algorithms with source code – a courageous step, because when the first edition appeared in 1994 the legal status of such publication was unclear. Schneier's book remains the definitive reference on most aspects of modern cryptography. A more recent book co-authored by Schneier [Ferguson and Schneier 2003] provides an excellent introduction to computer cryptography and a discursive overview of virtually all the important algorithms and techniques in current use, including several published since Schneier's earlier book. In addition, Menezes *et al.* [1997] provide a good practical handbook with a strong theoretical basis and the Network Security Library [www.secinf.net] is an excellent online source of practical knowledge and experience.

Ross Anderson's *Security Engineering* [Anderson 2008] is also outstanding. It is replete with object lessons on the design of secure systems, drawn from real-world situations and system security failures.

The new openness is largely a result of the tremendous growth of interest in non-military applications of cryptography and the security requirements of distributed computer systems, which has led to the existence for the first time of a self-sustaining community of cryptographic researchers outside the military domain.

Ironically, the opening of cryptography to public access and use has resulted in a great improvement in cryptographic techniques, both in their strength to withstand attacks by enemies and in the convenience with which they can be deployed. Public-key cryptography is one of the fruits of this openness. As another example, we note that the DES encryption algorithm that was adopted and used by the US military and government agencies was initially a military secret. Its eventual publication and successful efforts to crack it resulted in the development of much stronger secret-key encryption algorithms.

Another useful spin-off has been the development of a common terminology and approach. An example of the latter is the adoption of a set of familiar names for protagonists (principals) involved in the transactions that are to be secured. The use of

Figure 11.1 Familiar names for the protagonists in security protocols

Alice	First participant
Bob	Second participant
Carol	Participant in three- and four-party protocols
Dave	Participant in four-party protocols
Eve	Eavesdropper
Mallory	Malicious attacker
Sara	A server

familiar names for principals and attackers helps to clarify and bring to life descriptions of security protocols and potential attacks on them, which aids in identifying their weaknesses. The names shown in Figure 11.1 are used extensively in the security literature and we use them freely here. We have not been able to discover their origins; the earliest occurrence of which we are aware is in the original RSA public-key cryptography paper [Rivest *et al.* 1978]. An amusing commentary on their use can be found in Gordon [1984].

11.1.1 Threats and attacks

Some threats are obvious – for example, in most types of local network it is easy to construct and run a program on a connected computer that obtains copies of the messages transmitted between other computers. Other threats are more subtle – if clients fail to authenticate servers, a program might install itself in place of an authentic file server and thereby obtain copies of confidential information that clients unwittingly send to it for storage.

In addition to the danger of loss or damage to information or resources through direct violations, fraudulent claims may be made against the owner of a system that is not demonstrably secure. To avoid such claims, the owner must be in a position to disprove the claim by showing that the system is secure against such violations or by producing a log of all of the transactions for the period in question. A common instance is the 'phantom withdrawal' problem in automatic cash dispensers (teller machines). The best answer that a bank can supply to such a claim is to provide a record of the transaction that is digitally signed by the account holder in a manner that cannot be forged by a third party.

The main goal of security is to restrict access to information and resources to just those principals that are authorized to have access. Security threats fall into three broad classes:

Leakage: Refers to the acquisition of information by unauthorized recipients.

Tampering: Refers to the unauthorized alteration of information.

Vandalism: Refers to interference with the proper operation of a system without gain to the perpetrator.

Attacks on distributed systems depend upon obtaining access to existing communication channels or establishing new channels that masquerade as authorized connections. (We use the term *channel* to refer to any communication mechanism between processes.) Methods of attack can be further classified according to the way in which a channel is misused:

Eavesdropping: Obtaining copies of messages without authority.

Masquerading: Sending or receiving messages using the identity of another principal without their authority.

Message tampering: Intercepting messages and altering their contents before passing them on to the intended recipient. The *man-in-the-middle attack* is a form of message tampering in which an attacker intercepts the very first message in an exchange of encryption keys to establish a secure channel. The attacker substitutes compromised keys that enable them to decrypt subsequent messages before re-encrypting them in the correct keys and passing them on.

Replaying: Storing intercepted messages and sending them at a later date. This attack may be effective even with authenticated and encrypted messages.

Denial of service: Flooding a channel or other resource with messages in order to deny access for others.

These are the dangers in theory, but how are attacks carried out in practice? Successful attacks depend upon the discovery of loopholes in the security of systems. Unfortunately, these are all too common in today's systems, and they are not necessarily particularly obscure. Cheswick and Bellovin [1994] identify 42 weaknesses that they regard as posing serious risks in widely used Internet systems and components. They range from password guessing to attacks on the programs that perform the Network Time Protocol or handle mail transmission. Some of these have led to successful and well-publicized attacks [Stoll 1989, Spafford 1989], and many of them have been exploited for mischievous or criminal purposes.

When the Internet and the systems that are connected to it were designed, security was not a priority. The designers probably had no conception of the scale to which the Internet would grow, and the basic design of systems such as UNIX predates the advent of computer networks. As we shall see, the incorporation of security measures needs to be carefully thought out at the basic design stage, and the material in this chapter is intended to provide the basis for such thinking.

We focus on the threats to distributed systems that arise from the exposure of their communication channels and their interfaces. For many systems, these are the only threats that need to be considered (other than those that arise from human error – security mechanisms cannot guard against a badly chosen password or one that is carelessly disclosed). But for systems that include mobile programs and systems whose security is particularly sensitive to information leakage, there are further threats.

Threats from mobile code • Several recently developed programming languages have been designed to enable programs to be loaded into a process from a remote server and then executed locally. In that case, the internal interfaces and objects within an executing process may be exposed to attack by mobile code.

Java is the most widely used language of this type, and its designers paid considerable attention to the design and construction of the language and the mechanisms for remote loading in an effort to restrict the exposure (the *sandbox* model of protection against mobile code).

The Java virtual machine (JVM) is designed with mobile code in view. It gives each application its own environment in which to run. Each environment has a security manager that determines which resources are available to the application. For example, the security manager might stop an application reading and writing files or give it limited access to network connections. Once a security manager has been set, it cannot be replaced. When a user runs a program such as a browser that downloads mobile code to be run locally on their behalf, they have no very good reason to trust the code to behave in a responsible manner. In fact, there is a danger of downloading and running malicious code that removes files or accesses private information. To protect users against untrusted code, most browsers specify that applets cannot access local files, printers or network sockets. Some applications of mobile code are able to assume various levels of trust in downloaded code. In this case, the security managers are configured to provide more access to local resources.

The JVM takes two further measures to protect the local environment:

1. The downloaded classes are stored separately from the local classes, preventing them from replacing local classes with spurious versions.

2. The bytecodes are checked for validity. Valid Java bytecode is composed of Java virtual machine instructions from a specified set. The instructions are also checked to ensure that they will not produce certain errors when the program runs, such as accessing illegal memory addresses.

The security of Java has been the subject of much subsequent investigation, in the course of which it became clear that the original mechanisms adopted were not free of loopholes [McGraw and Felden 1999]. The identified loopholes were corrected and the Java protection system was refined to allow mobile code to access local resources when authorized to do so [java.sun.com V].

Despite the inclusion of type-checking and code-validation mechanisms, the security mechanisms incorporated into mobile code systems do not yet produce the same level of confidence in their effectiveness as those used to protect communication channels and interfaces. This is because the construction of an environment for execution of programs offers many opportunities for error, and it is difficult to be confident that all have been avoided. Volpano and Smith [1999] have pointed out that an alternative approach, based on proofs that the behaviour of mobile code is sound, might offer a better solution.

Information leakage • If the transmission of a message between two processes can be observed, some information can be gleaned from its mere existence – for example, a flood of messages to a dealer in a particular stock might indicate a high level of trading in that stock. There are many more subtle forms of information leakage, some malicious and others arising from inadvertent error. The potential for leakage arises whenever the results of a computation can be observed. Work was done on the prevention of this type of security threat in the 1970s [Denning and Denning 1977]. The approach taken is to assign security levels to information and channels and to analyze the flow of information

into channels with the aim of ensuring that high-level information cannot flow into lower-level channels. A method for the secure control of information flows was first described by Bell and LaPadula [1975]. The extension of this approach to distributed systems with mutual distrust between components is the subject of recent research [Myers and Liskov 1997].

11.1.2 Securing electronic transactions

Many uses of the Internet in industry, commerce and elsewhere involve transactions that depend crucially on security. For example:

Email: Although email systems did not originally include support for security, there are many uses of email in which the contents of messages must be kept secret (for example, when sending a credit card number) or the contents and sender of a message must be authenticated (for example when submitting an auction bid by email). Cryptographic security based on the techniques described in this chapter is now included in many mail clients.

Purchase of goods and services: Such transactions are now commonplace. Buyers select goods and pay for them via the Web and the goods are delivered through an appropriate delivery mechanism. Software and other digital products (such as recordings and videos) can be delivered by downloading. Tangible goods such as books, CDs and almost every other type of product are also sold by Internet vendors; these are supplied via a delivery service.

Banking transactions: Electronic banks now offer users virtually all of the facilities provided by conventional banks. They can check their balances and statements, transfer money between accounts, set up regular automatic payments and so on.

Micro-transactions: The Internet lends itself to the supply of small quantities of information and other services to many customers. Most web pages currently can be viewed without charge, but the development of the Web as a high-quality publishing medium surely depends upon the ability of information providers to obtain payments from consumers of the information. Voice and videoconferencing on the Internet is currently also free, but it is charged for when a telephone network is also involved. The price for such services may amount to only a fraction of a cent, and the payment overheads must be correspondingly low. In general, schemes based on the involvement of a bank or credit card server for each transaction cannot achieve this.

Transactions such as these can be safely performed only when they are protected by appropriate security policies and mechanisms. A purchaser must be protected against the disclosure of credit codes (card numbers) during transmission and against fraudulent vendors who obtain payment with no intention of supplying the goods. Vendors must obtain payment before releasing the goods, and for downloadable products they must ensure that only paying customers obtain the data in a usable form. The required protection must be achieved at a cost that is reasonable in comparison with the value of the transaction.

Sensible security policies for Internet vendors and buyers lead to the following requirements for securing web purchases:

1. Authenticate the vendor to the buyer, so that the buyer can be confident that they are in contact with a server operated by the vendor with whom they intended to deal.
2. Keep the buyer's credit card number and other payment details from falling into the hands of any third party and ensure that they are transmitted unaltered from the buyer to the vendor.
3. If the goods are in a form suitable for downloading, ensure that their content is delivered to the buyer without alteration and without disclosure to third parties.

The identity of the buyer is not normally required by the vendor (except for the purpose of delivering the goods, if they are not downloaded). The vendor will wish to check that the buyer has sufficient funds to pay for the purchase, but this is usually done by demanding payment from the buyer's bank before delivering the goods.

The security needs of banking transactions using an open network are similar to those for purchase transactions, with the buyer as the account holder and the bank as the vendor, but there there is a fourth requirement as well:

4. Authenticate the identity of the account holder to the bank before giving them access to their account.

Note that in this situation, it is important for the bank to ensure that the account holder cannot deny that they participated in a transaction. *Non-repudiation* is the name given to this requirement.

In addition to the above requirements, which are dictated by security policies, there are some system requirements. These arise from the very large scale of the Internet, which makes it impractical to require buyers to enter into special relationships with vendors (by registering encryption keys for later use, etc.). It should be possible for a buyer to complete a secure transaction with a vendor even if there has been no previous contact between buyer and vendor and without the involvement of a third party. Techniques such as the use of 'cookies' – records of previous transactions stored on the user's client host – have obvious security weaknesses; desktop and mobile hosts are often located in insecure physical environments.

Because of the importance of security for Internet commerce and the rapid growth in Internet commerce, we have chosen to illustrate the use of cryptographic security techniques by describing in Section 11.6 the *de facto* standard security protocol used in most electronic commerce – Transport Layer Security (TLS). A description of Millicent, a protocol specifically designed for micro-transactions, can be found at www.cdk5.net/security.

Internet commerce is an important application of security techniques, but it is certainly not the only one. It is needed wherever computers are used by individuals or organizations to store and communicate important information. The use of encrypted email for private communication between individuals is a case in point that has been the subject of considerable political discussion. We refer to this debate in Section 11.5.2.

11.1.3 Designing secure systems

Immense strides have been made in recent years in the development of cryptographic techniques and their application, yet designing secure systems remains an inherently difficult task. At the heart of this dilemma is the fact that the designer's aim is to exclude *all* possible attacks and loopholes. The situation is analogous to that of the programmer whose aim is to exclude all bugs from their program. In neither case is there a concrete method to ensure this goal is met during the design. One designs to the best available standards and applies informal analysis and checks. Once a design is complete, formal validation is an option. Work on the formal validation of security protocols has produced some important results [Lampson *et al.* 1992, Schneider 1996, Abadi and Gordon 1999]. A description of one of the first steps in this direction, the BAN logic of authentication [Burrows *et al.* 1990], and its application can be found at www.cdk5.net/security.

Security is about avoiding disasters and minimizing mishaps. When designing for security it is necessary to assume the worst. The box on page 472 shows a set of useful assumptions and design guidelines. These assumptions underly the thinking behind the techniques that we describe in this chapter.

To demonstrate the validity of the security mechanisms employed in a system, the system's designers must first construct a list of threats – methods by which the security policies might be violated – and show that each of them is prevented by the mechanisms employed. This demonstration may take the form of an informal argument or, better, a logical proof.

No list of threats is likely to be exhaustive, so auditing methods must also be used in security-sensitive applications to detect violations. These are straightforward to implement if a secure log of security-sensitive system actions is always recorded with details of the users performing the actions and their authority.

A security log will contain a sequence of timestamped records of users' actions. At a minimum the records will include the identity of a principal, the operation performed (e.g., delete file, update accounting record), the identity of the object operated on and a timestamp. Where particular violations are suspected, the records may be extended to include physical resource utilization (network bandwidth, peripherals), or the logging process may be targeted at operations on particular objects. Subsequent analysis may be statistical or search-based. Even when no violations are suspected, the statistics may be compared over time to help to discover any unusual trends or events.

The design of secure systems is an exercise in balancing costs against the threats. The range of techniques that can be deployed for protecting processes and securing interprocess communication are strong enough to withstand almost any attack, but their use incurs expense and inconvenience:

- A cost (in terms of computational effort and network usage) is incurred for their use. The costs must be balanced against the threats.
- Inappropriately specified security measures may exclude legitimate users from performing necessary actions.

Such trade-offs are difficult to identify without compromising security and may seem to conflict with the advice in the first paragraph of this subsection, but the strength of security techniques required can be quantified and techniques can be selected based on

the estimated cost of attacks. The relatively low-cost techniques employed in the Millicent protocol, described at www.cdk5.net/security provide an example.

As an illustration of the difficulties and mishaps that can arise in the design of secure systems, we review difficulties that arose with the security design originally incorporated in the IEEE 802.11 WiFi networking standard in Section 11.6.4.

11.2 Overview of security techniques

The purpose of this section is to introduce the reader to some of the more important techniques and mechanisms for securing distributed systems and applications. Here we describe them informally, reserving more rigorous descriptions for Sections 11.3 and 11.4. We use the familiar names for principals introduced in Figure 11.1 and the notations for encrypted and signed items shown in Figure 11.2.

Worst-case assumptions and design guidelines

Interfaces are exposed: Distributed systems are composed of processes that offer services or share information. Their communication interfaces are necessarily open (to allow new clients to access them) – an attacker can send a message to any interface.

Networks are insecure: For example, message sources can be falsified – messages can be made to look as though they came from Alice when they were actually sent by Mallory. Host addresses can be 'spoofed' – Mallory can connect to the network with the same address as Alice and receive copies of messages intended for her.

Limit the lifetime and scope of each secret: When a secret key is first generated we can be confident that it has not been compromised. The longer we use it and the more widely it is known, the greater the risk. The use of secrets such as passwords and shared secret keys should be time-limited, and sharing should be restricted.

Algorithms and program code are available to attackers: The bigger and the more widely distributed a secret is, the greater the risk of its disclosure. Secret encryption algorithms are totally inadequate for today's large-scale network environments. Best practice is to publish the algorithms used for encryption and authentication, relying only on the secrecy of cryptographic keys. This helps to ensure that the algorithms are strong by throwing them open to scrutiny by third parties.

Attackers may have access to large resources: The cost of computing power is rapidly decreasing. We should assume that attackers will have access to the largest and most powerful computers projected in the lifetime of a system, then add a few orders of magnitude to allow for unexpected developments.

Minimize the trusted base: The portions of a system that are responsible for the implementation of its security, *and all the hardware and software components upon which they rely*, have to be trusted – this is often referred to as the *trusted computing base*. Any defect or programming error in this trusted base can produce security weaknesses, so we should aim to minimize its size. For example, application programs should not be trusted to protect data from their users.

Figure 11.2 Cryptography notations

K_A	Alice's secret key
K_B	Bob's secret key
K_{AB}	Secret key shared between Alice and Bob
K_{Apriv}	Alice's private key (known only to Alice)
K_{Apub}	Alice's public key (published by Alice for all to read)
$\{M\}_K$	Message M encrypted with key K
$[M]_K$	Message M signed with key K

11.2.1 Cryptography

Encryption is the process of encoding a message in such a way as to hide its contents. Modern cryptography includes several secure algorithms for encrypting and decrypting messages. They are all based on the use of secrets called *keys*. A cryptographic key is a parameter used in an encryption algorithm in such a way that the encryption cannot be reversed without knowledge of the key.

There are two main classes of encryption algorithm in general use. The first uses *shared secret keys* – the sender and the recipient must share a knowledge of the key and it must not be revealed to anyone else. The second class of encryption algorithms uses *public/private key pairs*. Here the sender of a message uses a *public key* – one that has already been published by the recipient – to encrypt the message. The recipient uses a corresponding *private key* to decrypt the message. Although many principals may examine the public key, only the recipient can decrypt the message, because they have the private key.

Both classes of encryption algorithm are extremely useful and are used widely in the construction of secure distributed systems. Public-key encryption algorithms typically require 100 to 1000 times as much processing power as secret-key algorithms, but there are situations where their convenience outweighs this disadvantage.

11.2.2 Uses of cryptography

Cryptography plays three major roles in the implementation of secure systems. We introduce them here in outline by means of some simple scenarios. In later sections of this chapter, we describe these and other protocols in greater detail, addressing some unresolved problems that are merely highlighted here.

In all of our scenarios below, we can assume that Alice, Bob and any other participants have already agreed about the encryption algorithms that they wish to use and have implementations of them. We also assume that any secret keys or private keys that they hold can be stored securely to prevent attackers obtaining them.

Secrecy and integrity • Cryptography is used to maintain the secrecy and integrity of information whenever it is exposed to potential attacks – for example, during transmission across networks that are vulnerable to eavesdropping and message tampering. This use of cryptography corresponds to its traditional role in military and

intelligence activities. It exploits the fact that a message that is encrypted with a particular encryption key can only be decrypted by a recipient who knows the corresponding decryption key. Thus it maintains the secrecy of the encrypted message as long as the decryption key is not *compromised* (disclosed to non-participants in the communication) and provided that the encryption algorithm is strong enough to defeat any possible attempts to crack it. Encryption also maintains the integrity of the encrypted information, provided that some redundant information such as a checksum is included and checked.

Scenario 1. Secret communication with a shared secret key: Alice wishes to send some information secretly to Bob. Alice and Bob share a secret key K_{AB}.

1. Alice uses K_{AB} and an agreed encryption function $E(K_{AB}, M)$ to encrypt and send any number of messages $\{M_i\}K_{AB}$ to Bob. (Alice can go on using K_{AB} as long as it is safe to assume that K_{AB} has not been compromised.)

2. Bob decrypts the encrypted messages using the corresponding decryption function $D(K_{AB}, M)$.

Bob can now read the original message M. If the decrypted message makes sense, or better, if it includes some value agreed between Alice and Bob (such as a checksum of the message) then Bob knows that the message is from Alice and that it hasn't been tampered with. But there are still some problems:

Problem 1: How can Alice send a shared key K_{AB} to Bob securely?

Problem 2: How does Bob know that any $\{M_i\}$ isn't a copy of an earlier encrypted message from Alice that was captured by Mallory and replayed later? Mallory needn't have the key K_{AB} to carry out this attack – he can simply copy the bit pattern that represents the message and send it to Bob later. For example, if the message is a request to pay some money to someone, Mallory might trick Bob into paying twice.

We show how these problems can be resolved later in this chapter.

Authentication • Cryptography is used in support of mechanisms for authenticating communication between pairs of principals. A principal who decrypts a message successfully using a particular key can assume that the message is authentic if it contains a correct checksum or (if the block-chaining mode of encryption, described in Section 11.3, is used) some other expected value. They can infer that the sender of the message possessed the corresponding encryption key and hence deduce the identity of the sender if the key is known only to two parties. Thus if keys are held in private, a successful decryption authenticates the decrypted message as coming from a particular sender.

Scenario 2. Authenticated communication with a server: Alice wishes to access files held by Bob, a file server on the local network of the organization where she works. Sara is an authentication server that is securely managed. Sara issues users with passwords and holds current secret keys for all of the principals in the system it serves (generated by applying some transformation to the user's password). For example, it knows Alice's key K_A and Bob's K_B. In our scenario we refer to a *ticket*. A ticket is an encrypted item issued by an authentication server, containing the identity of the principal to whom it is issued and a shared key that has been generated for the current communication session.

1. Alice sends an (unencrypted) message to Sara stating her identity and requesting a ticket for access to Bob.

2. Sara sends a response to Alice encrypted in K_A consisting of a ticket (to be sent to Bob with each request for file access) encrypted in K_B and a new secret key K_{AB} for use when communicating with Bob. So the response that Alice receives looks like this: $\{\{Ticket\}_{K_B}, K_{AB}\}_{K_A}$.

3. Alice decrypts the response using K_A (which she generates from her password using the same transformation; the password is not transmitted over the network, and once it has been used it is deleted from local storage to avoid compromising it). If Alice has the correct password-derived key K_A, she obtains a valid ticket for using Bob's service and a new encryption key for use in communicating with Bob. Alice can't decrypt or tamper with the ticket, because it is encrypted in K_B. If the recipient isn't Alice then they won't know Alice's password, so they won't be able to decrypt the message.

4. Alice sends the ticket to Bob together with her identity and a request R to access a file: $\{Ticket\}_{K_B}, Alice, R$.

5. The ticket, originally created by Sara, is actually: $\{K_{AB}, Alice\}_{K_B}$. Bob decrypts the ticket using his key K_B. So Bob gets the authentic identity of Alice (based on the knowledge shared between Alice and Sara of Alice's password) and a new shared secret key K_{AB} for use when interacting with Alice. (This is called a *session key* because it can safely be used by Alice and Bob for a sequence of interactions.)

This scenario is a simplified version of the authentication protocol originally developed by Roger Needham and Michael Schroeder [1978] and subsequently used in the Kerberos system developed and used at MIT [Steiner *et al.* 1988], which is described in Section 11.6.2. In our simplified description of their protocol above there is no protection against the replay of old authentication messages. This and some other weaknesses are dealt with in our description of the full Needham–Schroeder protocol in Section 11.6.1.

The authentication protocol we have described depends upon prior knowledge by the authentication server Sara of Alice's and Bob's keys, K_A and K_B. This is feasible in a single organization where Sara runs on a physically secure computer and is managed by a trusted principal who generates initial values of the keys and transmits them to users by a separate secure channel. But it isn't appropriate for electronic commerce or other wide area applications, where the use of a separate channel is extremely inconvenient and the requirement for a trusted third party is unrealistic. Public-key cryptography rescues us from this dilemma.

The usefulness of challenges: An important aspect of Needham and Schroeder's 1978 breakthrough was the realization that a user's password does not have to be submitted to an authentication service (and hence exposed in the network) each time it is authenticated. Instead, they introduced the concept of a cryptographic *challenge*. This can be seen in step 2 of our scenario above, where the server, Sara, issues a ticket to Alice *encrypted in Alice's secret key*, K_A. This constitutes a challenge because Alice cannot make use of the ticket unless she can decrypt it, and she can only decrypt it if she

can determine K_A, which is derived from Alice's password. An imposter claiming to be Alice would be defeated at this point.

Scenario 3. Authenticated communication with public keys: Assuming that Bob has generated a public/private key pair, the following dialogue enables Bob and Alice to establish a shared secret key, K_{AB}:

1. Alice accesses a key distribution service to obtain a *public-key certificate* giving Bob's public key. It's called a certificate because it is signed by a trusted authority – a person or organization that is widely known to be reliable. After checking the signature, she reads Bob's public key, K_{Bpub}, from the certificate. (We discuss the construction and use of public-key certificates in Section 11.2.3.)

2. Alice creates a new shared key, K_{AB}, and encrypts it using K_{Bpub} with a public-key algorithm. She sends the result to Bob, along with a name that uniquely identifies a public/private key pair (since Bob may have several of them) – that is, Alice sends $keyname, \{K_{AB}\}K_{Bpub}$.

3. Bob selects the corresponding private key, K_{Bpriv}, from his private key store and uses it to decrypt K_{AB}. Note that Alice's message to Bob might have been corrupted or tampered with in transit. The consequence would simply be that Bob and Alice don't share the same key K_{AB}. If this is a problem, it can be circumvented by adding an agreed value or string to the message, such as Bob's and Alice's names or email addresses, which Bob can check after decrypting.

The above scenario illustrates the use of public-key cryptography to distribute a shared secret key. This technique is known as a *hybrid cryptographic protocol* and is very widely used, since it exploits useful features of both public-key and secret-key encryption algorithms.

Problem: This key exchange is vulnerable to man-in-the-middle attacks. Mallory may intercept Alice's initial request to the key distribution service for Bob's public-key certificate and send a response containing his own public key. He can then intercept all the subsequent messages. In our description above, we guard against this attack by requiring Bob's certificate to be signed by a well-known authority. To protect against this attack, Alice must ensure that Bob's public-key certificate is signed with a public key (as described below) that she has received in a totally secure manner.

Digital signatures • Cryptography is used to implement a mechanism known as a *digital signature*. This emulates the role of a conventional signature, verifying to a third party that a message or a document is an unaltered copy of one produced by the signer.

Digital signature techniques are based upon an irreversible binding to the message or document of a secret known only to the signer. This can be achieved by encrypting the message – or better, a compressed form of the message called a *digest* – using a key that is known only to the signer. A digest is a fixed-length value computed by applying a *secure digest function*. A secure digest function is similar to a checksum function, but it is very unlikely to produce a similar digest value for two different messages. The resulting encrypted digest acts as a signature that accompanies the message. Public-key cryptography is generally used for this: the originator generates a signature with their private key, and the signature can be decrypted by any recipient using the corresponding

Figure 11.3 Alice's bank account certificate

1. *Certificate type*:	Account number
2. *Name*:	Alice
3. *Account*:	6262626
4. *Certifying authority*:	Bob's Bank
5. *Signature*:	$\{Digest(field\ 2 + field\ 3)\}_{K_{Bpriv}}$

public key. There is an additional requirement: the verifier should be sure that the public key really is that of the principal claiming to be the signer – this is dealt with by the use of public-key certificates, described in Section 11.2.3.

Scenario 4. Digital signatures with a secure digest function: Alice wants to sign a document M so that any subsequent recipient can verify that she is the originator of it. Thus when Bob later accesses the signed document after receiving it by any route and from any source (for example, it could be sent in a message or it could be retrieved from a database), he can verify that Alice is the originator.

1. Alice computes a fixed-length digest of the document, $Digest(M)$.
2. Alice encrypts the digest in her private key, appends it to M and makes the result, M, $\{Digest(M)\}_{K_{Apriv}}$, available to the intended users.
3. Bob obtains the signed document, extracts M and computes $Digest(M)$.
4. Bob decrypts $\{Digest(M)\}_{K_{Apriv}}$ using Alice's public key, K_{Apub}, and compares the result with his calculated $Digest(M)$. If they match, the signature is valid.

11.2.3 Certificates

A digital certificate is a document containing a statement (usually short) signed by a principal. We illustrate the concept with a scenario.

Scenario 5. The use of certificates: Bob is a bank. When his customers establish contact with him they need to be sure that they are talking to Bob the bank, even if they have never contacted him before. Bob needs to authenticate his customers before he gives them access to their accounts.

For example, Alice might find it useful to obtain a certificate from her bank stating her bank account number (Figure 11.3). Alice could use this certificate when shopping to certify that she has an account with Bob's Bank. The certificate is signed using Bob's private key, K_{Bpriv}. A vendor, Carol, can accept such a certificate for charging items to Alice's account provided that she can validate the signature in field 5. To do so, Carol needs to have Bob's public key and she needs to be sure that it is authentic to guard against the possibility that Alice might sign a false certificate associating her name with someone else's account. To carry out this attack, Alice would simply generate a new key pair, $K_{B'pub}$, $K_{B'priv}$, and use them to generate a forged certificate purporting to come from Bob's Bank.

Figure 11.4 Public-key certificate for Bob's Bank

1. *Certificate type*:	Public key
2. *Name*:	Bob's Bank
3. *Public key*:	K_{Bpub}
4. *Certifying authority*:	Fred – The Bankers Federation
5. *Signature*:	$\{Digest(field\ 2 + field\ 3)\}_{K_{Fpriv}}$

What Carol needs is a certificate stating Bob's public key, signed by a well-known and trusted authority. Let us assume that Fred represents the Bankers Federation, one of whose roles is to certify the public keys of banks. Fred could issue a *public-key certificate* for Bob (Figure 11.4).

Of course, this certificate depends upon the authenticity of Fred's public key, K_{Fpub}, so we have a recursive problem of authenticity – Carol can only rely on this certificate if she can be sure she knows Fred's authentic public key, K_{Fpub}. We can break this recursion by ensuring that Carol obtains K_{Fpub} by some means in which she can have confidence – she might be handed it by a representative of Fred or she might receive a signed copy of it from someone she knows and trusts who says they got it directly from Fred. Our example illustrates a certification chain – one with two links, in this case.

We have already alluded to one of the problems arising with certificates – the difficulty of choosing a trusted authority from which a chain of authentications can start. Trust is seldom absolute, so the choice of an authority must depend upon the purpose to which the certificate is to be put. Other problems arise over the risk of private keys being compromised (disclosed) and the permissible length of a certification chain – the longer the chain, the greater the risk of a weak link.

Provided that care is taken to address these issues, chains of certificates are an important cornerstone for electronic commerce and other kinds of real-world transaction. They help to address the problem of scale: there are six billion people in the world, so how can we construct an electronic environment in which we can establish the credentials of any of them?

Certificates can be used to establish the authenticity of many types of statement. For example, the members of a group or association might wish to maintain an email list that is open only to members of the group. A good way to do this would be for the membership manager (Bob) to issue a membership certificate $(S, Bob, \{Digest(S)\}_{K_{Bpriv}})$ to each member, where S is a statement of the form *Alice is a member of the Friendly Society* and K_{Bpriv} is Bob's private key. A member applying to join the Friendly Society email list would have to supply a copy of this certificate to the list management system, which checks the certificate before allowing the member to join the list.

To make certificates useful, two things are needed:

- a standard format and representation for them so that certificate issuers and certificate users can successfully construct and interpret them;

- agreement on the manner in which chains of certificates are constructed, and in particular the notion of a trusted authority.

We return to these requirements in Section 11.4.4.

There is sometimes a need to revoke a certificate – for example, Alice might discontinue her membership of the Friendly Society, but she and others would probably continue to hold stored copies of her membership certificate. It would be expensive, if not impossible, to track down and delete all such certificates, and it is not easy to invalidate a certificate – it would be necessary to notify all possible recipients of the revocation. The usual solution to this problem is to include an expiry date in the certificate. Anyone receiving an expired certificate should reject it, and the subject of the certificate must request its renewal. If a more rapid revocation is required, then one of the more cumbersome mechanisms mentioned above must be resorted to.

11.2.4 Access control

Here we outline the concepts on which the control of access to resources is based in distributed systems and the techniques by which it is implemented. The conceptual basis for protection and access control was very clearly set out in a classic paper by Lampson [1971], and details of non-distributed implementations can be found in many books on operating systems (see e.g., [Stallings 2008]).

Historically, the protection of resources in distributed systems has been largely service-specific. Servers receive request messages of the form *<op, principal, resource>*, where *op* is the requested operation, *principal* is an identity or a set of credentials for the principal making the request and *resource* identifies the resource to which the operation is to be applied. The server must first authenticate the request message and the principal's credentials and then apply access control, refusing any request for which the requesting principal does not have the necessary access rights to perform the requested operation on the specified resource.

In object-oriented distributed systems there may be many types of object to which access control must be applied, and the decisions are often application-specific. For example, Alice may be allowed only one cash withdrawal from her bank account per day, while Bob is allowed three. Access control decisions are usually left to the application-level code, but generic support is provided for much of the machinery that supports the decisions. This includes the authentication of principals, the signing and authentication of requests, and the management of credentials and access rights data.

Protection domains • A protection domain is an execution environment shared by a collection of processes: it contains a set of *<resource, rights>* pairs, listing the resources that can be accessed by all processes executing within the domain and specifying the operations permitted on each resource. A protection domain is usually associated with a given principal – when a user logs in, their identity is authenticated and a protection domain is created for the processes that they will run. Conceptually, the domain includes all of the access rights that the principal possesses, including any rights that they acquire through membership of various groups. For example, in UNIX, the protection domain of a process is determined by the user and group identifiers attached to the process at login time. Rights are specified in terms of allowed operations. For example, a file might be readable and writable by one process and only readable by another.

A protection domain is only an abstraction. Two alternative implementations are commonly used in distributed systems: *capabilities* and *access control lists*.

Capabilities: A set of capabilities is held by each process according to the domain in which it is located. A capability is a binary value that acts as an access key, allowing the holder access to certain operations on a specified resource. For use in distributed systems, where capabilities must be unforgeable, they take a form such as:

Resource identifier	A unique identifier for the target resource
Operations	A list of the operations permitted on the resource
Authentication code	A digital signature making the capability unforgeable

Services only supply capabilities to clients when they have authenticated them as belonging to the claimed protection domain. The list of operations in the capability is a subset of the operations defined for the target resource and is often encoded as a bit map. Different capabilities are used for different combinations of access rights to the same resource.

When capabilities are used, client requests are of the form *<op, userid, capability>*. That is, they include a capability for the resource to be accessed instead of a simple identifier, giving the server immediate proof that the client is authorized to access the resource identified by the capability with the operations specified by the capability. An access-control check on a request that is accompanied by a capability involves only the validation of the capability and a check that the requested operation is in the set permitted by the capability. This feature is the major advantage of capabilities – they constitute a self-contained access key, just as a physical key to a door lock is an access key to the building that the lock protects.

Capabilities share two drawbacks of keys to a physical lock:

Key theft: Anyone who holds the key to a building can use it to gain access, whether or not they are an authorized holder of the key – they may have stolen the key or obtained it in some fraudulent manner.

The revocation problem: The entitlement to hold a key changes with time. For example, the holder may cease to be an employee of the owner of the building, but they might retain the key, or a copy of it, and use it in an unauthorized manner.

The only available solutions to these problems for physical keys are (a) to put the illicit key holder in jail – not always feasible on a timescale that will prevent them doing damage – or (b) to change the lock and reissue keys to all key holders – a clumsy and expensive operation.

The analogous problems for capabilities are clear:

- Capabilities may, through carelessness or as a result of an eavesdropping attack, fall into the hands of principals other than those to whom they were issued. If this happens, servers are powerless to prevent them being used illicitly.
- It is difficult to cancel capabilities. The status of the holder may change and their access rights should change accordingly, but they can still use their capabilities.

Solutions to both of these problems, based on the inclusion of information identifying the holder and on timeouts plus lists of revoked capabilities, respectively, have been proposed and developed [Gong 1989, Hayton *et al.* 1998]. Although they add

complexity to an otherwise simple concept, capabilities remain an important technique – for example, they can be used in conjunction with access control lists to optimize access control on repeated access to the same resource, and they provide the neatest mechanism for the implementation of delegation (see Section 11.2.5).

It is interesting to note the similarity between capabilities and certificates. Consider Alice's certificate of ownership of her bank account introduced in Section 11.2.3. It differs from capabilities as described here only in that there is no list of permitted operations and that the issuer is identified. Certificates and capabilities may be interchangeable concepts in some circumstances. Alice's certificate might be regarded as an access key allowing her to perform all the operations permitted to account holders on her bank account, provided her identity can be proven.

Access control lists: A list is stored with each resource, containing an entry of the form *<domain, operations>* for each domain that has access to the resource and giving the operations permitted to the domain. A domain may be specified by an identifier for a principal or it may be an expression that can be used to determine a principal's membership of the domain. For example, *the owner of this file* is an expression that can be evaluated by comparing the requesting principal's identity with the owner's identity stored with a file.

This is the scheme adopted in most file systems, including UNIX and Windows NT, where a set of access permission bits is associated with each file, and the domains to which the permissions are granted are defined by reference to the ownership information stored with each file.

Requests to servers are of the form *<op, principal, resource>*. For each request, the server authenticates the principal and checks to see that the requested operation is included in the principal's entry in the access control list of the relevant resource.

Implementation • Digital signatures, credentials and public-key certificates provide the cryptographic basis for secure access control. Secure channels offer performance benefits, enabling multiple requests to be handled without a need for repeated checking of principals and credentials [Wobber *et al.* 1994].

Both CORBA and Java offer Security APIs. Support for access control is one of their major purposes. Java provides support for distributed objects to manage their own access control with *Principal*, *Signer* and *ACL* classes and default methods for authentication and support for certificates, signature validation and access-control checks. Secret-key and public-key cryptography are also supported. Farley [1998] provides a good introduction to these features of Java. The protection of Java programs that include mobile code is based upon the protection domain concept – local code and downloaded code are provided with different protection domains in which to execute. There can be a protection domain for each download source, with access rights for different sets of local resources depending upon the level of trust that is placed in the downloaded code.

Corba offers a Security Service specification [Blakley 1999, OMG 2002b] with a model for ORBs to provide secure communication, authentication, access control with credentials, ACLs and auditing; these are described further in Section 8.3.

11.2.5 Credentials

Credentials are a set of evidence provided by a principal when requesting access to a resource. In the simplest case, a certificate from a relevant authority stating the principal's identity is sufficient, and this would be used to check the principal's permissions in an access control list (see Section 11.2.4). This is often all that is required or provided, but the concept can be generalized to deal with many more subtle requirements.

It is not convenient to require users to interact with the system and authenticate themselves each time their authority is required to perform an operation on a protected resource. Instead, the notion that a credential *speaks for* a principal is introduced. Thus a user's public-key certificate speaks for that user – any process receiving a request authenticated with the user's private key can assume that the request was issued by that user.

The *speaks for* idea can be carried much further. For example, in a cooperative task, it might be required that certain sensitive actions should only be performed with the authority of two members of the team; in that case, the principal requesting the action would submit their own identifying credential and a backing credential from another member of the team, together with an indication that they are to be taken together when checking the credentials.

Similarly, to vote in an election, a vote request would be accompanied by an elector certificate as well as an identifying certificate. A delegation certificate allows a principal to act on behalf of another, and so on. In general, an access-control check involves the evaluation of a logical formula combining the certificates supplied. Lampson *et al.* [1992] have developed a comprehensive logic of authentication for use in evaluating the *speaks for* authority carried by a set of credentials. Wobber *et al.* [1994] describe a system that supports this very general approach. Further work on useful forms of credential for use in real-world cooperative tasks can be found in Rowley [1998].

Role-based credentials seem particularly useful in the design of practical access control schemes [Sandhu *et al.* 1996]. Sets of role-based credentials are defined for organizations or for cooperative tasks, and application-level access rights are constructed with reference to them. Roles can then be assigned to specific principals by the generation of role certificates associating principals with named roles in specific tasks or organizations [Coulouris *et al.* 1998].

Delegation • A particularly useful form of credential is one that entitles a principal, or a process acting for a principal, to perform an action with the authority of another principal. A need for delegation can arise in any situation where a service needs to access a protected resource in order to complete an action on behalf of its client. Consider the example of a print server that accepts requests to print files. It would be wasteful of resources to copy the file, so the name of the file is passed to the print server and it is accessed by the print server on behalf of the user making the request. If the file is read-protected, this does not work unless the print server can acquire temporary rights to read the file. Delegation is a mechanism designed to solve problems such as this.

Delegation can be achieved using a delegation certificate or a capability. The certificate is signed by the requesting principal and it authorizes another principal (the print server in our example) to access a named resource (the file to be printed). In systems that support them, capabilities can achieve the same result without the need to

identify the principals – a capability to access a resource can be passed in a request to a server. The capability is an unforgeable, encoded set of rights to access the resource.

When rights are delegated, it is common to restrict them to a subset of the rights held by the issuing principal, so that the delegated principal cannot misuse them. In our example, the certificate could be time-limited to reduce the risk of the print server's code subsequently being compromised and the file disclosed to third parties. The CORBA Security Service includes a mechanism for the delegation of rights based on certificates, with support for the restriction of the rights carried.

11.2.6 Firewalls

Firewalls were introduced and described in Section 3.4.8. They protect intranets, performing filtering actions on incoming and outgoing communications. Here we discuss their advantages and drawbacks as security mechanisms.

In an ideal world, communication would always be between mutually trusting processes and secure channels would always be used. There are many reasons why this ideal is not attainable, some fixable, but others inherent in the open nature of distributed systems or resulting from the errors that are present in most software. The ease with which request messages can be sent to any server, anywhere, and the fact that many servers are not designed to withstand malicious attacks from hackers or accidental errors, makes it easy for information that is intended to be confidential to leak out of the owning organization's servers. Undesirable items can also penetrate an organization's network, allowing worm programs and viruses to enter its computers. See [web.mit.edu II] for a further critique of firewalls.

Firewalls produce a local communication environment in which all external communication is intercepted. Messages are forwarded to the intended local recipient only for communications that are explicitly authorized.

Access to internal networks may be controlled by firewalls, but access to public services on the Internet is unrestricted because their purpose is to offer services to a wide range of users. The use of firewalls offers no protection against attacks from inside an organization, and it is crude in its control of external access. There is a need for finer-grained security mechanisms, enabling individual users to share information with selected others without compromising privacy and integrity. Abadi *et al.* [1998] describe an approach to the provision of access to private web data for external users based on a *web tunnel* mechanism that can be integrated with a firewall. It offers access for trusted and authenticated users to internal web servers via a secure proxy based on the HTTPS (HTTP over TLS) protocol.

Firewalls are not particularly effective against denial-of-service attacks such as the one based on IP spoofing that was outlined in Section 3.4.2. The problem is that the flood of messages generated by such attacks overwhelms any single point of defence such as a firewall. Any remedy for incoming floods of messages must be applied well upstream of the target. Remedies based on the use of quality of service mechanisms to restrict the flow of messages from the network to a level that the target can handle seem the most promising.

11.3 Cryptographic algorithms

A message is encrypted by the sender applying some rule to transform the *plaintext* message (any sequence of bits) to a *ciphertext* (a different sequence of bits). The recipient must know the inverse rule in order to transform the ciphertext back into the original plaintext. Other principals are unable to decipher the message unless they also know the inverse rule. The encryption transformation is defined with two parts, a *function E* and a *key K*. The resulting encrypted message is written $\{M\}_K$.

$$E(K, M) = \{M\}_K$$

The encryption function E defines an algorithm that transforms data items in plaintext into encrypted data items by combining them with the key and transposing them in a manner that is heavily dependent on the value of the key. We can think of an encryption algorithm as the specification of a large family of functions from which a particular member is selected by any given key. Decryption is carried out using an inverse function D, which also takes a key as a parameter. For secret-key encryption, the key used for decryption is the same as that used for encryption:

$$D(K, E(K, M)) = M$$

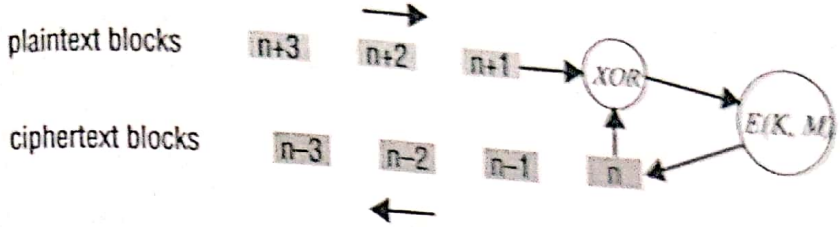
Because of its symmetrical use of keys, secret-key cryptography is often referred to as *symmetric cryptography*, whereas public-key cryptography is referred to as *asymmetric* because the keys used for encryption and decryption are different, as we shall see below. In the next section, we describe several widely used encryption functions of both types.

Symmetric algorithms • If we remove the key parameter from consideration by defining $F_K([M]) = E(K, M)$, then it is a property of strong encryption functions that $F_K([M])$ is relatively easy to compute, whereas the inverse, $F_K^{-1}([M])$, is so hard to compute that it is not feasible. Such functions are known as one-way functions. The effectiveness of any method for encrypting information depends upon the use of an encryption function F_K that has this one-way property. It is this that protects against attacks designed to discover M given $\{M\}_K$.

For well-designed symmetric algorithms such as those described in the next section, their strength against attempts to discover K given a plaintext M and the corresponding ciphertext $\{M\}_K$ depends on the size of K. This is because the most effective general form of attack is the crudest, known as a *brute-force attack*. The brute-force approach is to run through all possible values of K, computing $E(K, M)$ until the result matches the value of $\{M\}_K$ that is already known. If K has N bits then such an attack requires 2^{N-1} iterations on average, and a maximum of 2^N iterations, to find K. Hence the time to crack K is exponential in the number of bits in K.

Asymmetric algorithms • When a public/private key pair is used, one-way functions are exploited in another way. The feasibility of a public-key scheme was first proposed by Diffie and Hellman [1976] as a cryptographic method that eliminates the need for trust between the communicating parties. The basis for all public-key schemes is the existence of *trap-door functions*. A trap-door function is a one-way function with a secret exit – it is easy to compute in one direction but infeasible to compute the inverse unless a secret is known. It was the possibility of finding such functions and using them

Figure 11.5 Cipher block chaining



in practical cryptography that Diffie and Hellman first suggested. Since then, several practical public-key schemes have been proposed and developed. They all depend upon the use of trap-door functions involving large numbers.

The pair of keys needed for asymmetric algorithms is derived from a common root. For the RSA algorithm, described in Section 11.3.2, the root is an arbitrarily chosen pair of very large prime numbers. The derivation of the pair of keys from the root is a one-way function. In the case of the RSA algorithm, the large primes are multiplied together – a computation that takes only a few seconds, even for the very large primes used. The resulting product, N, is of course much larger than the multiplicands. This use of multiplication is a one-way function in the sense that it is computationally infeasible to derive the original multiplicands from the product – that is, to factorize the product.

One of the pair of keys is used for encryption. For RSA, the encryption function obscures the plaintext by treating each block of bits as a binary number and raising it to the power of the key, modulo N. The resulting number is the corresponding ciphertext block.

The size of N and at least one of the pair of keys is much larger than the safe key size for symmetric keys to ensure that N is not factorizable. For this reason, the potential for brute-force attacks on RSA is small; its resistance to attacks depends on the infeasibility of factorizing N. We discuss safe sizes for N in Section 11.3.2.

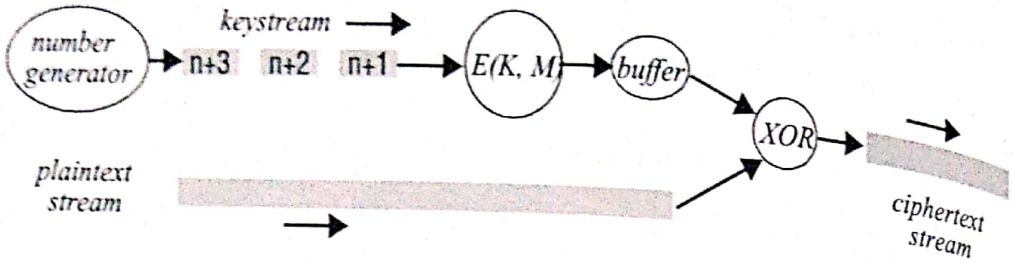
Block ciphers • Most encryption algorithms operate on fixed-size blocks of data; 64 bits is a popular size for the blocks. A message is subdivided into blocks, the last block is padded to the standard length if necessary and each block is encrypted independently. The first block is available for transmission as soon as it has been encrypted.

For a simple block cipher, the value of each block of ciphertext does not depend upon the preceding blocks. This constitutes a weakness, since an attacker can recognize repeated patterns and infer their relationship to the plaintext. Nor is the integrity of messages guaranteed unless a checksum or secure digest mechanism is used. Most block cipher algorithms employ cipher block chaining (CBC) to overcome these weaknesses.

Cipher block chaining: In cipher block chaining mode, each plaintext block is combined with the preceding ciphertext block using the exclusive-or operation (XOR) before it is encrypted (Figure 11.5). On decryption, the block is decrypted and then the preceding encrypted block (which should have been stored for this purpose) is XOR-ed with it to obtain the new plaintext block. This works because the XOR operation is its own inverse – two applications of it produce the original value.

CBC is intended to prevent identical portions of plaintext encrypting to identical pieces of ciphertext. But there is a weakness at the start of each sequence of blocks – if

Figure 11.6 Stream cipher



we open encrypted connections to two destinations and send the same message, the encrypted sequences of blocks will be the same, and an eavesdropper might gain some useful information from this. To prevent this, we need to insert a different piece of plaintext in front of each message. Such text is called an *initialization vector*. A timestamp makes a good initialization vector, forcing each message to start with a different plaintext block. This, combined with CBC operation, will result in different ciphertexts even for two identical plaintexts.

The use of CBC mode is restricted to the encryption of data that is transferred across a reliable connection. Decryption will fail if any blocks of ciphertext are lost, since the decryption process will be unable to decrypt any further blocks. It is therefore unsuitable for use in applications such as those described in Chapter 18, in which some data loss can be tolerated. A stream cipher should be used in such circumstances.

Stream ciphers • For some applications, such as the encryption of telephone conversations, encryption in blocks is inappropriate because the data streams are produced in real time in small chunks. Data samples can be as small as 8 bits or even a single bit, and it would be wasteful to pad each of these to 64 bits before encrypting and transmitting them. Stream ciphers are encryption algorithms that can perform encryption incrementally, converting plaintext to ciphertext one bit at a time.

This sounds difficult to achieve, but in fact it is very simple to convert a block cipher algorithm for use as a stream cipher. The trick is to construct a *keystream generator*. A keystream is an arbitrary-length sequence of bits that can be used to obscure the contents of a data stream by XOR-ing the keystream with the data stream (Figure 11.6). If the keystream is secure, then so is the resulting encrypted data stream.

The idea is analogous to a technique used in the intelligence community to foil eavesdroppers, where 'white noise' is played to hide the conversation in a room while still recording the conversation. If the noisy room sound and the white noise are recorded separately, the conversation can be played back without noise by subtracting the white noise recording from the noisy room recording.

A keystream generator can be constructed by iterating a mathematical function over a range of input values to produce a continuous stream of output values. The output values are then concatenated to make plaintext blocks, and the blocks are encrypted using a key shared by the sender and the receiver. The keystream can be further disguised by applying CBC. The resulting encrypted blocks are used as the keystream. An iteration of almost any function that delivers a range of different non-integer values will do for the source material, but a random number generator is generally used with a starting value for the iteration agreed between the sender and receiver. To maintain quality of service for the data stream, the keystream blocks should be produced a little

ahead of the time at which they will be used, and the process that produces them should not demand so much processing effort that the data stream is delayed.

Thus in principle, real-time data streams can be encrypted just as securely as batched data, provided that sufficient processing power is available to encrypt the keystream in real time. Of course, some devices that could benefit from real-time encryption, such as mobile phones, are not equipped with very powerful processors, and in that case it may be necessary to reduce the security of the keystream algorithm.

Design of cryptographic algorithms • There are many well-designed cryptographic algorithms such that $E(K, M) = \{M\}_K$ conceals the value of M and makes it practically impossible to retrieve K more quickly than by brute force. All encryption algorithms rely on information-preserving manipulations of M using principles based on information theory [Shannon 1949]. Schneier [1996] describes Shannon's principles of *confusion* and *diffusion* to conceal the content of a ciphertext block M, combining it with a key K of sufficient size to render it proof against brute-force attacks.

Confusion: Non-destructive operations such as XOR and circular shifting are used to combine each block of plaintext with the key, producing a new bit pattern that obscures the relationship between the blocks in M and $\{M\}_K$. If the blocks are larger than a few characters this will defeat analysis based on a knowledge of character frequencies. (The WWII German Enigma machine used chained single-letter blocks, and was eventually defeated by statistical analysis.)

Diffusion: There is usually repetition and redundancy in the plaintext. Diffusion dissipates the regular patterns that result by transposing portions of each plaintext block. If CBC is used, the redundancy is also distributed throughout a longer text. Stream ciphers cannot use diffusion since there are no blocks.

In the next two sections, we describe the design of several important practical algorithms. All of them have been designed in the light of the above principles have been subject to rigorous analysis and are considered to be secure against all known attacks with a considerable margin of safety. With the exception of the TEA algorithm, which is described for illustrative purposes, the algorithms described here are among those most widely used in applications where strong security is required. In some of them there remain some minor weaknesses or areas of concern; space does not allow us to describe all of those concerns here, and the reader is referred to Schneier [1996] for further information. We summarize and compare the security and performance of the algorithms in Section 11.5.1.

Readers who do not require an understanding of the operation of cryptographic algorithms may omit Sections 11.3.1 and 11.3.2.

11.3.1 Secret-key (symmetric) algorithms

Many cryptographic algorithms have been developed and published in recent years. Schneier [1996] describes more than 25 symmetric algorithms, many of which he identifies as secure against known attacks. Here we have room to describe only three of them. We have chosen the first, TEA, for the simplicity of its design and implementation, and we use it to give a concrete illustration of the nature of such algorithms. We go on to discuss the DES and IDEA algorithms in less detail. DES was

Figure 11.7 TEA encryption function

```
void encrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];
    unsigned long delta = 0x9e3779b9, sum = 0; int n;
    for (n= 0; n < 32; n++) {
        sum += delta;
        y += ((z << 4) + k[0]) ^ (z+sum) ^ ((z >> 5) + k[1]);
        z += ((y << 4) + k[2]) ^ (y+sum) ^ ((y >> 5) + k[3]);
    }
    text[0] = y;  text[1] = z;
}
```

a US national standard for many years, but it is now largely of historical interest because its 56-bit keys are too small to resist brute-force attack with modern hardware. IDEA uses a 128-bit key. It is one of the most effective symmetric block encryption algorithms and a good all-round choice for bulk encryption.

In 1997, the US National Institute for Standards and Technology (NIST) issued an invitation for proposals for an algorithm to replace DES as a new US Advanced Encryption Standard (AES). In October 2000 the winner was selected from 21 algorithms submitted by cryptographers from 11 countries. The winning Rijndael algorithm was chosen for its combination of strength and efficiency. Further information on it is given below.

TEA • The design principles for symmetric algorithms outlined above are illustrated well in the Tiny Encryption Algorithm (TEA) developed at Cambridge University [Wheeler and Needham 1994]. The encryption function, programmed in C, is given in its entirety in Figure 11.7.

The TEA algorithm uses rounds of integer addition, XOR (the ^ operator) and bitwise logical shifts (<< and >>) to achieve diffusion and confusion of the bit patterns in the plaintext. The plaintext is a 64-bit block represented as two 32-bit integers in the vector *text[]*. The key is 128 bits long, represented as four 32-bit integers.

On each of the 32 rounds, the two halves of the text are repeatedly combined with shifted portions of the key and each other in lines 5 and 6. The use of XOR and shifted portions of the text provides confusion, and the shifting and swapping of the two portions of the text provides diffusion. The non-repeating constant *delta* is combined with each portion of the text on each cycle to obscure the key in case it might be revealed by a section of text that does not vary. The decryption function is the inverse of that for encryption and is given in Figure 11.8.

This short program provides secure and reasonably fast secret-key encryption. It is somewhat faster than the DES algorithm, and the conciseness of the program lends itself to optimization and hardware implementation. The 128-bit key is secure against brute-force attacks. Studies by its authors and others have revealed only two very minor weaknesses, which the authors addressed in a subsequent note [Wheeler and Needham 1997].

Figure 11.8 TEA decryption function

```
void decrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];
    unsigned long delta = 0x9e3779b9, sum = delta << 5;  int n;
    for (n = 0; n < 32; n++) {
        z -= ((y << 4) + k[2]) ^ (y + sum) ^ ((y >> 5) + k[3]);
        y -= ((z << 4) + k[0]) ^ (z + sum) ^ ((z >> 5) + k[1]);
        sum -= delta;
    }
    text[0] = y; text[1] = z;
}
```

To illustrate its use, Figure 11.9 shows a simple procedure that uses TEA to encrypt and decrypt a pair of previously opened files (using the C *stdio* library).

DES • The Data Encryption Standard (DES) [National Bureau of Standards 1977] was developed by IBM and subsequently adopted as a US national standard for government and business applications. In this standard, the encryption function maps a 64-bit plaintext input into a 64-bit encrypted output using a 56-bit key. The algorithm has 16 key-dependent stages known as *rounds*, in which the data to be encrypted is bit-rotated by a number of bits determined by the key and three key-independent transpositions. The algorithm was time-consuming to perform in software on the computers of the 1970s and 1980s, but it was implemented in fast VLSI hardware and can easily be incorporated into network interface and other communication chips.

Figure 11.9 TEA in use

```
void tea(char mode, FILE *infile, FILE *outfile, unsigned long k[]) {
/* mode is 'e' for encrypt, 'd' for decrypt, k[] is the key.*/
    char ch, Text[8]; int i;

    while(!feof(infile)) {
        i = fread(Text, 1, 8, infile);          /* read 8 bytes from infile into Text */
        if (i <= 0) break;
        while (i < 8) { Text[i++] = ' ';}       /* pad last block with spaces */
        switch (mode) {
        case 'e':
            encrypt(k, (unsigned long*) Text); break;
        case 'd':
            decrypt(k, (unsigned long*) Text); break;
        }
        fwrite(Text, 1, 8, outfile);            /* write 8 bytes from Text to outfile */
    }
}
```

In June 1997, it was successfully cracked in a widely publicized brute-force attack. The attack was performed in the context of a competition to demonstrate the lack of security of encryption with keys shorter than 128 bits [www.rsasecurity.com II]. A consortium of Internet users ran a client application program on a their PCs and other workstations, whose numbers reached 14,000 during one 24-hour period [Curtin and Dolske 1998].

The client program was aimed at cracking the particular key used in a known plaintext/ciphertext sample and then using it to decrypt a secret challenge message. The clients interacted with a single server that coordinated their work, issuing each client with ranges of key values to check and receiving progress reports from them. The typical client computer ran the client program only as a background activity and had a performance approximately equal to a 200 MHz Pentium processor. The key was cracked in about 12 weeks, after approximately 25% of the possible 2^{56} or 6×10^{16} values had been checked. In 1998 a machine was developed by the Electronic Frontier Foundation [EFF 1998] that can successfully crack DES keys in around three days.

Although it is still used in many commercial and other applications, DES in its basic form should be considered obsolete for the protection of all but low-value information. A solution that is frequently used is known as *triple-DES* (or *3DES*) [ANSI 1985, Schneier 1996]. This involves applying DES three times with two keys, K_1 and K_2:

$$E_{3DES}(K_1, K_2, M) = E_{DES}(K_1, D_{DES}(K_2, E_{DES}(K_1, M)))$$

This gives a strength against brute-force attacks equivalent to a key length of 112 bits – adequate for the foreseeable future – but it has the drawback of poor performance resulting from the triple application of an algorithm that is already slow by modern standards.

IDEA • The International Data Encryption Algorithm (IDEA) was developed in the early 1990s [Lai and Massey 1990, Lai 1992] as a successor to DES. Like TEA, it uses a 128-bit key to encrypt 64-bit blocks. Its algorithm is based on the algebra of groups and has eight rounds of XOR, addition modulo 2^{16} and multiplication. For both DES and IDEA, the same function is used for encryption and decryption: a useful property for algorithms that are to be implemented in hardware.

The strength of IDEA has been extensively analyzed, and no significant weaknesses have been found. It performs encryption and decryption at approximately three times the speed of DES.

RC4 • RC4 is a stream cipher developed by Ronald Rivest [Rivest 1992b]. Keys can be of any length up to 256 bytes. RC4 is easy to implement [Schneier 1996, pp. 397–8] and performs encryption and decryption about 10 times as fast as DES. It was therefore widely adopted in applications including IEEE 802.11 WiFi networks, but a weakness was subsequently discovered by Fluhrer *et al.* [2001] that enabled attackers to crack some keys. This led to a redesign of 802.11 security (see Section 11.6.4 for further details).

AES • The Rijndael algorithm selected to become the Advanced Encryption Standard algorithm by NIST was developed by Joan Daemen and Vincent Rijmen [Daemen and Rijmen 2000, 2002]. The cipher has a variable block length and key length, with specifications for keys with a length of 128, 192 or 256 bits to encrypt blocks with a

length of 128, 192 or 256 bits. Both block length and key length can be extended by multiples of 32 bits. The number of rounds in the algorithm varies from 9 to 13 depending on the key and block sizes. Rijndael can be implemented efficiently on a wide range of processors and in hardware.

11.3.2 Public-key (asymmetric) algorithms

Only a few practical public-key schemes have been developed to date. They depend upon the use of trap-door functions of large numbers to produce the keys. The keys K_e and K_d are a pair of very large numbers, and the encryption function performs an operation, such as exponentiation on M, using one of them. Decryption is a similar function using the other key. If the exponentiation uses modular arithmetic, it can be shown that the result is the same as the original value of M; that is:

$$D(K_d, E(K_e, M)) = M$$

A principal wishing to participate in secure communication with others makes a pair of keys, K_e and K_d, and keeps the decryption key K_d a secret. The encryption key K_e can be made known publicly for use by anyone who wants to communicate. The encryption key K_e can be seen as a part of the one-way encryption function E, and the decryption key K_d is the piece of secret knowledge that enables principal p to reverse the encryption. Any holder of K_e (which is widely available) can encrypt messages $\{M\}_{K_e}$, but only the principal who has the secret K_d can operate the trapdoor.

The use of functions of large numbers leads to large processing costs in computing the functions E and D. We shall see later that this is a problem that has to be addressed by the use of public keys only in the initial stages of secure communication sessions. The RSA algorithm is certainly the most widely known public-key algorithm and we describe it in some detail here. Another class of algorithms is based on functions derived from the behaviour of elliptic curves in a plane. These algorithms offer the possibility of less costly encryption and decryption functions with the same level of security, but their practical application is less advanced and we deal with them only briefly.

RSA • The Rivest, Shamir and Adelman (RSA) design for a public-key cipher [Rivest *et al.* 1978] is based on the use of the product of two very large prime numbers (greater than 10^{100}), relying on the fact that the determination of the prime factors of such large numbers is so computationally difficult as to be effectively impossible.

Despite extensive investigations no flaws have been found in it, and it is now very widely used. An outline of the method follows. To find a key pair $\langle e,d \rangle$:

1. Choose two large prime numbers, P and Q (each greater than 10^{100}), and form
   $N = P \times Q$
   $Z = (P-1) \times (Q-1)$
2. For d, choose any number that is relatively prime with Z (that is, such that d has no common factors with Z).

We illustrate the computations involved using small integer values for P and Q:

$P = 13, Q = 17 \rightarrow N = 221, Z = 192$

$d = 5$

3. To find e, solve the equation:
   $e \times d = 1 \ mod \ Z$

That is, $e \times d$ is the smallest element divisible by d in the series $Z+1, 2Z+1, 3Z+1, \ldots$

$e \times d = 1 \ mod \ 192 \ = 1, 193, 385, \ldots$
385 is divisible by d
$e = 385/5 = 77$

To encrypt text using the RSA method, the plaintext is divided into equal blocks of length k bits, where $2^k < N$ (that is, such that the numerical value of a block is always less than N; in practical applications, k is usually in the range 512 to 1024).

$k = 7$, since $2^7 = 128$

The function for encrypting a single block of plaintext M is:

$$E'(e,N,M) = M^e \ mod \ N$$

for a message M, the ciphertext is $M^{77} \ mod \ 221$

The function for decrypting a block of encrypted text c to produce the original plaintext block is:

$$D'(d,N,c) = c^d \ mod \ N$$

Rivest, Shamir and Adelman proved that E' and D' are mutual inverses (that is, $E'(D'(x)) = D'(E'(x)) = x$) for all values of P in the range $0 \leq P \leq N$.

The two parameters e,N can be regarded as a key for the encryption function, and similarly the parameters d,N represent a key for the decryption function. So we can write $K_e = <e,N>$ and $K_d = <d,N>$, and we get the encryption functions $E(K_e, M) = \{M\}_K$ (the notation here indicating that the encrypted message can be decrypted only by the holder of the *private* key K_d) and $D(K_d, \{M\}_K) = M$.

It is worth noting one potential weakness of all public-key algorithms – because the public key is available to attackers, they can easily generate encrypted messages. Thus they can attempt to decrypt an unknown message by exhaustively encrypting arbitrary bit sequences until a match with the target message is achieved. This attack, which is known as a *chosen plaintext attack*, is defeated by ensuring that all messages are longer than the key length, so that this form of brute-force attack is less feasible than a direct attack on the key.

An intending recipient of secret information must publish or otherwise distribute the pair $<e,N>$ while keeping d secret. The publication of $<e,N>$ does not compromise the secrecy of d, because any attempt to determine d requires knowledge of the original prime numbers P and Q, and these can only be obtained by the factorization of N. Factoring of large numbers (we recall that P and Q were chosen to be $> 10^{100}$, so $N > 10^{200}$) is extremely time-consuming, even on very high-performance computers. In 1978, Rivest *et al.* concluded that factoring a number as large as 10^{200} would take more than four billion years with the best known algorithm on a computer that performs one

million instructions per second. A similar calculation for today's computers would reduce this time to around a million years,

The RSA Corporation has issued a series of challenges to factor numbers of more than 100 decimal digits [www.rsasecurity.com III]. At the time of writing, numbers of up to 174 decimal digits (576 binary digits) have been successfully factored, so the use of the RSA algorithm with 512-bit keys is clearly unacceptably weak for many purposes. The RSA Corporation (holders of the patents in the RSA algorithm) recommends a key length of at least 768 bits, or about 230 decimal digits, for long-term (~20 years) security. Keys as large as 2048 bits are used in some applications.

The above strength calculations assume that the currently known factoring algorithms are the best available. RSA and other forms of asymmetric cryptography that use prime number multiplication as their one-way function will be vulnerable if a faster factorization algorithm is discovered.

Elliptic curve algorithms • A method for generating public/private key pairs based on the properties of elliptic curves has been developed and tested. Full details can be found in the book by Menezes devoted to the subject [Menezes 1993]. The keys are derived from a different branch of mathematics, and unlike RSA their security does not depend upon the difficulty of factoring large numbers. Shorter keys are secure, and the processing requirements for encryption and decryption are lower than those for RSA. Elliptic curve encryption algorithms are likely to be adopted more widely in the future, especially in systems such as those incorporating mobile devices, which have limited processing resources. The relevant mathematics involves some quite complex properties of elliptic curves and is beyond the scope of this book.

11.3.3 Hybrid cryptographic protocols

Public-key cryptography is convenient for electronic commerce because there is no need for a secure key-distribution mechanism. (There is a need to authenticate public keys, but this is much less onerous, requiring only a public-key certificate to be sent with the key.) But the processing costs of public-key cryptography are too high for the encryption of even the medium-sized messages normally encountered in electronic commerce. The solution adopted in most large-scale distributed systems is to use a hybrid encryption scheme in which public-key cryptography is used to authenticate the parties and to encrypt an exchange of secret keys, which are used for all subsequent communication. We describe the implementation of a hybrid protocol in the TLS case study in Section 11.6.3.

11.4 Digital signatures

Strong digital signatures are an essential requirement for secure systems. They are needed in order to certify certain pieces of information – for example, to provide trustworthy statements binding users' identities to their public keys or binding some access rights or roles to users' identities.

The need for signatures in many kinds of business and personal transaction is beyond dispute. Handwritten signatures have been used as a means of verifying documents for as long as documents have existed. Handwritten signatures are used to meet the needs of document recipients to verify that the document is:

Authentic: It convinces the recipient that the signer deliberately signed the document and it has not been altered by anyone else.

Unforgeable: It provides proof that the signer, and no one else, deliberately signed the document. The signature cannot be copied and placed on another document.

Non-repudiable: The signer cannot credibly deny that the document was signed by them.

In reality, none of these desirable properties of signing is entirely achieved by conventional signatures – forgeries and copies are hard to detect, documents can be altered after signing and signers are sometimes deceived into signing a document involuntarily or unwittingly – but we are willing to live with their imperfection because of the difficulty of cheating and the risk of detection. Like handwritten signatures, digital signatures depend upon the binding of a unique and secret attribute of the signer to a document. In the case of handwritten signatures, the secret is the handwriting pattern of the signer.

The properties of digital documents held in stored files or messages are completely different from those of paper documents. Digital documents are trivially easy to generate, copy and alter. Simply appending the identity of the originator, whether as a text string, a photograph or a handwritten image, has no value for verification purposes.

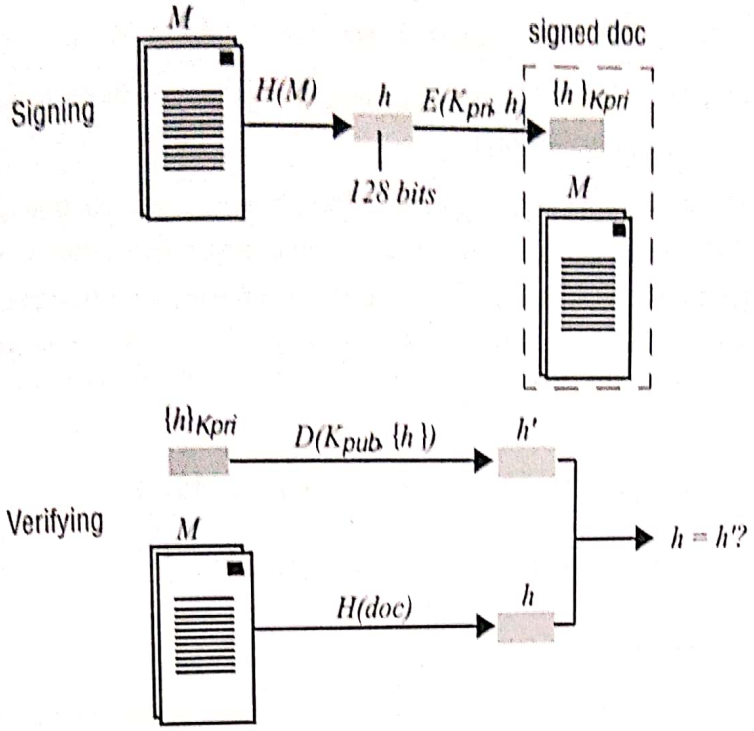
What is needed is a means to irrevocably bind a signer's identity to the entire sequence of bits representing a document. This should meet the first requirement above, for authenticity. As with handwritten signatures, though, the date of a document cannot be guaranteed by a signature. The recipient of a signed document knows only that the document was signed before they received it.

Regarding non-repudiation, there is a problem that does not arise with handwritten signatures. What if the signer deliberately reveals their private key and subsequently denies having signed, saying that there are others who could have done so, since the key was not private? Some protocols have been developed to address this problem under the heading of *undeniable digital signatures* [Schneier 1996], but they add considerably to the complexity.

A document with a digital signature can be considerably more resistant to forgery than a handwritten one. But the word 'original' has little meaning with reference to digital documents. As we shall see in our discussion of the needs of electronic commerce, digital signatures alone cannot, for example, prevent double-spending of electronic cash – other measures are needed to prevent that. We now describe two techniques for signing documents digitally, binding a principal's identity to the document. Both depend upon the use of cryptography.

Digital signing • An electronic document or message M can be signed by a principal A by encrypting a copy of M with a key K_A and attaching it to a plaintext copy of M and A's identifier. The signed document then consists of: $M, A, [M]_{K_A}$. The signature can be

Figure 11.10 Digital signatures with public keys



verified by a principal that subsequently receives the document to check that it was originated by A and that its contents, M, have not subsequently been altered.

If a secret key is used to encrypt the document, only principals that share the secret can verify the signature. But if public-key cryptography is used, then the signer uses their private key and anyone who has the corresponding public key can verify the signature. This is a better analogue for conventional signatures and meets a wider range of user needs. The verification of signatures proceeds differently depending on whether secret-key or public-key cryptography is used to produce the signature. We describe the two cases in Sections 11.4.1 and 11.4.2.

Digest functions • Digest functions are also called *secure hash functions* and denoted $H(M)$. They must be carefully designed to ensure that $H(M)$ is different from $H(M')$ for all likely pairs of messages M and M'. If there are any pairs of different messages M and M' such that $H(M) = H(M')$, then a duplicitous principal could send a signed copy of M, but when confronted with it claim that M' was originally sent and that it must have been altered in transit. We discuss some secure hash functions in Section 11.4.3.

11.4.1 Digital signatures with public keys

Public-key cryptography is particularly well adapted for the generation of digital signatures because it is relatively simple and does not require any communication between the recipient of a signed document and the signer or any third party.

The method for A to sign a message M and B to verify it is as follows (and is illustrated graphically in Figure 11.10):

1. A generates a key pair K_{pub} and K_{priv} and publishes the public key K_{pub} by placing it in a well-known location.

2. A computes the digest of M, $H(M)$ using an agreed secure hash function H and encrypts it using the private key K_{priv} to produce the signature $S = \{H(M)\}_{K_{priv}}$
3. A sends the signed message $[M]_K = M,S$ to B.
4. B decrypts S using K_{pub} and computes the digest of M, $H(M)$. If they match, the signature is valid.

The RSA algorithm is quite suitable for use in constructing digital signatures. Note that the *private* key of the signer is used to encrypt the signature, in contrast to the use of the recipient's *public* key for encryption when the aim is to transmit information in secrecy. The explanation for this difference is straightforward – a signature must be created using a secret known only to the signer and it should be accessible to all for verification.

11.4.2 Digital signatures with secret keys – MACs

There is no technical reason why a secret-key encryption algorithm should not be used to encrypt a digital signature, but in order to verify such signatures the key must be disclosed, and this causes some problems:

- The signer must arrange for the verifier to receive the secret key used for signing securely.
- It may be necessary to verify a signature in several contexts and at different times – at the time of signing, the signer may not know the identities of the verifiers. To resolve this, verification could be delegated to a trusted third party who holds secret keys for all signers, but this adds complexity to the security model and requires secure communication with the trusted third party.
- The disclosure of a secret key used for signing is undesirable because it weakens the security of signatures made with that key – a signature could be forged by a holder of the key who is not the owner of it.

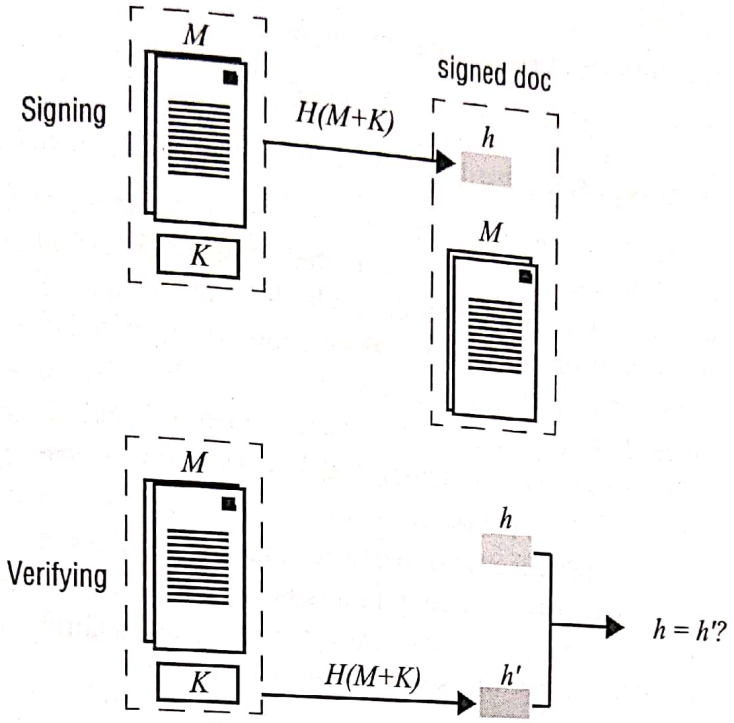
For all these reasons, the public-key method for generating and verifying signatures offers the most convenient solution in most situations.

An exception arises when a secure channel is used to transmit unencrypted messages but there is a need to verify the authenticity of the messages. Since a secure channel provides secure communication between a pair of processes, a shared secret key can be established using the hybrid method outlined in Section 11.3.3 and used to produce low-cost signatures. These signatures are called *message authentication codes* (MACs) to reflect their more limited purpose – they authenticate communication between pairs of principals based on a shared secret.

A low-cost signing technique based on shared secret keys that has adequate security for many purposes is illustrated in Figure 11.11 and outlined below. The method depends upon the existence of a secure channel through which the shared key can be distributed:

1. A generates a random key K for signing and distributes it using secure channels to one or more principals who will need to authenticate messages received from A. The principals are *trusted* not to disclose the shared key.

Figure 11.11 Low-cost signatures with a shared secret key



2. For any document M that A wishes to sign, A concatenates M with K, computes the digest of the result, $h = H(M+K)$, and sends the signed document $[M]_K = M, h$ to anyone wishing to verify the signature. (The digest h is a MAC.) K will not be compromised by the disclosure of h, since the hash function has totally obscured its value.
3. The receiver, B, concatenates the secret key K with the received document M and computes the digest $h' = H(M+K)$. The signature is verified if $h = h'$.

Although this method suffers from the disadvantages listed above, it has a performance advantage because it involves no encryption. (Secure hashing is typically about 3–10 times faster than symmetric encryption; see Section 11.5.1.) The TLS secure channel protocol described in Section 11.6.3 supports the use of a wide variety of MACs, including the scheme described here. The method is also used in the Millicent electronic cash protocol described at www.cdk5.net/security, where it is important to keep the processing cost low for low-value transactions.

11.4.3 Secure digest functions

There are many ways to produce a fixed-length bit pattern that characterizes an arbitrary-length message or document. Perhaps the simplest is to use the XOR operation iteratively to combine fixed-length pieces of the source document. Such a function is often used in communication protocols to produce a short fixed-length hash to characterize a message for error-detection purposes, but it is inadequate as the basis for

a digital signature scheme. A secure digest function $h = H(M)$ should have the following properties:

1. Given M, it is easy to compute h.
2. Given h, it is hard to compute M.
3. Given M, it is hard to find another message M', such that $H(M) = H(M')$.

Such functions are also called *one-way hash functions*. The reason for this name is self-evident based on the first two properties. Property 3 demands an additional feature: even though we know that the result of a hash function cannot be unique (because the digest is an information-reducing transformation), we need to be sure that an attacker, given a message M that produces a hash h, cannot discover another message M' that also produces h. If an attacker *could* do this, then they could forge a signed document M' without knowledge of the signing key by copying the signature from the signed document M and appending it to M'.

Admittedly, the set of messages that hash to the same value is restricted and the attacker would have difficulty in producing a meaningful forgery, but with patience it could be done, so it must be guarded against. The feasibility of doing so is considerably enhanced in the case of a so-called *birthday attack*:

1. Alice prepares two versions, M and M', of a contract for Bob. M is favourable to Bob and M' is not.
2. Alice makes several subtly different versions of both M and M' that are visually indistinguishable from each other by methods such as adding spaces at the ends of lines. She compares the hashes of all the Ms with all the M's. If she finds two that are the same, she can proceed to the next step; if not, she goes on producing visually indistinguishable versions of the two documents until she gets a match.
3. When she has a pair of documents M and M' that hash to the same value, she gives the favourable document M to Bob for him to sign with a digital signature using his private key. When he returns it, she substitutes the matching unfavourable version M', retaining the signature from M.

If our hash values are 64 bits long, we require only 2^{32} versions of M and M' on average. This is too small for comfort. We need to make our hash values at least 128 bits long to guard against this type of attack.

The attack relies on a statistical paradox known as the *birthday paradox* – the probability of finding a matching pair in a given set is far greater than that for finding a match for a given individual. Stallings [2005] gives the statistical derivation for the probability that there will be two people with the same birthday in a set of n people. The result is that for a set of only 23 people the chances are even, whereas we require a set of 253 people for an even chance that there will be one with a birthday on a given day.

To satisfy the properties listed above, a secure digest function needs to be carefully designed. The bit-level operations used and their sequencing are similar to those found in symmetric cryptography, but in this case the operations need not be information-preserving, since the function is definitely not intended to be reversible. So a secure digest function can make use of the full range of arithmetic and bit-wise logical operations. The length of the source text is usually included in the digested data.

Figure 11.12 X509 Certificate format

Subject	Distinguished Name, Public Key
Issuer	Distinguished Name, Signature
Period of validity	Not Before Date, Not After Date
Administrative information	Version, Serial Number
Extended information	

Two widely used digest functions for practical applications are the MD5 algorithm (so called because it is the fifth in a sequence of message digest algorithms developed by Ron Rivest) and SHA-1 (the Secure Hash Algorithm), which has been adopted for standardization by the US National Institute for Standards and Technology (NIST). Both have been carefully tested and analyzed and can be considered adequately secure for the foreseeable future, while their implementations are reasonably efficient. We describe them briefly here. Schneier [1996] and Mitchell *et al.* [1992] survey digital signature techniques and message digest functions in depth.

MD5 • The MD5 algorithm [Rivest 1992a] uses four rounds, each applying one of four nonlinear functions to each of 16 32-bit segments of a 512-bit block of source text. The result is a 128-bit digest. MD5 is one of the most efficient algorithms currently available.

SHA-1 • SHA-1 [NIST 2002] is an algorithm that produces a 160-bit digest. It is based on Rivest's MD4 algorithm (which is similar to MD5), with some additional operations. It is substantially slower than MD5, but the 160-bit digest does offer greater security against brute-force and birthday-style attacks. SHA algorithms that deliver longer digests (224, 256 and 512 bits) are also included in the standard [NIST 2002]. Of course, their additional length implies additional costs for the generation, storage and communication of digital signatures and MACs, but following the publication of attacks on SHA-1's predecessors, which suggest that SHA-1 is vulnerable [Randall and Szydlo 2004], NIST announced that is to be superseded by the longer SHA digest versions in US government software by 2010 [NIST 2004].

Using an encryption algorithm to make a digest • It is possible to use a symmetric encryption algorithm such as those detailed in Section 11.3.1 to produce a secure digest. In this case, the key should be published so that the digest algorithm can be applied by anyone wishing to verify a digital signature. The encryption algorithm is used in CBC mode, and the digest is the result of combining the penultimate CBC value with the final encrypted block.

11.4.4 Certificate standards and certificate authorities

X.509 is the most widely used standard format for certificates [CCITT 1988b]. Although the X.509 certificate format is a part of the X.500 standard for the construction of global directories of names and attributes [CCITT 1988a], it is commonly used in cryptographic work as a format definition for freestanding certificates. We describe the X.500 naming standard in Chapter 13.

The structure and content of an X.509 certificate are illustrated in Figure 11.12. As can be seen, it binds a public key to a named entity called a *subject*. The binding is in the signature, which is issued by another named entity called the *issuer*. The certificate has a *period of validity*, which is defined by a pair of dates. The *<Distinguished Name>* entries are intended to be the name of a person, organization or other entity together with sufficient contextual information to render it unique. In a full X.500 implementation this contextual information would be drawn from a directory hierarchy in which the named entity appears, but in the absence of global X.500 implementations it can only be a descriptive string.

This format is included in the TLS protocol for electronic commerce and is widely used in practice to authenticate the public keys of services and their clients. Certain well-known companies and organizations have established themselves to act as *certificate authorities* (for example, Verisign [www.verisign.com] and CREN [www.cren.net]), and other companies and individuals can obtain X.509 public-key certificates from them by submitting satisfactory evidence of their identity. This leads to a two-step verification procedure for any X.509 certificate:

1. Obtain the public-key certificate of the issuer (a certification authority) from a reliable source.
2. Validate the signature.

The SPKI approach • The X.509 approach is based on the global uniqueness of distinguished names. It has been pointed out that this is an impractical goal that does not reflect the reality of current legal and commercial practice [Ellison 1996], in which the identities of individuals are not assumed to be unique but are made unique by reference to other people and organizations. This can be seen in the use of a driving licence or a letter from a bank to authenticate an individual's name and address (a name alone is unlikely to be unique among the world's population). This leads to longer verification chains, because there are many possible issuers of public-key certificates, and their signatures must be validated through a chain of verification that leads back to someone known and trusted by the principal performing the verification. But the resulting verification is likely to be more convincing, and many of the steps in such a chain can be cached to shorten the process on future occasions.

The arguments above are the basis for the recently developed Simple Public-Key Infrastructure (SPKI) proposals (see RFC 2693 [Ellison *et al.* 1999]). This is a scheme for the creation and management of sets of public certificates. It enables chains of certificates to be processed using logical inference to produce derived certificates. For example, 'Bob believes that Alice's public key is K_{Apub}' and 'Carol trusts Bob on Alice's keys' implies 'Carol believes that Alice's public key is K_{Apub}'.

11.5 Cryptography pragmatics

In Section 11.5.1, we compare the performance of the encryption and secure hash algorithms described or mentioned above. We consider encryption algorithms alongside secure hash functions because encryption can also be used as a method for digital signing.

Figure 11.13 Performance of symmetric encryption and secure digest algorithms

	Key size/hash size (bits)	*PRB optimized 90 MHz Pentium 1 (Mbytes/s)*	*Crypto++ 2.1 GHz Pentium 4 (Mbytes/s)*
TEA	128	–	23.801
DES	56	2.113	21.340
Triple-DES	112	0.775	9.848
IDEA	128	1.219	18.963
AES	128	–	61.010
AES	192	–	53.145
AES	256	–	48.229
MD5	128	17.025	216.674
SHA-1	160	–	67.977

In Section 11.5.2, we discuss some non-technical issues surrounding the use of cryptography. There is not space to do justice to the vast amount of political discussion that has taken place on this subject since strong cryptographic algorithms first appeared in the public domain, nor have the debates yet reached many definitive conclusions. Our aim is merely to give the reader some awareness of this ongoing debate.

11.5.1 Performance of cryptographic algorithms

Figure 11.13 compares the speeds of the symmetric encryption algorithms and secure digest functions that we have discussed in this chapter. Where available, we give two speed measurements. In the column labelled *PRB optimized* we give figures based on those published by Preneel *et al.* [1998]. The figures in the column labelled *Crypto++* were obtained much more recently by the authors of the Crypto++ open source library of cryptographic schemes [www.cryptopp.com]. The column headings indicate the speed of the hardware used for these benchmarks. The PRB implementations were hand-optimized assembler programs whereas the Crypto++ ones were C++ programs generated with an optimizing compiler.

The key lengths give an indication of computational cost of a brute-force attack on the key; the true strength of cryptographic algorithms is much more difficult to evaluate and rests on reasoning about the success of the algorithm in obscuring the plain text. Preneel *et al.* provide a useful discussion on the strength and performance of the main symmetric algorithms.

What do these performance figures signify for real applications of cryptography, such as their use in the TLS scheme for secure web interactions (the *https* protocol, described in Section 11.6.3)? Web pages are seldom larger than 100 kilobytes, so the contents of a page can be encrypted using any of the symmetric algorithms in a few milliseconds, even with a processor that is quite slow by today's standards. RSA is used primarily for digital signatures, and that step can also be performed in a few milliseconds. Thus the impact of algorithm performance on the perceived speed of the *https* application is minimal.

Asymmetric algorithms such as RSA are seldom used for data encryption, but their performance for signing is of interest. The Crypto++ library pages indicate that with the hardware mentioned in the last column of Figure 11.13 it takes about 4.75 ms using RSA with a 1024-bit key to sign a secure hash (presumably using 160-bit SHA-1) and about 0.18 ms to verify the signature.

11.5.2 Applications of cryptography and political obstacles

The algorithms described above all emerged during the 1980s and 1990s, when computer networks were beginning to be used for commercial purposes and it was becoming evident that their lack of security was a major problem. As we mentioned in the introduction to this chapter, the emergence of cryptographic software was strongly resisted by the US government. The resistance had two sources: the US National Security Agency (NSA), which was thought to have a policy to restrict the strength of cryptography available to other nations to a level at which the NSA could decrypt any secret communication for military intelligence purposes; and the US Federal Bureau of Investigation (FBI) which aimed to ensure that its agents could have privileged access to the cryptographic keys used by all private organizations and individuals in the US for law-enforcement purposes.

Cryptographic software was classified as a munition in the United States and was subject to stringent export restrictions. Other countries, especially allies of the US, applied similar (or in some cases even more stringent) restrictions. The problem was compounded by the general ignorance among politicians and the general public as to what cryptographic software was and its potential non-military applications. US software companies protested that the restrictions were inhibiting the export of software such as browsers, and the export restrictions were eventually formulated in a form that allowed the export of code using keys of no more than 40 bits – hardly strong cryptography!

The export restrictions may have hindered the growth of electronic commerce, but they were not particularly effective in preventing the spread of cryptographic expertise or in keeping cryptographic software out of the hands of users in other countries, since many programmers inside and outside the US were eager and able to implement and distribute cryptographic code. The current position is that software that implements most of the major cryptographic algorithms has been available world-wide for several years, in print [Schneier 1996] and online, in commercial and freeware versions [www.rsasecurity.com I, cryptography.org, privacy.nb.ca, www.openssl.org].

An example is the program called PGP (Pretty Good Privacy) [Garfinkel 1994, Zimmermann 1995], originally developed by Philip Zimmermann and carried forward by him and others. This is part of a technical and political campaign to ensure that the availability of cryptographic methods is not controlled by the US government. PGP has been developed and distributed with the aim of enabling all computer users to enjoy the level of privacy and integrity afforded by the use of public-key cryptography in their communications. PGP generates and manages public and secret keys on behalf of a user. It uses RSA public-key encryption for authentication and to transmit secret keys to the intended communication partner, and it uses the IDEA or 3DES secret-key encryption algorithms to encrypt mail messages and other documents. (At the time PGP was first developed, use of the DES algorithm was controlled by the US government.) PGP is

widely available in both free and commercial versions. It is distributed via separate distribution sites for North American users [www.pgp.com] and those in other parts of the world [International PGP] to circumvent (perfectly legally) the US export regulations.

The US government eventually recognized the futility of the NSA's position and the harm that it was causing to the US computer industry (which was unable to market secure versions of web browsers, distributed operating systems and many other products world-wide). In January 2000 the US government introduced a new policy [www.bxa.doc.gov] intended to allow US software vendors to export software that incorporates strong encryption. But a legal bar was retained on delivery to certain countries and end-users, see www.rsa.com for further information. Of course, the US does not have a monopoly on the production or the publication of cryptographic software; open source implementations are available for all the well-known algorithms [www.cryptopp.com]. The effect of the regulations is simply to hamper the marketing of some US-produced commercial software products.

Other political initiatives have aimed to maintain control over the use of cryptography by introducing legislation insisting on the inclusion of loopholes or trap doors available only to government law-enforcement and security agencies. Such proposals spring from the perception that secret communication channels can be very useful to criminals of all sorts. Before the advent of digital cryptography, governments always had the means to intercept and analyze communications between members of the public. Strong digital cryptography radically alters that situation. But these proposals to legislate to prevent the use of strong, uncompromized cryptography have been strongly resisted by citizens and civil liberties bodies, who are concerned about their impact on citizens' privacy rights. So far, none of these legislative proposals has been adopted, but political efforts are continuing and the eventual introduction of a legal framework for the use of cryptography may be inevitable.

11.6 Case studies: Needham–Schroeder, Kerberos, TLS, 802.11 WiFi

The authentication protocols originally published by Needham and Schroeder [1978] are at the heart of many security techniques. We present them in detail in Section 11.6.1. One of the most important applications of their secret-key authentication protocol is the Kerberos system [Neuman and Ts'o 1994], which is the subject of our second case study (Section 11.6.2). Kerberos was designed to provide authentication between clients and servers in networks that form a single management domain (intranets).

Our third case study (Section 11.6.3) deals with the Transport Layer Security (TLS) protocol. This was designed specifically to meet the need for secure electronic transactions. It is now supported by most web browsers and servers and is employed in most of the commercial transactions that take place via the Web.

Our final case study (Section 11.6.4) illustrates the difficulty of engineering secure systems. The IEEE 802.11 WiFi standard was published in 1999 with a security specification included. But subsequent analysis and attacks have shown the specification to be severely inadequate. We identify the weaknesses and relate them to the cryptographic principles covered in this chapter.

11.6.1 The Needham–Schroeder authentication protocol

The protocols described here were developed in response to the need for a secure means to manage keys (and passwords) in a network. At the time the work was published [Needham and Schroeder 1978], network file services were just emerging and there was an urgent need for better ways to manage security in local networks.

In networks that are integrated for management purposes, this need can be met by a secure key service that issues session keys in the form of challenges (see Section 11.2.2). That is the purpose of the secret-key protocol developed by Needham and Schroeder. In the same paper, Needham and Schroeder also set out a protocol based on the use of public keys for authentication and key distribution that does not depend upon the existence of secure key servers and is hence more suitable for use in networks with many independent management domains, such as the Internet. We do not describe the public-key version here, but the TLS protocol described in Section 11.6.3 is a variation of it.

Needham and Schroeder proposed a solution to authentication and key distribution based on an *authentication server* that supplies secret keys to clients. The job of the authentication server is to provide a secure way for pairs of processes to obtain shared keys. To do this, it must communicate with its clients using encrypted messages.

Needham and Schroeder with secret keys • In their model, a process acting on behalf of a principal A that wishes to initiate secure communication with another process acting on behalf of a principal B can obtain a key for this purpose. The protocol is described for two arbitrary processes A and B, but in client-server systems, A is likely to be a client initiating a sequence of requests to some server B. The key is supplied to A in two forms: one that A can use to encrypt the messages that it sends to B and one that it can transmit securely to B. (The latter is encrypted in a key that is known to B but not to A, so that B can decrypt it and the key is not compromised during transmission.)

The authentication server S maintains a table containing a name and a secret key for each principal known to the system. The secret key is used only to authenticate client processes to the authentication server and to transmit messages securely between client processes and the authentication server. It is never disclosed to third parties and it is transmitted across the network at most once, when it is generated. (Ideally, a key should always be transmitted by some other means, such as on paper or in a verbal message, avoiding any exposure on the network.) A secret key is the equivalent of the password used to authenticate users in centralized systems. For human principals, the name held by the authentication service is their username and the secret key is their password. Both are supplied by the user on request to client processes acting on the user's behalf.

The protocol is based on the generation and transmission of tickets by the authentication server. A ticket is an encrypted message containing a secret key for use in communication between A and B. We tabulate the messages in the Needham and Schroeder secret-key protocol in Figure 11.14. The authentication server is S.

N_A and N_B are *nonces*. A nonce is an integer value that is added to a message to demonstrate its freshness. Nonces are used only once and are generated on demand. For example, the nonces may be generated as a sequence of integer values or by reading the clock at the sending machine.

If the protocol is successfully completed, both A and B can be sure that any message encrypted in K_{AB} that they receive comes from the other, and that any message

Figure 11.14 The Needham–Schroeder secret-key authentication protocol

Header	*Message*	*Notes*
1. A → S:	A, B, N_A	A requests S to supply a key for communication with B.
2. S → A:	$\{N_A, B, K_{AB}, \{K_{AB}, A\}_{K_B}\}_{K_A}$	S returns a message encrypted in A's secret key, containing a newly generated key K_{AB}, and a 'ticket' encrypted in B's secret key. The nonce N_A demonstrates that the message was sent in response to the preceding one. A believes that S sent the message because only S knows A's secret key.
3. A → B:	$\{K_{AB}, A\}_{K_B}$	A sends the ticket to B.
4. B → A:	$\{N_B\}_{K_{AB}}$	B decrypts the ticket and uses the new key, K_{AB}, to encrypt another nonce, N_B.
5. A → B:	$\{N_B - 1\}_{K_{AB}}$	A demonstrates to B that it was the sender of the previous message by returning an agreed transformation of N_B.

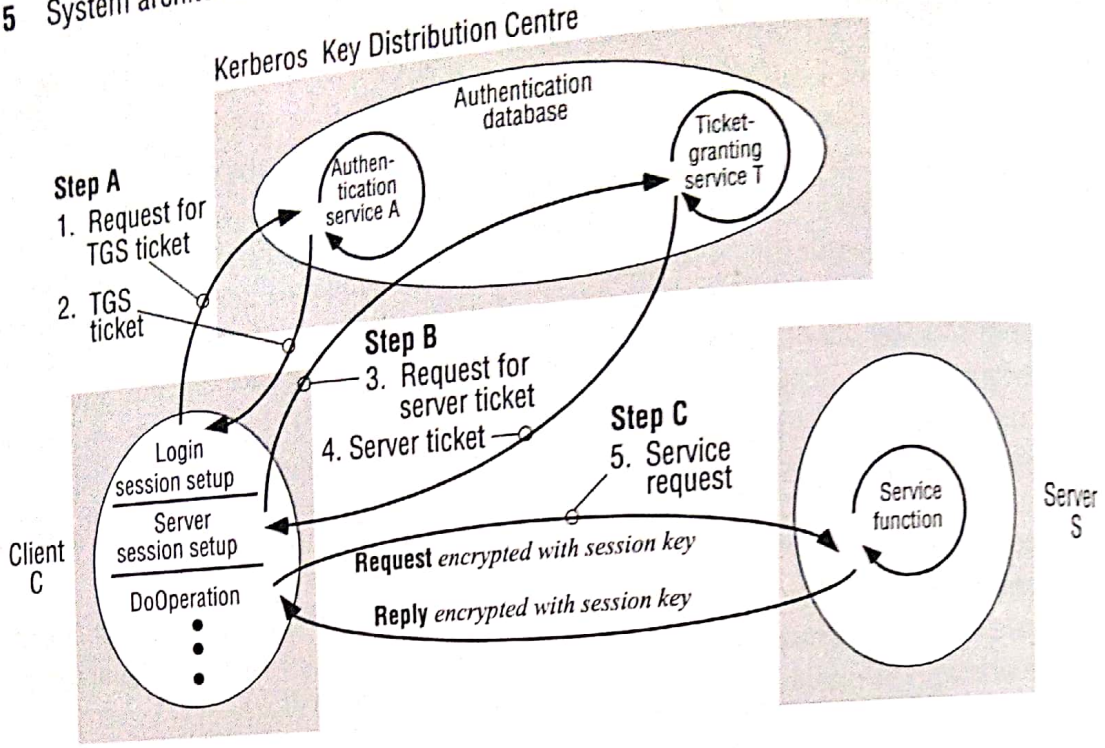
encrypted in K_{AB} that they send can be understood only by the other or by S (and S is assumed to be trustworthy). This is so because the only messages that have been sent containing K_{AB} were encrypted in A's secret key or B's secret key.

There is a weakness in this protocol in that B has no reason to believe that message 3 is fresh. An intruder who manages to obtain the key K_{AB} and make a copy of the ticket and authenticator $\{K_{AB}, A\}_{K_B}$ (both of which might have been left in an exposed storage location by a careless or a failed client program running under A's authority) can use them to initiate a subsequent exchange with B, impersonating A. For this attack to occur an old value of K_{AB} has to be compromised; in today's terminology, Needham and Schroeder did not include this possibility on their threat list, and the consensus of opinion is that one should do so. The weakness can be remedied by adding a nonce or timestamp to message 3, so that it becomes: $\{K_{AB}, A, t\}_{K_{Bpub}}$. B decrypts this message and checks that t is recent. This is the solution adopted in Kerberos.

11.6.2 Kerberos

Kerberos was developed at MIT in the 1980s [Steiner *et al.* 1988] to provide a range of authentication and security facilities for use in the campus computing network at MIT and other intranets. It has undergone several revisions and enhancements in the light of experience and feedback from user organizations. Kerberos version 5 [Neuman and Ts'o 2005]) and is used by many companies and organizations. Source code for an implementation of Kerberos is available from MIT [web.mit.edu I]; it is included in the OSF Distributed Computing Environment (DCE) [OSF 1997] and as the default authentication service in Microsoft Windows [www.microsoft.com II]. An extension was included to incorporate the use of public-key certificates for the initial

Figure 11.15 System architecture of Kerberos



authentication of principals (Step A in Figure 11.15) [Neuman *et al.* 1999].

Figure 11.15 shows the process architecture. Kerberos deals with three kinds of security object:

Ticket: A token issued to a client by the Kerberos ticket-granting service for presentation to a particular server, verifying that the sender has recently been authenticated by Kerberos. Tickets include an expiry time and a newly generated session key for use by the client and the server.

Authenticator: A token constructed by a client and sent to a server to prove the identity of the user and the currency of any communication with a server. An authenticator can be used only once. It contains the client's name and a timestamp and is encrypted in the appropriate session key.

Session key: A secret key randomly generated by Kerberos and issued to a client for use when communicating with a particular server. Encryption is not mandatory for all communication with servers; the session key is used for encrypting communication with those servers that demand it and for encrypting all authenticators (see above).

Client processes must possess a ticket and a session key for each server that they use. It would be impractical to supply a new ticket and key for each client-server interaction, so most tickets are granted to clients with a lifetime of several hours so that they can be used for interaction with a particular server until they expire.

A Kerberos server is known as a Key Distribution Centre (KDC). Each KDC offers an Authentication Service (AS) and a Ticket-Granting Service (TGS). On login, users are authenticated by the AS, using a network-secure variation of the password

method, and the client process acting on behalf of the user is supplied with a *ticket-granting ticket* and a session key for communicating with the TGS. Subsequently, the original client process and its descendants can use the ticket-granting ticket to obtain tickets and session keys for specific services from the TGS.

The Needham and Schroeder [1978] protocol is followed quite closely in Kerberos, with time values (integers representing a date and time) used as nonces. This serves two purposes:

- to guard against replay of old messages intercepted in the network or the reuse of old tickets found lying in the memory of machines from which the authorized user has logged out (nonces were used to achieve this purpose in Needham and Schroeder);
- to apply a lifetime to tickets, enabling the system to revoke users' rights when, for example, they cease to be authorized users of the system.

Below we describe the Kerberos protocols in detail, using the notation defined at the bottom of the page. First, we describe the protocol by which the client obtains a ticket and a session key for access to the TGS.

A Kerberos ticket has a fixed period of validity starting at time t_1 and ending at time t_2. A ticket for a client C to access a server S takes the form:

$\{C, S, t_1, t_2, K_{CS}\}_{K_S}$, which we denote as $\{ticket(C, S)\}_{K_S}$

The client's name is included in the ticket to avoid possible use by impostors, as we shall see later. The step and message numbers in Figure 11.15 correspond to those in tabulated description *A*. Note that message 1 is not encrypted and does not include C's password. It contains a nonce that is used to check the validity of the reply.

A.	*Obtain Kerberos session key and TGS ticket, once per login session*	
Header	*Message*	*Notes*
1. C → A: Request for TGS ticket	C, T, n	Client C requests the Kerberos authentication server A to supply a ticket for communication with the ticket-granting service T.
2. A → C: TGS session key and ticket	$\{K_{CT}, n\}_{K_C}$, $\{ticket(C,T)\}_{K_T}$ containing C, T, t_1, t_2, K_{CT}	A returns a message containing a ticket encrypted in its secret key and a session key for C to use with T. The inclusion of the nonce n encrypted in K_C shows that the message comes from the recipient of message 1, who must know K_C.

Notation:

A	Name of Kerberos authentication service.	n	A nonce.
T	Name of Kerberos ticket-granting service.	t	A timestamp.
C	Name of client.	t_1	Starting time for validity of ticket.
		t_2	Ending time for validity of ticket.

Message 2 is sometimes called a 'challenge' because it presents the requester with information that is only useful if it knows C's secret key, K_C. An impostor who attempts to impersonate C by sending message 1 can get no further, since they cannot decrypt message 2. For principals that are users, K_C is a scrambled version of the user's password. The client process will prompt the user to type their password and will attempt to decrypt message 2 with it. If the user gives the right password, the client process obtains the session key K_{CT} and a valid ticket for the ticket-granting service; if not, it obtains gibberish. Servers have secret keys of their own, known only to the relevant server process and to the authentication server.

When a valid ticket has been obtained from the authentication service, the client C can use it to communicate with the ticket-granting service to obtain tickets for other servers any number of times until the ticket expires. Thus to obtain a ticket for any server S, C constructs an authenticator encrypted in K_{CT} of the form:

$\{C, t\}_{K_{CT}}$, which we denote as $\{auth(C)\}_{K_{CT}}$, and sends a request to T:

B.	*Obtain ticket for a server S, once per client-server session*	
3. C → T: Request ticket for service S	$\{auth(C)\}_{K_{CT}}$, $\{ticket(C,T)\}_{K_T}$, S, n	C requests the ticket-granting server T to supply a ticket for communication with another server, S.
4. T → C: Service ticket	$\{K_{CS}, n\}_{K_{CT}}$, $\{ticket(C,S)\}_{K_S}$	T checks the ticket. If it is valid T generates a new random session key, K_{CS}, and returns it with a ticket for S (encrypted in the server's secret key, K_S).

C is then ready to issue request messages to the server, S:

C.	*Issue a server request with a ticket*	
5. C → S: Service request	$\{auth(C)\}_{K_{CS}}$, $\{ticket(C,S)\}_{K_S}$, *request*, n	C sends the ticket to S with a newly generated authenticator for C and a request. The request would be encrypted in K_{CS} if secrecy of the data is required.

For the client to be sure of the server's authenticity, S should return the nonce n to C (to reduce the number of messages required, this could be included in the messages that contain the server's reply to the request):

D.	*Authenticate server (optional)*	
6. S → C: Server authentication	$\{n\}_{K_{CS}}$	(Optional): S sends the nonce to C, encrypted in K_{CS}.

Application of Kerberos • Kerberos was developed for use in Project Athena at MIT – a campus-wide networked computing facility for undergraduate education with many workstations and servers providing a service to more than 5000 users. The environment is such that neither the trustworthiness of clients nor the security of the network and the machines that offer network services can be assumed – for example, workstations are not protected against the installation of user-developed system software, and server machines (other than the Kerberos server) are not necessarily secured against physical interference with their software configuration.

Kerberos provides virtually all of the security in the Athena system. It is used to authenticate users and other principals. Most of the servers running on the network have been extended to require a ticket from each client at the start of every client-server interaction. These include file storage (NFS and Andrew File System), electronic mail, remote login and printing. Users' passwords are known only to the user and to the Kerberos authentication service. Services have secret keys that are known only to Kerberos and the servers that provide the service.

We describe here the way in which Kerberos is applied to the authentication of users on login. Its use to secure the NFS file service is described in Chapter 12.

Login with Kerberos • When a user logs into a workstation, the login program sends the user's name to the Kerberos authentication service. If the user is known to the authentication service, it replies with a session key, a nonce encrypted in the user's password and a ticket for the TGS. The login program then attempts to decrypt the session key and the nonce using the password that the user typed in response to the password prompt. If the password is correct, the login program obtains the session key and the nonce. It checks the nonce and stores the session key with the ticket for subsequent use when communicating with the TGS. At this point, the login program can erase the user's password from its memory, since the ticket now serves to authenticate the user. A login session is then started for the user on the workstation. Note that the user's password is never exposed to eavesdropping on the network – it is retained in the workstation and is erased from memory soon after it is entered.

Accessing servers with Kerberos • Whenever a program running on a workstation needs to access a new service, it requests a ticket for the service from the ticket-granting service. For example, when a UNIX user wishes to log into a remote computer, the *rlogin* command program on the user's workstation obtains a ticket from the Kerberos ticket-granting service for access to the *rlogind* network service. The *rlogin* command program sends the ticket, together with a new authenticator, in a request to the *rlogind* process on the computer where the user wishes to log in. The *rlogind* program decrypts the ticket with the *rlogin* service's secret key and checks the validity of the ticket (that is, that the ticket's lifetime has not expired). Server machines must take care to store their secret keys in storage that is inaccessible to intruders.

The *rlogind* program then uses the session key included in the ticket to decrypt the authenticator and checks that the authenticator is fresh (authenticators can be used only once). Once the *rlogind* program is satisfied that the ticket and authenticator are valid, there is no need for it to check the user's name and password, because the user's identity is known to the *rlogind* program and a login session is established for that user on the remote machine.

Implementation of Kerberos • Kerberos is implemented as a server that runs on a secure machine. A set of libraries is provided for use by client applications and services. The DES encryption algorithm is used, but this is implemented as a separate module that can be easily replaced.

The Kerberos service is scalable – the world is divided into separate domains of authentication authority, called *realms*, each with its own Kerberos server. Most principals are registered in just one realm, but the Kerberos ticket-granting servers are registered in all of the realms. Principals can authenticate themselves to servers in other realms through their local ticket-granting server.

Within a single realm, there can be several authentication servers, all of which have copies of the same authentication database. The authentication database is replicated by a simple master–slave technique. Updates are applied to the master copy by a single Kerberos Database Management service (KDBM) that runs only on the master machine. The KDBM handles requests from users to change their passwords and requests from system administrators to add or delete principals and to change their passwords.

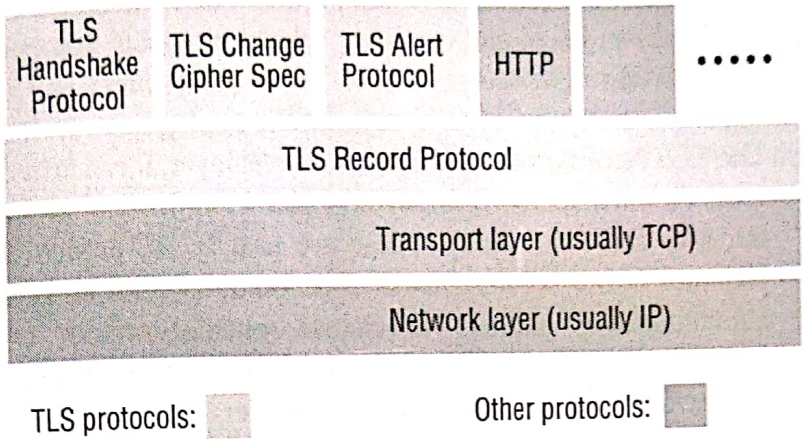
To make this scheme transparent to users, the lifetime of TGS tickets ought to be as long as the longest possible login session, since the use of an expired ticket will result in the rejection of service requests; the only remedy is for the user to reauthenticate the login session and then request new server tickets for all of the services in use. In practice, ticket lifetimes in the region of 12 hours are used.

Critiques of Kerberos • The protocol for Kerberos version 5 described above contains several improvements designed to deal with criticisms of earlier versions [Bellovin and Merritt 1990, Burrows *et al.* 1990]. The most important criticism of version 4 was that the nonces used in authenticators were implemented as timestamps, and protection against the replay of authenticators depended upon at least loose synchronization of clients' and servers' clocks. Furthermore, if a synchronization protocol is used to bring client and server clocks into loose synchrony, the synchronization protocol must itself be secure against security attacks. See Chapter 14 for information on clock synchronization protocols.

The protocol definition for version 5 allows the nonces in authenticators to be implemented as timestamps or as sequence numbers. In both cases, it requires that they be unique and that servers hold a list of recently received nonces from each client to check that they are not replayed. This is an inconvenient implementation requirement and is difficult for servers to guarantee in case of failures. Kehne *et al.* [1992] have published a proposed improvement to the Kerberos protocol that does not rely on synchronized clocks.

The security of Kerberos depends on limited session lifetimes. The period of validity of TGS tickets is generally limited to a few hours; the period must be chosen to be long enough to avoid inconvenient interruptions of service but short enough to ensure that users who have been deregistered or downgraded do not continue to use the resources for more than a short period. This might cause difficulties in some commercial environments, because the consequent requirement for the user to supply a new set of authentication details at an arbitrary point in the interaction might intrude on the application.

Figure 11.16 TLS protocol stack
(Figures 11.16 to 11.19 are based on diagrams in Hirsch [1997] and are used with Frederick Hirsch's permission)



11.6.3 Securing electronic transactions with secure sockets

The Secure Sockets Layer (SSL) protocol was originally developed by the Netscape Corporation [www.mozilla.org] and proposed as a standard specifically to meet the needs described below. An extended version of SSL has been adopted as an Internet standard under the name Transport Layer Security (TLS), described in RFC 2246 [Dierks and Allen 1999]. TLS is supported by most browsers and is widely used in Internet commerce. We explore its main features below:

Negotiable encryption and authentication algorithms • In an open network we should not assume that all parties use the same client software or that all client and server software includes a particular encryption algorithm. In fact, the laws of some countries attempt to restrict the use of certain encryption algorithms to those countries alone. TLS has been designed so that the algorithms used for encryption and authentication are negotiated between the processes at the two ends of the connection during the initial handshake. It may turn out that they do not have sufficient algorithms in common, and in that case the connection attempt will fail.

Bootstrapped secure communication • To meet the need for secure communication without previous negotiation or help from third parties, the secure channel is established using a protocol similar to the hybrid scheme mentioned in Section 11.3.3. Unencrypted communication is used for the initial exchanges, then public-key cryptography and finally secret-key cryptography once a shared secret key has been established. Each switch is optional and preceded by a negotiation.

Thus the secure channel is fully configurable, allowing communication in each direction to be encrypted and authenticated but not requiring it, so that computing resources need not be consumed in performing unnecessary encryption.

The details of the TLS protocol are published and standardized, and several software libraries and toolkits are available to support it [Hirsch 1997, www.openssl.org], some of them in the public domain. It has been incorporated in a wide range of application software, and its security has been verified by independent review.

TLS consists of two layers (Figure 11.16): TLS Record Protocol layer, which implements a secure channel, encrypting and authenticating messages transmitted through any connection-oriented protocol; and a handshake layer, containing the TLS handshake protocol and two other related protocols that establish and maintain a TLS session (that is, a secure channel) between a client and a server. Both are usually implemented by software libraries at the application level in the client and the server. The TLS record protocol is a session-level layer; it can be used to transport application-level data transparently between a pair of processes while guaranteeing its secrecy, integrity and authenticity. These are exactly the properties we specified for secure channels in our security model (Section 2.4.3), but in TLS there are options for the communicating partners to choose whether or not to deploy decryption and authentication of messages in each direction. Each secure session is given an identifier, and each partner can store session identifiers in a cache for subsequent reuse, avoiding the overhead of establishing a new session when another secure session with the same partner is required.

TLS is widely used to add a secure communication layer below existing application-level protocols. It is probably most widely used to secure HTTP interactions for use in Internet commerce and other security-sensitive applications. It is implemented by virtually all web browsers and web servers: the use of the protocol prefix *https:* in URLs initiates the establishment of a TLS secure channel between a browser and a web server. It has also been widely deployed to provide secure implementations of Telnet, FTP and many other application protocols. TLS is the *de facto* standard for use in applications requiring secure channels; there is a wide choice of available implementations, both commercial and public-domain, with APIs for CORBA and Java.

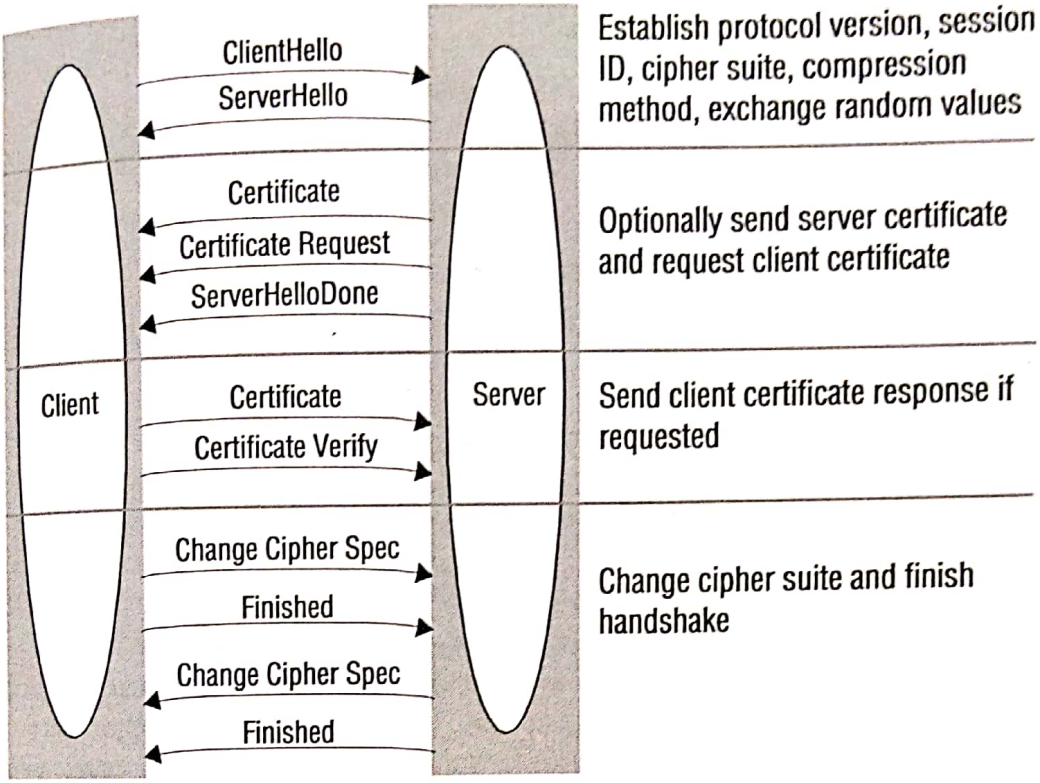
The TLS handshake protocol is illustrated in Figure 11.17. The handshake is performed over an existing connection. It begins in the clear and it establishes a TLS session by exchanging the agreed options and parameters needed to perform encryption and authentication. The handshake sequence varies depending on whether client and server authentication are required. The handshake protocol may also be invoked at a later time to change the specification of a secure channel – for example, communication may begin with message authentication using message authentication codes only, and at a later point, encryption may be added. This is achieved by performing the handshake protocol again to negotiate a new cipher specification using the existing channel.

The TLS initial handshake is potentially vulnerable to man-in-the-middle attacks, as described in Section 11.2.2, Scenario 3. To protect against them, the public key used to verify the first certificate received may be delivered by a separate channel – for example, browsers and other Internet software delivered on a CD-ROM may include a set of public keys for some well-known certificate authorities. Another defence for the clients of well-known services is based on the inclusion of the service's domain name in its public-key certificates – clients should only deal with the service at the IP address corresponding to that domain name.

TLS supports a variety of options for the cryptographic functions to be used. These are collectively known as a *cipher suite*. A cipher suite includes a single choice for each of the features shown in Figure 11.18.

A variety of popular cipher suites are preloaded, with standard identifiers in the client and the server. During the handshake, the server offers the client a list of the cipher suite identifiers that it has available, and the client responds by selecting one of them (or

Figure 11.17 TLS handshake protocol



giving an error indication if it has none that match). At this stage they also agree on a (optional) compression method and a random start value for CBC block encryption functions (see Section 11.3).

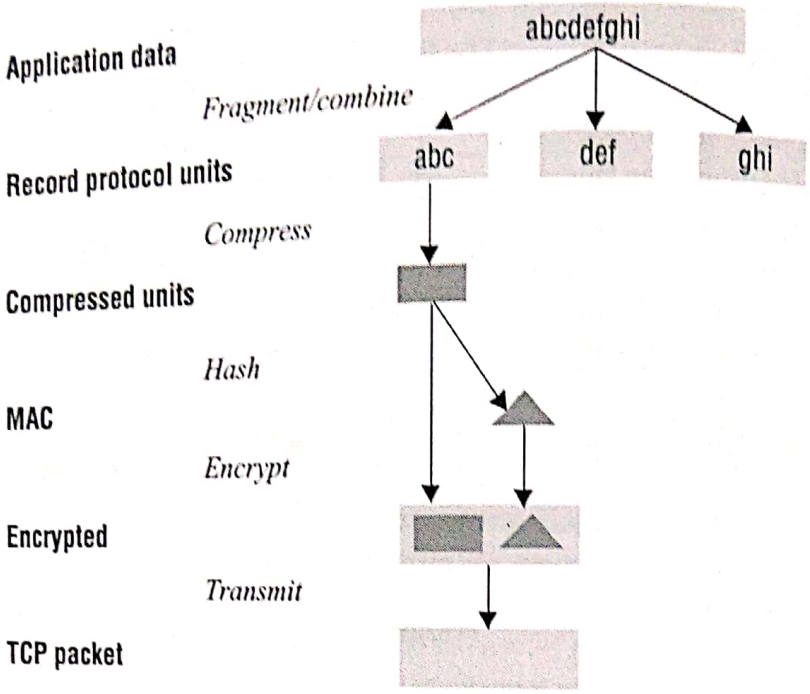
Next, the partners optionally authenticate each other by exchanging signed public-key certificates in X.509 format. These certificates may be obtained from a public-key authority or they may simply be generated temporarily for the purpose. In any case, at least one public key must be available for use in the next stage of the handshake.

One partner then generates a *pre-master secret* and sends it to the other partner encrypted with the public key. A pre-master secret is a large random value that is used by both partners to generate the two session keys (called *write* keys) for encrypting data

Figure 11.18 TLS handshake configuration options

Component	*Description*	*Example*
Key exchange method	The method to be used for exchange of a session key	RSA with public-key certificates
Cipher for data transfer	The block or stream cipher to be used for data	IDEA
Message digest function	For creating message authentication codes (MACs)	SHA-1

Figure 11.19 TLS record protocol



in each direction and the message authentication secrets to be used for message authentication. When all this has been done, a secure session begins. This is triggered by the *ChangeCipherSpec* messages exchanged between the partners. These are followed by *Finished* messages. Once the *Finished* messages have been exchanged, all further communication is encrypted and signed according to the chosen cipher suite with the agreed keys.

Figure 11.19 shows the operation of the record protocol. A message for transmission is first fragmented into blocks of a manageable size, then the blocks are optionally compressed. Compression is not strictly a feature of secure communication, but it is provided here because a compression algorithm can usefully share some of the work of processing the bulk data with the encryption and digital signature algorithms. In other words, a pipeline of data transformations can be set up within the TLS record layer that will perform all of the transformations required more efficiently than could be done independently.

The encryption and message authentication (MAC) transformations deploy the algorithms specified in the agreed cipher suite exactly as described in Sections 11.3.1 and 11.4.2. Finally, the signed and encrypted block is transmitted to the partner through the associated TCP connection, where the transformations are reversed to produce the original data block.

Summary • TLS provides a practical implementation of a hybrid encryption scheme with authentication and key exchange based on public keys. Because the ciphers are negotiated in the handshake, it does not depend upon the availability of any particular algorithms. Nor does it depend upon any secure services at the time of session

establishment. The only requirement is that the public-key certificates are issued by an authority recognized by both parties.

Because the SSL basis for TLS and its reference implementation were published [www.mozilla.org], it was the subject of review and debate. Some amendments were made to the early designs, and it was widely endorsed as a valuable standard. TLS is now integrated in most web browsers and web servers and is used in other applications such as secure Telnet and FTP. Commercial and public-domain [www.rsasecurity.com I, Hirsch 1997, www.openssl.org] implementations are widely available in the form of libraries and browser plug-ins.

11.6.4 Weaknesses in the original IEEE 802.11 WiFi security design

The IEEE 802.11 standard for wireless LANs described in Section 3.5.2 was first released in 1999 [IEEE 1999]. It was implemented in base stations, laptops and portable devices from a similar date and widely used for mobile communication. Unfortunately, the security design in the standard was subsequently found to be severely inadequate in several respects. We outline that initial design and its weaknesses as a case study in the difficulties of security design already referred to in Section 11.1.3.

It was recognized that wireless networks are by their nature more exposed to attack than wired networks because the network and the transmitted data are available for eavesdropping and masquerading by any device equipped with a transmitter/receiver within range. The initial 802.11 design therefore aimed to provide access control for WiFi networks and privacy and integrity for the data transmitted on them through a security specification entitled Wired Equivalent Privacy (WEP), which embodies the following measures, that can optionally be activated by a network administrator:

Access control by a challenge-response protocol (*cf.* Kerberos, Section 11.6.2), in which a joining node is challenged by the base station to demonstrate that it has the correct shared key. A single key, K, is assigned by a network administrator and shared between the base station and all authorized devices.

Privacy and integrity using an optional encryption mechanism based on the RC4 stream cipher. The same key, K, used for access control is also used in encryption. There are key length options of 40, 64 or 128 bits. An encrypted checksum is included in each packet to protect its integrity.

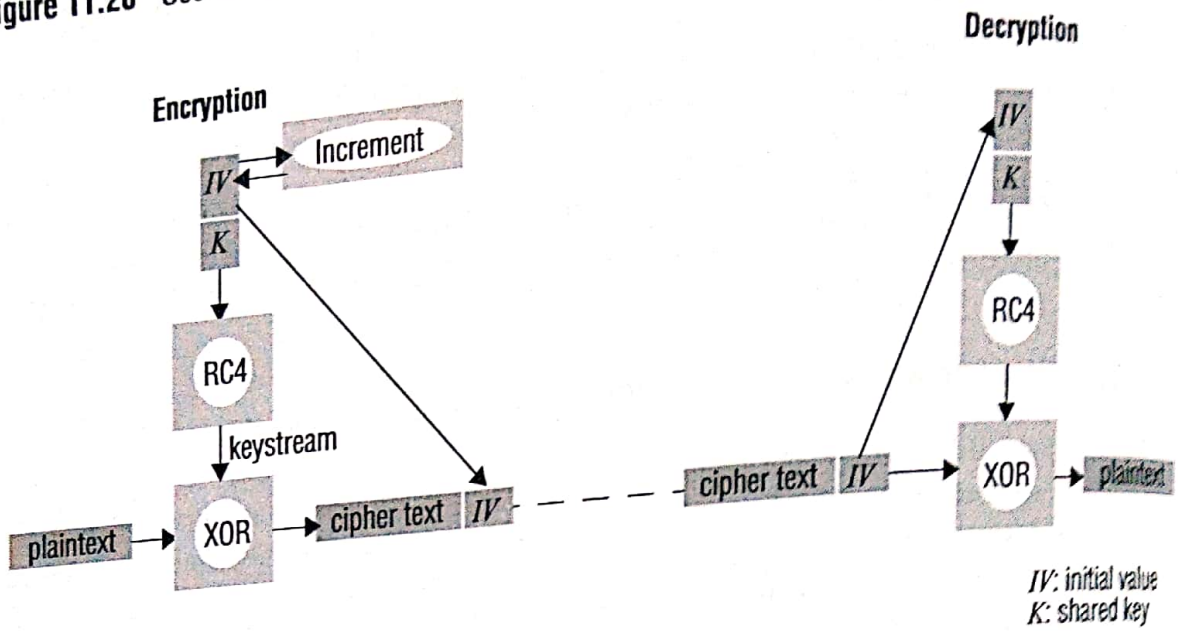
The following deficiencies and design weaknesses were discovered soon after the standard was deployed:

The sharing of a single key by all users of a network renders the design weak in practice, since:

- The key is liable to be transmitted to new users on unprotected channels.
- A single careless or malicious user (such as a disgruntled former employee) who has gained access to the key can compromise the security of the entire network, and this can go undiscovered.

Solution: Use a public-key-based protocol for negotiating individual keys, as is done in TLS/SSL (see Section 11.6.3).

Figure 11.20 Use of RC4 stream cipher in IEEE 802.11 WEP



Base stations are never authenticated, so an attacker who knows the current shared key could introduce a spoof base station and eavesdrop on, insert or tamper with any traffic.

Solution: Base stations should supply a certificate that can be authenticated by the use of a public key obtained from a third party.

Inappropriate use of a stream cipher rather than a block cipher (see descriptions of block and stream ciphers in Section 11.3). Figure 11.20 shows the process of encryption and decyption in 802.11 WEP security. Each packet is encrypted by XOR-ing its content with a keystream produced by the RC4 algorithm. The receiving station uses RC4 to generate the same keystream and decrypts the packet with another XOR. To avoid keystream synchronization errors when packets are lost or corrupted, RC4 is restarted with a start value consisting of a 24-bit *initial value* concatenated with the globally shared key. The initial value is updated and included (in the clear) in each packet transmitted. The shared key cannot easily be changed in normal use, so the starting value has only $s = 2^{24}$ (or about 10^7) different states, resulting in the repetition of the start value and hence the keystream, after 10^7 packets are sent. In practice this can occur within a few hours, and even shorter repetition cycles can arise if packets are lost. An attacker receiving the encrypted packets can always detect repetitions since the initial value is sent in the clear.

The RC4 specification explicitly warns against keystream repetition. This is because an attacker who receives an encrypted packet C_i and knows the plain text P_i (for example, by guessing that it is a standard enquiry to a server) can calculate the keystream K_i used to encrypt the packet. The same value of K_i will recur after s packets are transmitted, and the attacker can use it to decrypt the newly transmitted

packet. The attacker may eventually succeed in decrypting a high proportion of packets in this manner by guessing plaintext packets correctly. This weakness was first pointed out by Borisov *et al.* [2001] and led to a major reappraisal of WEP security and its replacement in later versions of 802.11.

Solution: Negotiate a new key after a time less than the worst case for repetition. An explicit termination code would be needed, as is the case in TLS.

Key lengths of 40 bits and 64 bits were included in the standard to enable products to be shipped abroad by US suppliers at a time when the US government regulations referred to in Section 11.5.2 restricted key lengths for exported devices to 40 bits (and subsequently 64 bits). But 40-bit keys are so easily cracked by brute force that they offer very little security, and even 64-bit keys are potentially crackable with a determined attack.

Solution: Use 128-bit keys only. This has been adopted in many recent WiFi products.

The RC4 stream cipher was shown, after publication of the 802.11 standard, to have weaknesses that enabled the key to be discovered after observation of a substantial quantity of traffic even without repetition of the keystream [Fluhrer *et al.* 2001]. This weakness was demonstrated in practice. It rendered the WEP scheme insecure even with 128-bit keys and led some companies to ban the use of WiFi networks by their employees.

Solution: Provide for the negotiation of cipher specifications as is done in TLS, giving a choice of encryption algorithms. RC4 is hard-wired into the WEP standard, with no provision for the negotiation of encryption algorithms.

Users often didn't deploy the protection offered by the WEP scheme, probably because they didn't realize just how exposed their data was. This was not a weakness in the design of the standard but in the marketing of products based on it. Most were designed to start up with security disabled and their documentation of the security risks was often weak.

Solution: Better default settings and documentation can help. Users want to obtain optimum performance, though, and communication was perceptibly slower with encryption enabled using the hardware available at the time. Attempts to avoid the use of WEP encryption led to the addition to base stations of features for the suppression of the identifying packets normally broadcast by base stations and the rejection of packets not sent from an authorized MAC address (see Section 3.5.1). Neither of these offered much security, since a network can be discovered by intercepting ('sniffing') packets in transmission and MAC addresses can be spoofed by operating system modifications.

IEEE set up a dedicated task group to create a replacement security solution and their work led to a completely new security protocol known as Wi-Fi Protected Access (WPA). This was specified in the IEEE 802.11i draft [IEEE 2004b, Edney and Arbaugh 2003], and started to appear in products in mid-2003. IEEE 802.11i (also known as WPA2) itself was ratified in June 2004, and uses AES encryption, instead of RC4, which was used in WEP. Subsequent developments of IEEE 802.11 also incorporate WPA2.

11.7 Summary

Threats to the security of distributed systems are pervasive. It is essential to protect the communication channels and the interfaces of any system that handles information that could be the subject of attacks. Personal mail, electronic commerce and other financial transactions are all examples of such information. Security protocols are carefully designed to guard against loopholes. The design of secure systems starts from a list of threats and a set of 'worst case' assumptions.

Security mechanisms are based on public-key and secret-key cryptography. Cryptographic algorithms scramble messages in a manner that cannot be reversed without knowledge of the decryption key. Secret-key cryptography is symmetric – the same key serves for both encryption and decryption. If two parties share a secret key, they can exchange encrypted information without risk of eavesdropping or tampering and with guarantees of authenticity.

Public-key cryptography is asymmetric – separate keys are used for encryption and decryption, and knowledge of one does not reveal the other. One key is made public, enabling anyone to send secure messages to the holder of the corresponding private key and allowing the holder of the private key to sign messages and certificates. Certificates can act as credentials for the use of protected resources.

Resources are protected by access-control mechanisms. Access-control schemes assign rights to principals (that is, the holders of credentials) to perform operations on distributed objects and collections of objects. Rights may be held in access control lists (ACLs) associated with collections of objects or they may be held by principals in the form of capabilities – unforgeable keys for access to collections of resources. Capabilities are convenient for the delegation of access rights but are hard to revoke. Changes to ACLs take effect immediately, revoking the previous access rights, but they are more complex and costly to manage than capabilities.

Until recently, the DES encryption algorithm was the most widely used symmetric encryption scheme, but its 56-bit keys are no longer safe against brute-force attacks. The triple version of DES gives 112-bit key strength, which is safe, but other modern algorithms (such as IDEA and AES) are much faster and provide greater strength.

RSA is the most widely used asymmetric encryption scheme. For safety against factoring attacks, it should be used with 768-bit keys or greater. Public-key (asymmetric) algorithms are outperformed by secret-key (symmetric) algorithms by several orders of magnitude, so they are generally used only in hybrid protocols such as TLS, for the establishment of secure channels that use shared keys for subsequent exchanges.

The Needham–Schroeder authentication protocol was the first general-purpose, practical security protocol, and it still provides the basis for many practical systems. Kerberos is a well-designed scheme for the authentication of users and the protection of services within a single organization. Kerberos is based on Needham–Schroeder and symmetric cryptography. TLS is the security protocol designed for and used widely in electronic commerce. It is a flexible protocol for the establishment and use of secure channels based on both symmetric and asymmetric cryptography. The weaknesses of IEEE 802.11 WiFi security provide an object lesson in the difficulties of security design.

EXERCISES

11.1 Describe some of the physical security policies in your organization. Express them in terms that could be implemented in a computerized door locking system. *page 464*

11.2 Describe some of the ways in which conventional email is vulnerable to eavesdropping, masquerading, tampering, replay and denial of service attacks. Suggest methods by which email could be protected against each of these forms of attack. *page 466*

11.3 Briefly discuss the various requirements for Internet vendors and buyers that need to be followed for securing web purchases. *page 470*

11.4 PGP is often used to secure email communication. Describe the steps that a pair of users using PGP must take before they can exchange email messages with privacy and authenticity guarantees. What scope is there to make the preliminary key negotiation invisible to users? (The PGP negotiation is an instance of the hybrid scheme.) *pages 493, 502*

11.5 Kerberos protocol attaches time values with the messages, which are very much similar to nonces in the Needham and Schroeder protocol. What is the purpose of these time values? *page 507*

11.6 The implementation of the TEA symmetric encryption algorithm given in Figures 11.7–11.9 is not portable between all machine architectures. Explain why. How could a message encrypted using the TEA implementation be transmitted to decrypt it correctly on all other architectures? *page 488*

11.7 Modify the TEA application program in Figure 11.9 to use cipher block chaining (CBC). *pages 485, 488*

11.8 Construct a stream cipher application based on the program in Figure 11.9. *pages 486, 488*

11.9 Estimate the time required to crack a 56-bit DES key by a brute-force attack using a 2000 MIPS (million instruction per second) computer, assuming that the inner loop for a brute-force attack program involves around 10 instructions per key value, plus the time to encrypt an 8-byte plaintext (see Figure 11.13). Perform the same calculation for a 128-bit IDEA key. Extrapolate your calculations to obtain the cracking time for a 200,000 MIPS parallel processor (or an Internet consortium with similar processing power). *page 489*

11.10 In the Needham and Shroeder authentication protocol with secret keys, explain why the following version of message 5 is not secure:

$$A \rightarrow B: \quad \{N_B\}_{K_{AB}}$$

page 504

11.11 Review the solutions proposed in the discussion of the 802.11 Wireless Equivalent Privacy protocol design, outlining ways in which each solution could be implemented and mentioning any drawbacks or inconveniences. (5 answers) *page 515*

12

DISTRIBUTED FILE SYSTEMS

A distributed file system enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network. The performance and reliability experienced for access to files stored at a server should be comparable to that for files stored on local disks.

In this chapter we define a simple architecture for file systems and describe two basic distributed file service implementations with contrasting designs that have been in widespread use for over two decades:

- the Sun Network File System, NFS;
- the Andrew File System, AFS.

Each emulates the UNIX file system interface, with differing degrees of scalability, fault tolerance and deviation from the strict UNIX one-copy file update semantics.

Several related file systems that exploit new modes of data organization on disk or across multiple servers to achieve high-performance, fault-tolerant and scalable file systems are also reviewed. Other types of distributed storage system are described elsewhere in the book. These include peer-to-peer storage systems (Chapter 10), replicated file systems (Chapter 18), multimedia data servers (Chapter 20) and the particular style of storage service required to support Internet search and other large-scale, data-intensive applications (Chapter 21).

In Chapters 1 and 2, we identified the sharing of resources as a key goal for distributed systems. The sharing of stored information is perhaps the most important aspect of distributed resource sharing. Mechanisms for data sharing take many forms and are described in several parts of this book. Web servers provide a restricted form of data sharing in which files stored locally, in file systems at the server or in servers on a local network, are made available to clients throughout the Internet. The design of large-scale wide area read-write file storage systems poses problems of load balancing, reliability, availability and security, whose resolution is the goal of the peer-to-peer file storage systems described in Chapter 10. Chapter 18 focuses on replicated storage systems that are suitable for applications requiring reliable access to data stored on systems where the availability of individual hosts cannot be guaranteed. In Chapter 20 we describe a media server that is designed to serve streams of video data to large numbers of users in real time. Chapter 21 describes a file system designed to support large-scale, data-intensive applications such as Internet search.

The requirements for sharing within local networks and intranets lead to a need for a different type of service – one that supports the persistent storage of data and programs of all types on behalf of clients and the consistent distribution of up-to-date data. The purpose of this chapter is to describe the architecture and implementation of these *basic* distributed file systems. We use the word 'basic' here to denote distributed file systems whose primary purpose is to emulate the functionality of a non-distributed file system for client programs running on multiple remote computers. They do not maintain multiple persistent replicas of files, nor do they support the bandwidth and timing guarantees required for multimedia data streaming – those requirements are addressed in later chapters. Basic distributed file systems provide an essential underpinning for organizational computing based on intranets.

File systems were originally developed for centralized computer systems and desktop computers as an operating system facility providing a convenient programming interface to disk storage. They subsequently acquired features such as access-control and file-locking mechanisms that made them useful for the sharing of data and programs. Distributed file systems support the sharing of information in the form of files and hardware resources in the form of persistent storage throughout an intranet. A well-designed file service provides access to files stored at a server with performance and reliability similar to, and in some cases better than, files stored on local disks. Their design is adapted to the performance and reliability characteristics of local networks, and hence they are most effective in providing shared persistent storage for use in intranets. The first file servers were developed by researchers in the 1970s [Birrell and Needham 1980, Mitchell and Dion 1982, Leach *et al.* 1983], and Sun's Network File System became available in the early 1980s [Sandberg *et al.* 1985, Callaghan 1999].

A file service enables programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer in an intranet. The concentration of persistent storage at a few servers reduces the need for local disk storage and (more importantly) enables economies to be made in the management and archiving of the persistent data owned by an organization. Other services, such as the name service, the user authentication service and the print service, can be more easily

Figure 12.1 Storage systems and their properties

	Sharing	*Persistence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	×	×	×	1	RAM
File system	×	✓	×	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Web server
Distributed shared memory	✓	×	✓	✓	Ivy (DSM, Ch. 6)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent State Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

Types of consistency:
1: strict one-copy ✓: slightly weaker guarantees 2: considerably weaker guarantees

implemented when they can call upon the file service to meet their needs for persistent storage. Web servers are reliant on filing systems for the storage of the web pages that they serve. In organizations that operate web servers for external and internal access via an intranet, the web servers often store and access the material from a local distributed file system.?

With the advent of distributed object-oriented programming, a need arose for the persistent storage and distribution of shared objects. One way to achieve this is to serialize objects (in the manner described in Section 4.3.2) and to store and retrieve the serialized objects using files. But this method for achieving persistence and distribution is impractical for rapidly changing objects, so several more direct approaches have been developed. Java remote object invocation and CORBA ORBs provide access to remote, shared objects, but neither of these ensures the persistence of the objects, nor are the distributed objects replicated.

Figure 12.1 provides an overview of types of storage system. In addition to those already mentioned, the table includes distributed shared memory (DSM) systems and persistent object stores. DSM was described in Chapter 6. It provides an emulation of a shared memory by the replication of memory pages or segments at each host, but it does not necessarily provide automatic persistence. Persistent object stores were introduced in Chapter 5. They aim to provide persistence for distributed shared objects. Examples include the CORBA Persistent State Service (see Chapter 8) and persistent extensions to Java [Jordan 1996, java.sun.com VIII]. Some research projects have developed in platforms that support the automatic replication and persistent storage of objects (for example, PerDiS [Ferreira *et al.* 2000] and Khazana [Carter *et al.* 1998]). Peer-to-peer storage systems offer scalability to support client loads much larger than the systems described in this chaper, but they incur high performance costs in providing secure access control and consistency between updatable replicas.

Figure 12.2 File system modules

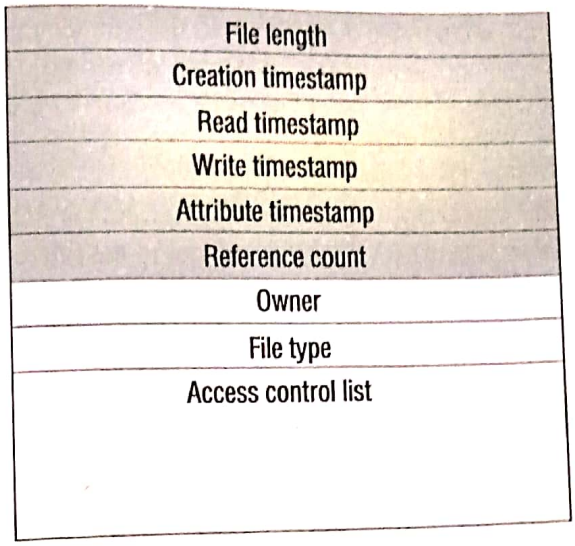
Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	performs disk I/O and buffering

The *consistency* column indicates whether mechanisms exist for the maintenance of consistency between multiple copies of data when updates occur. Virtually all storage systems rely on the use of caching to optimize the performance of programs. Caching was first applied to main memory and non-distributed file systems, and for those the consistency is strict (denoted by a '1', for one-copy consistency in Figure 12.1) – programs cannot observe any discrepancies between cached copies and stored data after an update. When distributed replicas are used, strict consistency is more difficult to achieve. Distributed file systems such as Sun NFS and the Andrew File System cache copies of portions of files at client computers, and they adopt specific consistency mechanisms to maintain an approximation to strict consistency – this is indicated by a tick (✓) in the consistency column of Figure 12.1. We discuss these mechanisms and the degree to which they deviate from strict consistency in Sections 12.3 and 12.4.

The Web uses caching extensively both at client computers and at proxy servers maintained by user organizations. The consistency between the copies stored at web proxies and client caches and the original server is only maintained by explicit user actions. Clients are not notified when a page stored at the original server is updated; they must perform explicit checks to keep their local copies up-to-date. This serves the purposes of web browsing adequately, but it does not support the development of cooperative applications such as a shared distributed whiteboard. The consistency mechanisms used in DSM systems are discussed in depth on the companion web site to the book [www.cdk5.net]. Persistent object systems vary considerably in their approach to caching and consistency. The CORBA and Persistent Java schemes maintain single copies of persistent objects, and remote invocation is required to access them, so the only consistency issue is between the persistent copy of an object on disk and the active copy in memory, which is not visible to remote clients. The PerDiS and Khazana projects that we mentioned above maintain cached replicas of objects and employ quite elaborate consistency mechanisms to produce forms of consistency similar to those found in DSM systems.

Having introduced some wider issues relating to storage and distribution of persistent and non-persistent data, we now return to the main topic of this chapter – the design of basic distributed file systems. We describe some relevant characteristics of (non-distributed) file systems in Section 12.1.1 and the requirements for distributed file systems in Section 12.1.2. Section 12.1.3 introduces the case studies that will be used throughout the chapter. In Section 12.2, we define an abstract model for a basic

Figure 12.3 File attribute record structure



distributed file service, including a set of programming interfaces. Sun NFS is described in Section 12.3; it shares many of the features of the abstract model. In Section 12.4 we describe the Andrew File System, a widely used system that employs substantially different caching and consistency mechanisms. Section 12.5 reviews some recent developments in the design of file services.

The systems described in this chapter do not cover the full spectrum of distributed file and data management systems. Several systems with more advanced characteristics will be described later in the book. Chapter 18 includes a description of Coda, a distributed file system that maintains persistent replicas of files for reliability, availability and disconnected working. Bayou, a distributed data management system that provides a weakly consistent form of replication for high availability, is also covered in Chapter 18. Chapter 20 covers the Tiger video file server, which is designed to provide timely delivery of streams of data to large numbers of clients. Chapter 21 describes the Google File System (GFS), a file system designed specifically to support large-scale, data-intensive applications including Internet search.

12.1.1 Characteristics of file systems

File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files. They provide a programming interface that characterizes the file abstraction, freeing programmers from concern with the details of storage allocation and layout. Files are stored on disks or other non-volatile storage media.

Files contain both *data* and *attributes*. The data consist of a sequence of data items (typically 8-bit bytes), accessible by operations to read and write any portion of the sequence. The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists. A typical attribute record structure is illustrated in Figure 12.3. The shaded attributes are managed by the file system and are not normally updatable by user programs.

Figure 12.4 UNIX file system operations

filedes = open(name, mode) *filedes = creat(name, mode)*	Opens an existing file with the given *name*. Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = close(filedes)	Closes the open file *filedes*.
count = read(filedes, buffer, n) *count = write(filedes, buffer, n)*	Transfers *n* bytes from the file referenced by *filedes* to *buffer*. Transfers *n* bytes to the file referenced by *filedes* from *buffer*. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to *offset* (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*).
status = stat(name, buffer)	Puts the file attributes for file *name* into *buffer*.

File systems are designed to store and manage large numbers of files, with facilities for creating, naming and deleting files. The naming of files is supported by the use of directories. A *directory* is a file, often of a special type, that provides a mapping from text names to internal file identifiers. Directories may include the names of other directories, leading to the familiar hierarchic file-naming scheme and the multi-part *pathnames* for files used in UNIX and other operating systems. File systems also take responsibility for the control of access to files, restricting access to files according to users' authorizations and the type of access requested (reading, updating, executing and so on).

The term *metadata* is often used to refer to all of the extra information stored by a file system that is needed for the management of files. It includes file attributes, directories and all the other persistent information used by the file system.

Figure 12.2 shows a typical layered module structure for the implementation of a non-distributed file system in a conventional operating system. Each layer depends only on the layers below it. The implementation of a distributed file service requires all of the components shown there, with additional components to deal with client-server communication and with the distributed naming and location of files.

File system operations • Figure 12.4 summarizes the main operations on files that are available to applications in UNIX systems. These are the system calls implemented by the kernel; application programmers usually access them through procedure libraries such as the C Standard Input/Output Library or the Java file classes. We give the primitives here as an indication of the operations that file services are expected to support and for comparison with the file service interfaces that we shall introduce below.

The UNIX operations are based on a programming model in which some file state information is stored by the file system for each running program. This consists of a list of currently open files with a read-write pointer for each, giving the position within the file at which the next read or write operation will be applied.

The file system is responsible for applying access control for files. In local file systems such as UNIX, it does so when each file is opened, checking the rights allowed for the user's identity in the access control list against the *mode* of access requested in the *open* system call. If the rights match the mode, the file is opened and the *mode* is recorded in the open file state information.

12.1.2 Distributed file system requirements

Many of the requirements and potential pitfalls in the design of distributed services were first observed in the early development of distributed file systems. Initially, they offered access transparency and location transparency; performance, scalability, concurrency control, fault tolerance and security requirements emerged and were met in subsequent phases of development. We discuss these and related requirements in the following subsections.

Transparency • The file service is usually the most heavily loaded service in an intranet, so its functionality and performance are critical. The design of the file service should support many of the transparency requirements for distributed systems identified in Section 1.5.7. The design must balance the flexibility and scalability that derive from transparency against software complexity and performance. The following forms of transparency are partially or wholly addressed by current file services:

Access transparency: Client programs should be unaware of the distribution of files. A single set of operations is provided for access to local and remote files. Programs written to operate on local files are able to access remote files without modification.

Location transparency: Client programs should see a uniform file name space. Files or groups of files may be relocated without changing their pathnames, and user programs see the same name space wherever they are executed.

Mobility transparency: Neither client programs nor system administration tables in client nodes need to be changed when files are moved. This allows file mobility – files or, more commonly, sets or volumes of files may be moved, either by system administrators or automatically.

Performance transparency: Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.

Scaling transparency: The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.

Concurrent file updates • Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file. This is the well-known issue of concurrency control, discussed in detail in Chapter 16. The need for concurrency control for access to shared data in many applications is widely accepted and techniques are known for its implementation, but they are costly. Most current file

services follow modern UNIX standards in providing advisory or mandatory file- or record-level locking.

File replication • In a file service that supports replication, a file may be represented by several copies of its contents at different locations. This has two benefits – it enables multiple servers to share the load of providing a service to clients accessing the same set of files, enhancing the scalability of the service, and it enhances fault tolerance by enabling clients to locate another server that holds a copy of the file when one has failed. Few file services support replication fully, but most support the caching of files or portions of files locally, a limited form of replication. The replication of data is discussed in Chapter 18, which includes a description of the Coda replicated file service.

Hardware and operating system heterogeneity • The service interfaces should be defined so that client and server software can be implemented for different operating systems and computers. This requirement is an important aspect of openness.

Fault tolerance • The central role of the file service in distributed systems makes it essential that the service continue to operate in the face of client and server failures. Fortunately, a moderately fault-tolerant design is straightforward for simple servers. To cope with transient communication failures, the design can be based on *at-most-once* invocation semantics (see Section 5.3.1); or it can use the simpler *at-least-once* semantics with a server protocol designed in terms of *idempotent* operations, ensuring that duplicated requests do not result in invalid updates to files. The servers can be *stateless*, so that they can be restarted and the service restored after a failure without any need to recover previous state. Tolerance of disconnection or server failures requires file replication, which is more difficult to achieve and will be discussed in Chapter 18.

Consistency • Conventional file systems such as that provided in UNIX offer *one-copy update semantics*. This refers to a model for concurrent access to files in which the file contents seen by all of the processes accessing or updating a given file are those that they would see if only a single copy of the file contents existed. When files are replicated or cached at different sites, there is an inevitable delay in the propagation of modifications made at one site to all of the other sites that hold copies, and this may result in some deviation from one-copy semantics.

Security • Virtually all file systems provide access-control mechanisms based on the use of access control lists. In distributed file systems, there is a need to authenticate client requests so that access control at the server is based on correct user identities and to protect the contents of request and reply messages with digital signatures and (optionally) encryption of secret data. We discuss the impact of these requirements in the case studies later in this chapter.

Efficiency • A distributed file service should offer facilities that are of at least the same power and generality as those found in conventional file systems and should achieve a comparable level of performance. Birrell and Needham [1980] expressed their design aims for the Cambridge File Server (CFS) in these terms:

> We would wish to have a simple, low-level file server in order to share an expensive resource, namely a disk, whilst leaving us free to design the filing system most appropriate to a particular client, but we would wish also to have available a high-level system shared between clients.

The changed economics of disk storage have reduced the significance of their first goal, but their perception of the need for a range of services addressing the requirements of clients with different goals remains and can best be addressed by a modular architecture of the type outlined above.

The techniques used for the implementation of file services are an important part of the design of distributed systems. A distributed file system should provide a service that is comparable with, or better than, local file systems in performance and reliability. It must be convenient to administer, providing operations and tools that enable system administrators to install and operate the system conveniently.

12.1.3 Case studies

We have constructed an abstract model for a file service to act as an introductory example, separating implementation concerns and providing a simplified model. We describe the Sun Network File System in some detail, drawing on our simpler abstract model to clarify its architecture. The Andrew File System is then described, providing a view of a distributed file system that takes a different approach to scalability and consistency maintenance.

File service architecture • This is an abstract architectural model that underpins both NFS and AFS. It is based upon a division of responsibilities between three modules – a client module that emulates a conventional file system interface for application programs, and server modules, that perform operations for clients on directories and on files. The architecture is designed to enable a *stateless* implementation of the server module.

SUN NFS • Sun Microsystems's *Network File System* (NFS) has been widely adopted in industry and in academic environments since its introduction in 1985. The design and development of NFS were undertaken by staff at Sun Microsystems in 1984 [Sandberg *et al.* 1985, Sandberg 1987, Callaghan 1999]. Although several distributed file services had already been developed and used in universities and research laboratories, NFS was the first file service that was designed as a product. The design and implementation of NFS have achieved success both technically and commercially.

To encourage its adoption as a standard, the definitions of the key interfaces were placed in the public domain [Sun 1989], enabling other vendors to produce implementations, and the source code for a reference implementation was made available to other computer vendors under licence. It is now supported by many vendors, and the NFS protocol (version 3) is an Internet standard, defined in RFC 1813 [Callaghan *et al.* 1995]. Callaghan's book on NFS [Callaghan 1999] is an excellent source on the design and development of NFS and related topics.

NFS provides transparent access to remote files for client programs running on UNIX and other systems. The client-server relationship is symmetrical: each computer in an NFS network can act as both a client and a server, and the files at every machine can be made available for remote access by other machines. Any computer can be a server, exporting some of its files, and a client, accessing files on other machines. But it is common practice to configure larger installations with some machines as dedicated servers and others as workstations.

An important goal of NFS is to achieve a high level of support for hardware and operating system heterogeneity. The design is operating system–independent: client and server implementations exist for almost all operating systems and platforms, including all versions of Windows, Mac OS, Linux and every other version of UNIX. Implementations of NFS on high-performance multiprocessor hosts have been developed by several vendors, and these are widely used to meet storage requirements in intranets with many concurrent users.

Andrew File System • Andrew is a distributed computing environment developed at Carnegie Mellon University (CMU) for use as a campus computing and information system [Morris *et al.* 1986]. The design of the Andrew File System (hereafter abbreviated AFS) reflects an intention to support information sharing on a large scale by minimizing client-server communication. This is achieved by transferring whole files between server and client computers and caching them at clients until the server receives a more up-to-date version. We shall describe AFS-2, the first 'production' implementation, following the descriptions by Satyanarayanan [1989a, 1989b]. More recent descriptions can be found in Campbell [1997] and [Linux AFS].

AFS was initially implemented on a network of workstations and servers running BSD UNIX and the Mach operating system at CMU and was subsequently made available in commercial and public-domain versions. A public-domain implementation of AFS is available in the Linux operating system [Linux AFS]. AFS was adopted as the basis for the DCE/DFS file system in the Open Software Foundation's Distributed Computing Environment (DCE) [www.opengroup.org]. The design of DCE/DFS went beyond AFS in several important respects, which we outline in Section 12.5.

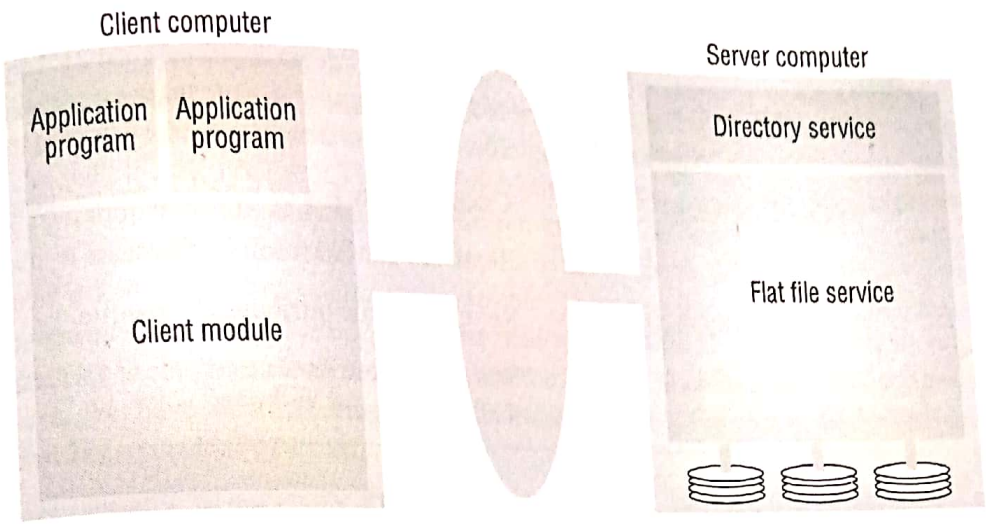
12.2 File service architecture

An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components – a *flat file service*, a *directory service* and a *client module*. The relevant modules and their relationships are shown in Figure 12.5. The flat file service and the directory service each export an interface for use by client programs, and their RPC interfaces, taken together, provide a comprehensive set of operations for access to files. The client module provides a single programming interface with operations on files similar to those found in conventional file systems. The design is *open* in the sense that different client modules can be used to implement different programming interfaces, simulating the file operations of a variety of different operating systems and optimizing the performance for different client and server hardware configurations.

The division of responsibilities between the modules can be defined as follows:

Flat file service • The flat file service is concerned with implementing operations on the contents of files. *Unique file identifiers* (UFIDs) are used to refer to files in all requests for flat file service operations. The division of responsibilities between the file service and the directory service is based upon the use of UFIDs. UFIDs are long sequences of bits chosen so that each file has a UFID that is unique among all of the files in a

Figure 12.5 File service architecture



distributed system. When the flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.

Directory service • The directory service provides a mapping between *text names* for files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to the directory service. The directory service provides the functions needed to generate directories, to add new file names to directories and to obtain UFIDs from directories. It is a client of the flat file service; its directory files are stored in files of the flat file service. When a hierarchic file-naming scheme is adopted, as in UNIX, directories hold references to other directories.

Client module • A client module runs in each client computer, integrating and extending the operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers. For example, in UNIX hosts, a client module would be provided that emulates the full set of UNIX file operations, interpreting UNIX multi-part file names by iterative requests to the directory service. The client module also holds information about the network locations of the flat file server and directory server processes. Finally, the client module can play an important role in achieving satisfactory performance through the implementation of a cache of recently used file blocks at the client.

Flat file service interface • Figure 12.6 contains a definition of the interface to a flat file service. This is the RPC interface used by client modules. It is not normally used directly by user-level programs. A *FileId* is invalid if the file that it refers to is not present in the server processing the request or if its access permissions are inappropriate for the operation requested. All of the procedures in the interface except *Create* throw exceptions if the *FileId* argument contains an invalid UFID or the user doesn't have sufficient access rights. These exceptions are omitted from the definition for clarity.

The most important operations are those for reading and writing. Both the *Read* and the *Write* operation require a parameter *i* specifying a position in the file. The *Read* operation copies the sequence of *n* data items beginning at item *i* from the specified file

Figure 12.6 Flat file service operations

Read(FileId, i, n) → *Data* — throws *BadPosition*	If $1 \leq i \leq Length(File)$: Reads a sequence of up to n items from a file starting at item i and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \leq i \leq Length(File)+1$: Writes a sequence of *Data* to a file, starting at item i, extending the file if necessary.
Create() → *FileId*	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) → *Attr*	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 12.3).

into *Data*, which is then returned to the client. The *Write* operation copies the sequence of data items in *Data* into the specified file beginning at item i, replacing the previous contents of the file at the corresponding position and extending the file if necessary.

Create creates a new, empty file and returns the UFID that is generated. *Delete* removes the specified file.

GetAttributes and *SetAttributes* enable clients to access the attribute record. *GetAttributes* is normally available to any client that is allowed to read the file. Access to the *SetAttributes* operation would normally be restricted to the directory service that provides access to the file. The values of the length and timestamp portions of the attribute record are not affected by *SetAttributes*; they are maintained separately by the flat file service itself.

Comparison with UNIX: Our interface and the UNIX file system primitives are functionally equivalent. It is a simple matter to construct a client module that emulates the UNIX system calls in terms of our flat file service and the directory service operations described in the next section.

In comparison with the UNIX interface, our flat file service has no *open* and *close* operations – files can be accessed immediately by quoting the appropriate UFID. The *Read* and *Write* requests in our interface include a parameter specifying a starting point within the file for each transfer, whereas the equivalent UNIX operations do not. In UNIX, each *read* or *write* operation starts at the current position of the read-write pointer, and the read-write pointer is advanced by the number of bytes transferred after each *read* or *write*. A *seek* operation is provided to enable the read-write pointer to be explicitly repositioned.

The interface to our flat file service differs from the UNIX file system interface mainly for reasons of fault tolerance:

Repeatable operations: With the exception of *Create*, the operations are *idempotent*, allowing the use of *at-least-once* RPC semantics – clients may repeat calls to which they receive no reply. Repeated execution of *Create* produces a different new file for each call.

Stateless servers: The interface is suitable for implementation by *stateless* servers. Stateless servers can be restarted after a failure and resume operation without any need for clients or the server to restore any state.

The UNIX file operations are neither idempotent nor consistent with the requirement for a stateless implementation. A read-write pointer is generated by the UNIX file system whenever a file is opened, and it is retained, together with the results of access-control checks, until the file is closed. The UNIX *read* and *write* operations are not idempotent; if an operation is accidentally repeated, the automatic advance of the read-write pointer results in access to a different portion of the file in the repeated operation. The read-write pointer is a hidden, client-related state variable. To mimic it in a file server, *open* and *close* operations would be needed, and the read-write pointer's value would have to be retained by the server as long as the relevant file is open. By eliminating the read-write pointer, we have eliminated most of the need for the file server to retain state information on behalf of specific clients.

Access control • In the UNIX file system, the user's access rights are checked against the access *mode* (read or write) requested in the *open* call (Figure 12.4 shows the UNIX file system API) and the file is opened only if the user has the necessary rights. The user identity (UID) used in the access rights check is retrieved during the user's earlier authenticated login and cannot be tampered with in non-distributed implementations. The resulting access rights are retained until the file is closed, and no further checks are required when subsequent operations on the same file are requested.

In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files. A user identity has to be passed with requests, and the server is vulnerable to forged identities. Furthermore, if the results of an access rights check were retained at the server and used for future accesses, the server would no longer be stateless. Two alternative approaches to the latter problem can be adopted:

- An access check is made whenever a file name is converted to a UFID, and the results are encoded in the form of a capability (see Section 11.2.4), which is returned to the client for submission with subsequent requests.
- A user identity is submitted with every client request, and access checks are performed by the server for every file operation.

Both methods enable stateless server implementation, and both have been used in distributed file systems. The second is more common; it is used in both NFS and AFS. Neither of these approaches overcomes the security problem concerning forged user identities, but we saw in Chapter 11 that this can be addressed by the use of digital signatures. Kerberos is an effective authentication scheme that has been applied to both NFS and AFS.

In our abstract model, we make no assumption about the method by which access control is implemented. The user identity is passed as an implicit parameter and can be used whenever it is needed.

Directory service interface • Figure 12.7 contains a definition of the RPC interface to a directory service. The primary purpose of the directory service is to provide a service for translating text names to UFIDs. In order to do so, it maintains directory files containing

Figure 12.7 Directory service operations

Lookup(Dir, Name) → FileId — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds *(Name, File)* to the directory and updates the file's attribute record. If *Name* is already in the directory, throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory, removes the entry containing *Name* from the directory. If *Name* is not in the directory, throws an exception.
GetNames(Dir, Pattern) → NameSeq	Returns all the text names in the directory that match the regular expression *Pattern*.

the mappings between text names for files and UFIDs. Each directory is stored as a conventional file with a UFID, so the directory service is a client of the file service.

We define only operations on individual directories. For each operation, a UFID for the file containing the directory is required (in the *Dir* parameter). The *Lookup* operation in the basic directory service performs a single *Name* → *UFID* translation. It is a building block for use in other services or in the client module to perform more complex translations, such as the hierarchic name interpretation found in UNIX. As before, exceptions caused by inadequate access rights are omitted from the definitions.

There are two operations for altering directories: *AddName* and *UnName*. *AddName* adds an entry to a directory and increments the reference count field in the file's attribute record.

UnName removes an entry from a directory and decrements the reference count. If this causes the reference count to reach zero, the file is removed. *GetNames* is provided to enable clients to examine the contents of directories and to implement pattern-matching operations on file names such as those found in the UNIX shell. It returns all or a subset of the names stored in a given directory. The names are selected by pattern matching against a regular expression supplied by the client.

The provision of pattern matching in the *GetNames* operation enables users to determine the names of one or more files by giving an incomplete specification of the characters in the names. A regular expression is a specification for a class of strings in the form of an expression containing a combination of literal substrings and symbols denoting variable characters or repeated occurrences of characters or substrings.

Hierarchic file system • A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure. Each directory holds the names of the files and other directories that are accessible from it. Any file or directory can be referenced using a *pathname* – a multi-part name that represents a path through

the tree. The root has a distinguished name, and each file or directory has a name in a directory. The UNIX file-naming scheme is not a strict hierarchy – files can have several names, and they can be in the same or different directories. This is implemented by a *link* operation, which adds a new name for a file to a specified directory.

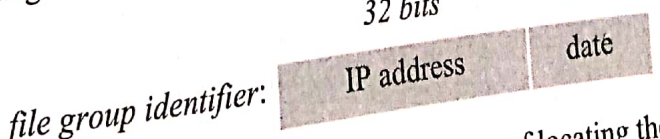
A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services that we have defined. A tree-structured network of directories is constructed with files at the leaves and directories at the other nodes of the tree. The root of the tree is a directory with a 'well-known' UFID. Multiple names for files can be supported using the *AddName* operation and the reference count field in the attribute record.

A function can be provided in the client module that gets the UFID of a file given its pathname. The function interprets the pathname starting from the root, using *Lookup* to obtain the UFID of each directory in the path.

In a hierarchic directory service, the file attributes associated with files should include a type field that distinguishes between ordinary files and directories. This is used when following a path to ensure that each part of the name, except the last, refers to a directory.

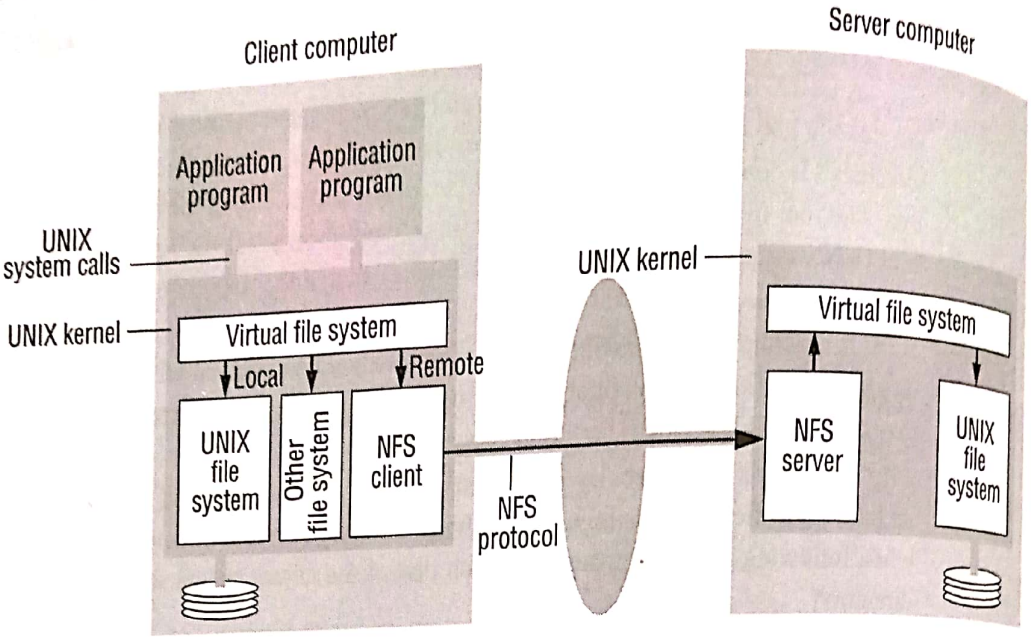
File groups • A *file group* is a collection of files located on a given server. A server may hold several file groups, and groups can be moved between servers, but a file cannot change the group to which it belongs. A similar construct called a *filesystem* is used in UNIX and in most other operating systems. (Terminology note: the single word *filesystem* refers to the set of files held in a storage device or partition, whereas the words *file system* refer to a software component that provides access to files.) File groups were originally introduced to support facilities for moving collections of files stored on removable media between computers. In a distributed file service, file groups support the allocation of files to file servers in larger logical units and enable the service to be implemented with files stored on several servers. In a distributed file system that supports file groups, the representation of UFIDs includes a file group identifier component, enabling the client module in each client computer to take responsibility for dispatching requests to the server that holds the relevant file group.

File group identifiers must be unique throughout a distributed system. Since file groups can be moved and distributed systems that are initially separate can be merged to form a single system, the only way to ensure that file group identifiers will always be distinct in a given system is to generate them with an algorithm that ensures global uniqueness. For example, whenever a new file group is created, a unique identifier can be generated by concatenating the 32-bit IP address of the host creating the new group with a 16-bit integer derived from the date, producing a unique 48-bit integer:



Note that the IP address *cannot* be used for the purpose of locating the file group, since it may be moved to another server. Instead, a mapping between group identifiers and servers should be maintained by the file service.

Figure 12.8 NFS architecture



12.3 Case study: Sun Network File System

Figure 12.8 shows the architecture of Sun NFS. It follows the abstract model defined in the preceding section. All implementations of NFS support the NFS protocol – a set of remote procedure calls that provide the means for clients to perform operations on a remote file store. The NFS protocol is operating system–independent but was originally developed for use in networks of UNIX systems, and we shall describe the UNIX implementation the NFS protocol (version 3).

The *NFS server* module resides in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module at the computer holding the relevant file system.

The NFS client and server modules communicate using remote procedure calls. Sun's RPC system, described in Section 5.3.3, was developed for use in NFS. It can be configured to use either UDP or TCP, and the NFS protocol is compatible with both. A port mapper service is included to enable clients to bind to services in a given host by name. The RPC interface to the NFS server is open: any process can send requests to an NFS server; if the requests are valid and they include valid user credentials, they will be acted upon. The submission of signed user credentials can be required as an optional security feature, as can the encryption of data for privacy and integrity.

Virtual file system • Figure 12.8 makes it clear that NFS provides access transparency: user programs can issue file operations for local or remote files without distinction. Other distributed file systems may be present that support UNIX system calls, and if so, they could be integrated in the same way.

The integration is achieved by a virtual file system (VFS) module, which has been added to the UNIX kernel to distinguish between local and remote files and to translate between the UNIX-independent file identifiers used by NFS and the internal file identifiers normally used in UNIX and other file systems. In addition, VFS keeps track of the filesystems that are currently available both locally and remotely, and it passes each request to the appropriate local system module (the UNIX file system, the NFS client module or the service module for another file system).

The file identifiers used in NFS are called *file handles*. A file handle is opaque to clients and contains whatever information the server needs to distinguish an individual file. In UNIX implementations of NFS, the file handle is derived from the file's *i-node number* by adding two extra fields as follows (the i-node number of a UNIX file is a number that serves to identify and locate the file within the file system in which the file is stored):

File handle:	Filesystem identifier	i-node number of file	i-node generation number

NFS adopts the UNIX mountable filesystem as the unit of file grouping defined in the preceding section. The *filesystem identifier* field is a unique number that is allocated to each filesystem when it is created (and in the UNIX implementation is stored in the superblock of the file system). The *i-node generation number* is needed because in the conventional UNIX file system i-node numbers are reused after a file is removed. In the VFS extensions to the UNIX file system, a generation number is stored with each file and is incremented each time the i-node number is reused (for example, in a UNIX *creat* system call). The client obtains the first file handle for a remote file system when it mounts it. File handles are passed from server to client in the results of *lookup*, *create* and *mkdir* operations (see Figure 12.9) and from client to server in the argument lists of all server operations.

The virtual file system layer has one VFS structure for each mounted file system and one *v-node* per open file. A VFS structure relates a remote file system to the local directory on which it is mounted. The v-node contains an indicator to show whether a file is local or remote. If the file is local, the v-node contains a reference to the index of the local file (an i-node in a UNIX implementation). If the file is remote, it contains the file handle of the remote file.

Client integration • The NFS client module plays the role described for the client module in our architectural model, supplying an interface suitable for use by conventional application programs. But unlike our model client module, it emulates the semantics of the standard UNIX file system primitives precisely and is integrated with the UNIX kernel. It is integrated with the kernel and not supplied as a library for loading into client processes so that:

- user programs can access files via UNIX system calls without recompilation or reloading;
- a single client module serves all of the user-level processes, with a shared cache of recently used blocks (described below);

Figure 12.9 NFS server operations (NFS version 3 protocol, simplified)

lookup(dirfh, name) → *fh, attr*	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) → *newfh, attr*	Creates a new file *name* in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) → *status*	Removes file *name* from directory *dirfh*.
getattr(fh) → *attr*	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
setattr(fh, attr) → *attr*	Sets the attributes (mode, user ID, group ID, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) → *attr, data*	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
write(fh, offset, count, data) → *attr*	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) → *status*	Changes the name of file *name* in directory *dirfh* to *toname* in directory *todirfh*.
link(newdirfh, newname, fh) → *status*	Creates an entry *newname* in the directory *newdirfh* that refers to the file *or* directory *fh*.
symlink(newdirfh, newname, string) → *status*	Creates an entry *newname* in the directory *newdirfh* of type *symbolic link* with the value *string*. The server does not interpret the *string* but makes a symbolic link file to hold it.
readlink(fh) → *string*	Returns the string that is associated with the symbolic link file identified by *fh*.
mkdir(dirfh, name, attr) → *newfh, attr*	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) → *status*	Removes the empty directory *name* from the parent directory *dirfh*. Fails if the directory is not empty.
readdir(dirfh, cookie, count) → *entries*	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle and an opaque pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the following entry. If the value of *cookie* is 0, reads from the first entry in the directory.
statfs(fh) → *fsstats*	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

- the encryption key used to authenticate user IDs passed to the server (see below) can be retained in the kernel, preventing impersonation by user-level clients.

The NFS client module cooperates with the virtual file system in each client machine. It operates in a similar manner to the conventional UNIX file system, transferring blocks of files to and from the server and caching the blocks in the local memory whenever possible. It shares the same buffer cache that is used by the local input-output system.

But since several clients in different host machines may simultaneously access the same remote file, a new and significant cache consistency problem arises.

Access control and authentication • Unlike the conventional UNIX file system, the NFS server is stateless and does not keep files open on behalf of its clients. So the server must check the user's identity against the file's access permission attributes afresh on each request, to see whether the user is permitted to access the file in the manner requested. The Sun RPC protocol requires clients to send user authentication information (for example, the conventional UNIX 16-bit user ID and group ID) with each request and this is checked against the access permission in the file attributes. These additional parameters are not shown in our overview of the NFS protocol in Figure 12.9; they are supplied automatically by the RPC system.

In its simplest form, there is a security loophole in this access-control mechanism. An NFS server provides a conventional RPC interface at a well-known port on each host and any process can behave as a client, sending requests to the server to access or update a file. The client can modify the RPC calls to include the user ID of any user, impersonating the user without their knowledge or permission. This security loophole has been closed by the use of an option in the RPC protocol for the DES encryption of the user's authentication information. More recently, Kerberos has been integrated with Sun NFS to provide a stronger and more comprehensive solution to the problems of user authentication and security; we describe this below.

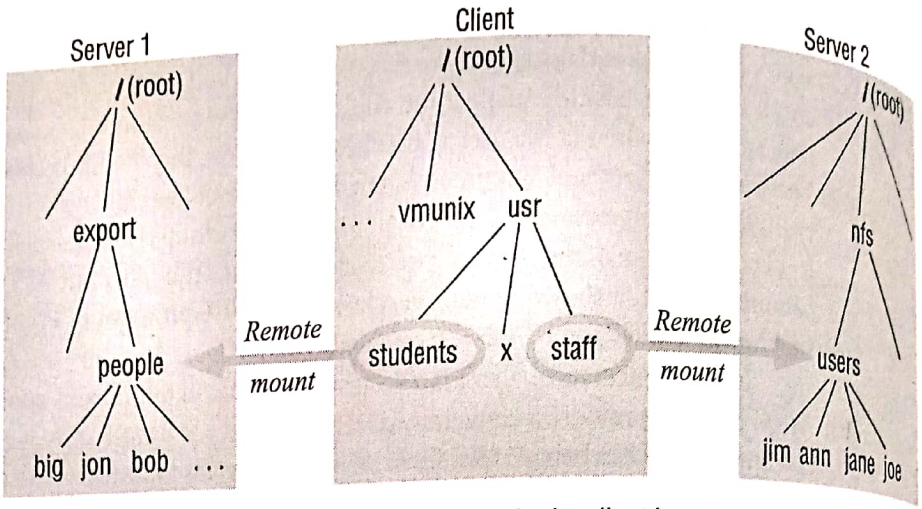
NFS server interface • A simplified representation of the RPC interface provided by NFS version 3 servers (defined in RFC 1813 [Callaghan *et al.* 1995]) is shown in Figure 12.9. The NFS file access operations *read*, *write*, *getattr* and *setattr* are almost identical to the *Read*, *Write*, *GetAttributes* and *SetAttributes* operations defined for our flat file service model (Figure 12.6). The *lookup* operation and most of the other directory operations defined in Figure 12.9 are similar to those in our directory service model (Figure 12.7).

The file and directory operations are integrated in a single service; the creation and insertion of file names in directories is performed by a single *create* operation, which takes the text name of the new file and the file handle for the target directory as arguments. The other NFS operations on directories are *create*, *remove*, *rename*, *link*, *symlink*, *readlink*, *mkdir*, *rmdir*, *readdir* and *statfs*. They resemble their UNIX counterparts with the exception of *readdir*, which provides a representation-independent method for reading the contents of directories, and *statfs*, which gives the status information on remote file systems.

Mount service • The mounting of subtrees of remote filesystems by clients is supported by a separate *mount service* process that runs at user level on each NFS server computer. On each server, there is a file with a well-known name (*/etc/exports*) containing the names of local filesystems that are available for remote mounting. An access list is associated with each filesystem name indicating which hosts are permitted to mount the filesystem.

Clients use a modified version of the UNIX *mount* command to request mounting of a remote filesystem, specifying the remote host's name, the pathname of a directory in the remote filesystem and the local name with which it is to be mounted. The remote directory may be any subtree of the required remote filesystem, enabling clients to mount any part of the remote filesystem. The modified *mount* command communicates

Figure 12.10 Local and remote filesystems accessible on an NFS client



Note: The file system mounted at */usr/students* in the client is actually the subtree located at */export/people* in Server 1; the filesystem mounted at */usr/staff* in the client is actually the subtree located at /nfs/users in Server 2.

with the mount service process on the remote host using a *mount protocol*. This is an RPC protocol and includes an operation that takes a directory pathname and returns the file handle of the specified directory if the client has access permission for the relevant filesystem. The location (IP address and port number) of the server and the file handle for the remote directory are passed on to the VFS layer and the NFS client.

Figure 12.10 illustrates a *Client* with two remotely mounted file stores. The nodes *people* and *users* in filesystems at *Server 1* and *Server 2* are mounted over nodes *students* and *staff* in *Client*'s local file store. The meaning of this is that programs running at *Client* can access files at *Server 1* and *Server 2* by using pathnames such as */usr/students/jon* and */usr/staff/ann*.

Remote filesystems may be *hard-mounted* or *soft-mounted* in a client computer. When a user-level process accesses a file in a filesystem that is hard-mounted, the process is suspended until the request can be completed, and if the remote host is unavailable for any reason the NFS client module continues to retry the request until it is satisfied. Thus in the case of a server failure, user-level processes are suspended until the server restarts and then they continue just as though there had been no failure. But if the relevant filesystem is soft-mounted, the NFS client module returns a failure indication to user-level processes after a small number of retries. Properly constructed programs will then detect the failure and take appropriate recovery or reporting actions. But many UNIX utilities and applications do not test for the failure of file access operations, and these behave in unpredictable ways in the case of failure of a soft-mounted filesystem. For this reason, many installations use hard mounting exclusively, with the consequence that programs are unable to recover gracefully when an NFS server is unavailable for a significant period.

Pathname translation • UNIX file systems translate multi-part file pathnames to i-node references in a step-by-step process whenever the *open*, *creat* or *stat* system calls are used. In NFS, pathnames cannot be translated at a server, because the name may cross a

'mount point' at the client – directories holding different parts of a multi-part name may reside in filesystems at different servers. So pathnames are parsed, and their translation is performed in an iterative manner by the client. Each part of a name that refers to a remote-mounted directory is translated to a file handle using a separate *lookup* request to the remote server.

The *lookup* operation looks for a single part of a pathname in a given directory and returns the corresponding file handle and file attributes. The file handle returned in the previous step is used as a parameter in the next *lookup* step. Since file handles are opaque to NFS client code, the virtual file system is responsible for resolving file handles to a local or a remote directory and performing the necessary indirection when it references a local mount point. Caching of the results of each step in pathname translations alleviates the apparent inefficiency of this process, taking advantage of locality of reference to files and directories; users and programs typically access files in only one or a small number of directories.

Automounter • The automounter was added to the UNIX implementation of NFS in order to mount a remote directory dynamically whenever an 'empty' mount point is referenced by a client. The original implementation of the automounter ran as a user-level UNIX process in each client computer. Later versions (called *autofs*) were implemented in the kernel for Solaris and Linux. We describe the original version here.

The automounter maintains a table of mount points (pathnames) with a reference to one or more NFS servers listed against each. It behaves like a local NFS server at the client machine. When the NFS client module attempts to resolve a pathname that includes one of these mount points, it passes to the local automounter a *lookup*() request that locates the required filesystem in its table and sends a 'probe' request to each server listed. The filesystem on the first server to respond is then mounted at the client using the normal mount service. The mounted filesystem is linked to the mount point using a symbolic link, so that accesses to it will not result in further requests to the automounter. File access then proceeds in the normal way without further reference to the automounter unless there are no references to the symbolic link for several minutes. In the latter case, the automounter unmounts the remote filesystem.

The later kernel implementations replaced the symbolic links with real mounts, avoiding some problems that arose with applications that cached the temporary pathnames used in user-level automounters [Callaghan 1999].

A simple form of read-only replication can be achieved by listing several servers containing identical copies of a filesystem or file subtree against a name in the automounter table. This is useful for heavily used file systems that change infrequently, such as UNIX system binaries. For example, copies of the */usr/lib* directory and its subtree might be held on more than one server. On the first occasion that a file in */usr/lib* is opened at a client, all of the servers will be sent probe messages, and the first to respond will be mounted at the client. This provides a limited degree of fault tolerance and load balancing, since the first server to respond will be one that has not failed and is likely to be one that is not heavily occupied with servicing other requests.

Server caching • Caching in both the client and the server computer are indispensable features of NFS implementations in order to achieve adequate performance.

In conventional UNIX systems, file pages, directories and file attributes that have been read from disk are retained in a main memory *buffer cache* until the buffer space

is required for other pages. If a process then issues a read or a write request for a page that is already in the cache, it can be satisfied without another disk access. *Read-ahead* anticipates read accesses and fetches the pages following those that have most recently been read, and *delayed-write* optimizes writes: when a page has been altered (by a write request), its new contents are written to disk only when the buffer page is required for another page. To guard against loss of data in a system crash, the UNIX *sync* operation flushes altered pages to disk every 30 seconds. These caching techniques work in a conventional UNIX environment because all read and write requests issued by user-level processes pass through a single cache that is implemented in the UNIX kernel space. The cache is always kept up-to-date, and file accesses cannot bypass the cache.

NFS servers use the cache at the server machine just as it is used for other file accesses. The use of the server's cache to hold recently read disk blocks does not raise any consistency problems; but when a server performs write operations, extra measures are needed to ensure that clients can be confident that the results of the write operations are persistent, even when server crashes occur. In version 3 of the NFS protocol, the *write* operation offers two options for this (not shown in Figure 12.9):

1. Data in *write* operations received from clients is stored in the memory cache at the server *and* written to disk before a reply is sent to the client. This is called *write-through* caching. The client can be sure that the data is stored persistently as soon as the reply has been received.

2. Data in *write* operations is stored only in the memory cache. It will be written to disk when a *commit* operation is received for the relevant file. The client can be sure that the data is persistently stored only when a reply to a *commit* operation for the relevant file has been received. Standard NFS clients use this mode of operation, issuing a *commit* whenever a file that was open for writing is closed.

Commit is an additional operation provided in version 3 of the NFS protocol; it was added to overcome a performance bottleneck caused by the write-through mode of operation in servers that receive large numbers of *write* operations.

The requirement for write-through in distributed file systems is an instance of the independent failure modes discussed in Chapter 1 – clients continue to operate when a server fails, and application programs may take actions on the assumption that the results of previous writes are committed to disk storage. This is unlikely to occur in the case of local file updates, because the failure of a local file system is almost certain to result in the failure of all the application processes running on the same computer.

Client caching • The NFS client module caches the results of *read*, *write*, *getattr*, *lookup* and *readdir* operations in order to reduce the number of requests transmitted to servers. Client caching introduces the potential for different versions of files or portions of files to exist in different client nodes, because writes by a client do not result in the immediate updating of cached copies of the same file in other clients. Instead, clients are responsible for polling the server to check the currency of the cached data that they hold.

A timestamp-based method is used to validate cached blocks before they are used. Each data or metadata item in the cache is tagged with two timestamps:

Tc is the time when the cache entry was last validated.

Tm is the time when the block was last modified at the server.

A cache entry is valid at time T if $T - Tc$ is less than a freshness interval t, or if the value for Tm recorded at the client matches the value of Tm at the server (that is, the data has not been modified at the server since the cache entry was made). Formally, the validity condition is:

$$(T - Tc < t) \vee (Tm_{client} = Tm_{server})$$

The selection of a value for t involves a compromise between consistency and efficiency. A very short freshness interval will result in a close approximation to one-copy consistency, at the cost of a relatively heavy load of calls to the server to check the value of Tm_{server}. In Sun Solaris clients, t is set adaptively for individual files to a value in the range 3 to 30 seconds, depending on the frequency of updates to the file. For directories the range is 30 to 60 seconds, reflecting the lower risk of concurrent updates.

There is one value of Tm_{server} for all the data blocks in a file and another for the file attributes. Since NFS clients cannot determine whether a file is being shared or not, the validation procedure must be used for all file accesses. A validity check is performed whenever a cache entry is used. The first half of the validity condition can be evaluated without access to the server. If it is true, then the second half need not be evaluated; if it is false, the current value of Tm_{server} is obtained (by means of a *getattr* call to the server) and compared with the local value Tm_{client}. If they are the same, then the cache entry is taken to be valid and the value of T_c for that cache entry is updated to the current time. If they differ, then the cached data has been updated at the server and the cache entry is invalidated, resulting in a request to the server for the relevant data.

Several measures are used to reduce the traffic of *getattr* calls to the server:

- Whenever a new value of Tm_{server} is received at a client, it is applied to all cache entries derived from the relevant file.
- The current attribute values are sent 'piggybacked' with the results of every operation on a file, and if the value of Tm_{server} has changed the client uses it to update the cache entries relating to the file.
- The adaptive algorithm for setting freshness interval t outlined above reduces the traffic considerably for most files.

The validation procedure does not guarantee the same level of consistency of files that is provided in conventional UNIX systems, since recent updates are not always visible to clients sharing a file. There are two sources of time lag; the delay after a write before the updated data leaves the cache in the updating client's kernel and the three-second 'window' for cache validation. Fortunately, most UNIX applications do not depend critically upon the synchronization of file updates, and few difficulties have been reported from this source.

Writes are handled differently. When a cached page is modified it is marked as 'dirty' and is scheduled to be flushed to the server asynchronously. Modified pages are flushed when the file is closed or a *sync* occurs at the client, and they are flushed more frequently if bio-daemons are in use (see below). This does not provide the same persistence guarantee as the server cache, but it emulates the behaviour for local writes.

To implement read-ahead and delayed-write, the NFS client needs to perform some reads and writes asynchronously. This is achieved in UNIX implementations of

NFS by the inclusion of one or more *bio–daemon* processes at each client. (*Bio* stands for block input-output; the term *daemon* is often used to refer to user-level processes that perform system tasks.) The role of the bio-daemons is to perform read-ahead and delayed-write operations. A bio-daemon is notified after each read request, and it requests the transfer of the following file block from the server to the client cache. In the case of writing, the bio-daemon will send a block to the server whenever a block has been filled by a client operation. Directory blocks are sent whenever a modification has occurred.

Bio-daemon processes improve performance, ensuring that the client module does not block waiting for *read*s to return or *write*s to commit at the server. They are not a logical requirement, since in the absence of read-ahead, a *read* operation in a user process will trigger a synchronous request to the relevant server, and the results of *write*s in user processes will be transferred to the server when the relevant file is closed or when the virtual file system at the client performs a *sync* operation.

Other optimizations • The Sun file system is based on the UNIX BSD Fast File System which uses 8-kbyte disk blocks, resulting in fewer file system calls for sequential file access than previous UNIX systems. The UDP packets used for the implementation of Sun RPC are extended to 9 kilobytes, enabling an RPC call containing an entire block as an argument to be transferred in a single packet and minimizing the effect of network latency when reading files sequentially. In NFS version 3, there is no limit on the maximum size of file blocks that can be handled in *read* and *write* operations; clients and servers can negotiate sizes larger than 8 kbytes if both are able to handle them.

As mentioned above, the file status information cached at clients must be updated at least every three seconds for active files. To reduce the consequential server load resulting from *getattr* requests, all operations that refer to files or directories are taken as implicit *getattr* requests, and the current attribute values are 'piggybacked' along with the other results of the operation.

Securing NFS with Kerberos • In Section 11.6.2 we described the Kerberos authentication system developed at MIT, which has become an industry standard for securing intranet servers against unauthorized access and imposter attacks. The security of NFS implementations has been strengthened by the use of the Kerberos scheme to authenticate clients. In this subsection, we describe the 'Kerberization' of NFS as carried out by the designers of Kerberos.

In the original standard implementation of NFS, the user's identity is included in each request in the form of an unencrypted numeric identifier. (The identifier can be encrypted in later versions of NFS.) NFS does not take any further steps to check the authenticity of the identifier supplied. This implies a high degree of trust in the integrity of the client computer and its software by NFS, whereas the aim of Kerberos and other authentication-based security systems is to reduce to a minimum the range of components in which trust is assumed. Essentially, when NFS is used in a 'Kerberized' environment it should accept requests only from clients whose identity can be shown to have been authenticated by Kerberos.

One obvious solution considered by the Kerberos developers was to change the nature of the credentials required by NFS to be a full-blown Kerberos ticket and authenticator. But because NFS is implemented as a stateless server, each individual file access request is handled on its face value and the authentication data would have to be

included in each request. This was considered unacceptably expensive in terms of the time required to perform the necessary encryptions and because it would have entailed adding the Kerberos client library to the kernel of all workstations.

Instead, a hybrid approach was adopted in which the NFS mount server is supplied with full Kerberos authentication data for the users when their home and root filesystems are mounted. The results of this authentication, including the user's conventional numerical identifier and the address of the client computer, are retained by the server with the mount information for each filesystem. (Although the NFS server does not retain state relating to individual client processes, it does retain the current mounts at each client computer.)

On each file access request, the NFS server checks the user identifier and the sender's address and grants access only if they match those stored at the server for the relevant client at mount time. This hybrid approach involves only minimal additional cost and is safe against most forms of attack, provided that only one user at a time can log in to each client computer. At MIT, the system is configured so that this is the case. Recent NFS implementations include Kerberos authentication as one of several options for authentication, and sites that also run Kerberos servers are advised to use this option.

Performance • Early performance figures reported by Sandberg [1987] showed that the use of NFS did not normally impose a performance penalty in comparison with access to files stored on local disks. He identified two remaining problem areas:

- frequent use of the *getattr* call in order to fetch timestamps from servers for cache validation;
- relatively poor performance of the *write* operation because write-through was used at the server.

He noted that writes are relatively infrequent in typical UNIX workloads (about 5% of all calls to the server), and the cost of write-through is therefore tolerable except when large files are written to the server. Further, the version of NFS that he tested did not include the *commit* mechanism outlined above, which has resulted in a substantial improvement in write performance in current versions. His results also show that the *lookup* operation accounts for almost 50% of server calls. This is a consequence of the step-by-step pathname translation method necessitated by UNIX's file-naming semantics.

Measurements are taken regularly by Sun and other NFS implementors using an updated version of an exhaustive set of benchmark programs known as LADDIS [Keith and Wittle 1993]. Current and past results are available at the SPEC web site [www.spec.org]. Performance is summarized there for NFS server implementations from many vendors and different hardware configurations. Single-CPU implementations based on PC hardware but with dedicated operating systems achieve throughputs in excess of 12,000 server operations per second and large multi-processor configurations with many disks and controllers have achieved throughputs of up to 300,000 server operations per second. These figures indicate that NFS offers a very effective solution to distributed storage needs in intranets of most sizes and types of use, ranging for example from a traditional UNIX load of development by several hundred software engineers to a battery of web servers serving material from an NFS server.

NFS summary • Sun NFS closely follows our abstract model. The resulting design provides good location and access transparency if the NFS mount service is used properly to produce similar name spaces at all clients. NFS supports heterogeneous hardware and operating systems. The NFS server implementation is stateless, enabling clients and servers to resume execution after a failure without the need for any recovery procedures. Migration of files or filesystems is not supported, except at the level of manual intervention to reconfigure mount directives after the movement of a filesystem to a new location.

The performance of NFS is much enhanced by the caching of file blocks at each client computer. This is important for the achievement of satisfactory performance but results in some deviation from strict UNIX one-copy file update semantics.

The other design goals of NFS and the extent to which they have been achieved are discussed below.

Access transparency: The NFS client module provides an application programming interface to local processes that is identical to the local operating system's interface. Thus in a UNIX client, accesses to remote files are performed using the normal UNIX system calls. No modifications to existing programs are required to enable them to operate correctly with remote files.

Location transparency: Each client establishes a file name space by adding mounted directories in remote filesystems to its local name space. File systems have to be *exported* by the node that holds them and *remote-mounted* by a client before they can be accessed by processes running in the client (see Figure 12.10). The point in a client's name hierarchy at which a remote-mounted file system appears is determined by the client; NFS does not enforce a single network-wide file name space – each client sees a set of remote filesystems that is determined locally, and remote files may have different pathnames on different clients, but a uniform name space can be established with appropriate configuration tables in each client, achieving the goal of location transparency.

Mobility transparency: Filesystems (in the UNIX sense, that is, subtrees of files) may be moved between servers, but the remote mount tables in each client must then be updated separately to enable the clients to access the filesystems in their new locations, thus migration transparency is not fully achieved by NFS.

Scalability: The published performance figures show that NFS servers can be built to handle very large real-world loads in an efficient and cost-effective manner. The performance of a single server can be increased by the addition of processors, disks and controllers. When the limits of that process are reached, additional servers must be installed and the filesystems must be reallocated between them. The effectiveness of that strategy is limited by the existence of 'hot spot' files – single files that are accessed so frequently that the server reaches a performance limit. When loads exceed the maximum performance available with that strategy, a distributed file system that supports replication of updatable files (such as Coda, described in Chapter 18), or one such as AFS that reduces the protocol traffic by the caching of whole files, may offer a better solution. We discuss other approaches to scalability in Section 12.5.

File replication: Read-only file stores can be replicated on several NFS servers, but NFS does not support file replication with updates. The Sun Network Information Service (NIS) is a separate service available for use with NFS that supports the replication of simple databases organized as key-value pairs (for example, the UNIX system files */etc/passwd* and */etc/hosts*). It manages the distribution of updates and accesses to the replicated files based on a simple master–slave replication model (also known as the *primary copy* model, discussed further in Chapter 18) with provision for the replication of part or all of the database at each site. NIS provides a shared repository for system information that changes infrequently and does not require updates to occur simultaneously at all sites.

Hardware and operating system heterogeneity: NFS has been implemented for almost every known operating system and hardware platform and is supported by a variety of filing systems.

Fault tolerance: The stateless and idempotent nature of the NFS file access protocol ensures that the failure modes observed by clients when accessing remote files are similar to those for local file access. When a server fails, the service that it provides is suspended until the server is restarted, but once it has been restarted user-level client processes proceed from the point at which the service was interrupted, unaware of the failure (except in the case of access to *soft-mounted* remote file systems). In practice, hard mounting is used in most instances, and this tends to impede application programs handling server failures gracefully.

The failure of a client computer or a user-level process in a client has no effect on any server that it may be using, since servers hold no state on behalf of their clients.

Consistency: We have described the update behaviour in some detail. It provides a close approximation to one-copy semantics and meets the needs of the vast majority of applications, but the use of file sharing via NFS for communication or close coordination between processes on different computers cannot be recommended.

Security: The need for security in NFS emerged with the connection of most intranets to the Internet. The integration of Kerberos with NFS was a major step forward. Other recent developments include the option to use a secure RPC implementation (RPCSEC_GSS, documented in RFC 2203 [Eisler *et al.* 1997]) for authentication and to ensure the privacy and security of the data transmitted with read and write operations. Installations that have not deployed these mechanisms abound, though, and they are insecure.

Efficiency: The measured performance of several implementations of NFS and its widespread adoption for use in situations that generate very heavy loads are clear indications of the efficiency with which the NFS protocol can be implemented.

12.4 Case study: The Andrew File System

Like NFS, AFS provides transparent access to remote shared files for UNIX programs running on workstations. Access to AFS files is via the normal UNIX file primitives, enabling existing UNIX programs to access AFS files without modification or recompilation. AFS is compatible with NFS. AFS servers hold 'local' UNIX files, but the filing system in the servers is NFS-based, so files are referenced by NFS-style file handles rather than i-node numbers, and the files may be remotely accessed via NFS.

AFS differs markedly from NFS in its design and implementation. The differences are primarily attributable to the identification of scalability as the most important design goal. AFS is designed to perform well with larger numbers of active users than other distributed file systems. The key strategy for achieving scalability is the caching of whole files in client nodes. AFS has two unusual design characteristics:

Whole-file serving: The entire contents of directories and files are transmitted to client computers by AFS servers (in AFS-3, files larger than 64 kbytes are transferred in 64-kbyte chunks).

Whole-file caching: Once a copy of a file or a chunk has been transferred to a client computer it is stored in a cache on the local disk. The cache contains several hundred of the files most recently used on that computer. The cache is permanent, surviving reboots of the client computer. Local copies of files are used to satisfy clients' *open* requests in preference to remote copies whenever possible.

Scenario • Here is a simple scenario illustrating the operation of AFS:

1. When a user process in a client computer issues an *open* system call for a file in the shared file space and there is not a current copy of the file in the local cache, the server holding the file is located and is sent a request for a copy of the file.
2. The copy is stored in the local UNIX file system in the client computer. The copy is then *open*ed and the resulting UNIX file descriptor is returned to the client.
3. Subsequent *read*, *write* and other operations on the file by processes in the client computer are applied to the local copy.
4. When the process in the client issues a *close* system call, if the local copy has been updated its contents are sent back to the server. The server updates the file contents and the timestamps on the file. The copy on the client's local disk is retained in case it is needed again by a user-level process on the same workstation.

We discuss the observed performance of AFS below, but we can make some general observations and predictions here based on the design characteristics described above:

- For shared files that are infrequently updated (such as those containing the code of UNIX commands and libraries) and for files that are normally accessed by only a single user (such as most of the files in a user's home directory and its subtree), locally cached copies are likely to remain valid for long periods – in the first case because they are not updated and in the second because if they are updated, the updated copy will be in the cache on the owner's workstation. These classes of file account for the overwhelming majority of file accesses.

- The local cache can be allocated a substantial proportion of the disk space on each workstation – say, 100 megabytes. This is normally sufficient for the establishment of a working set of the files used by one user. The provision of sufficient cache storage for the establishment of a working set ensures that files in regular use on a given workstation are normally retained in the cache until they are needed again.
- The design strategy is based on some assumptions about average and maximum file size and locality of reference to files in UNIX systems. These assumptions are derived from observations of typical UNIX workloads in academic and other environments [Satyanarayanan 1981, Ousterhout *et al.* 1985, Floyd 1986]. The most important observations are:
  - Files are small; most are less than 10 kilobytes in size.
  - Read operations on files are much more common than writes (about six times more common).
  - Sequential access is common, and random access is rare.
  - Most files are read and written by only one user. When a file is shared, it is usually only one user who modifies it.
  - Files are referenced in bursts. If a file has been referenced recently, there is a high probability that it will be referenced again in the near future.

  These observations were used to guide the design and optimization of AFS, *not* to restrict the functionality seen by users.
- AFS works best with the classes of file identified in the first point above. There is one important type of file that does not fit into any of these classes – databases are typically shared by many users and are often updated quite frequently. The designers of AFS have explicitly excluded the provision of storage facilities for databases from their design goals, stating that the constraints imposed by different naming structures (that is, content-based access) and the need for fine-grained data access, concurrency control and atomicity of updates make it difficult to design a distributed database system that is also a distributed file system. They argue that the provision of facilities for distributed databases should be addressed separately [Satyanarayanan 1989a].

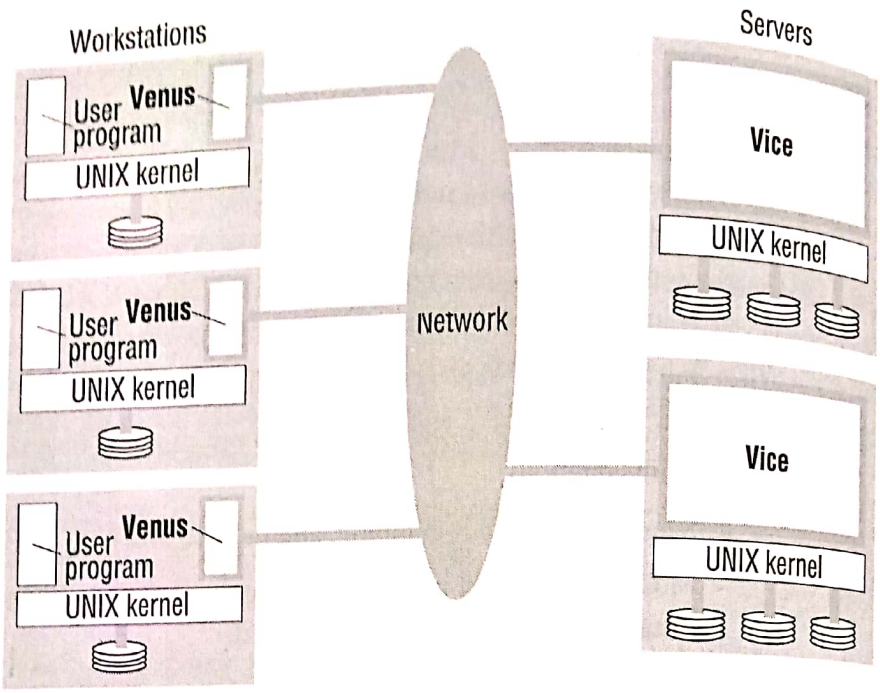
12.4.1 Implementation

The above scenario illustrates AFS's operation but leaves many questions about its implementation unanswered. Among the most important are:

- How does AFS gain control when an *open* or *close* system call referring to a file in the shared file space is issued by a client?
- How is the server holding the required file located?
- What space is allocated for cached files in workstations?
- How does AFS ensure that the cached copies of files are up-to-date when files may be updated by several clients?

We answer these questions below.

Figure 12.11 Distribution of processes in the Andrew File System



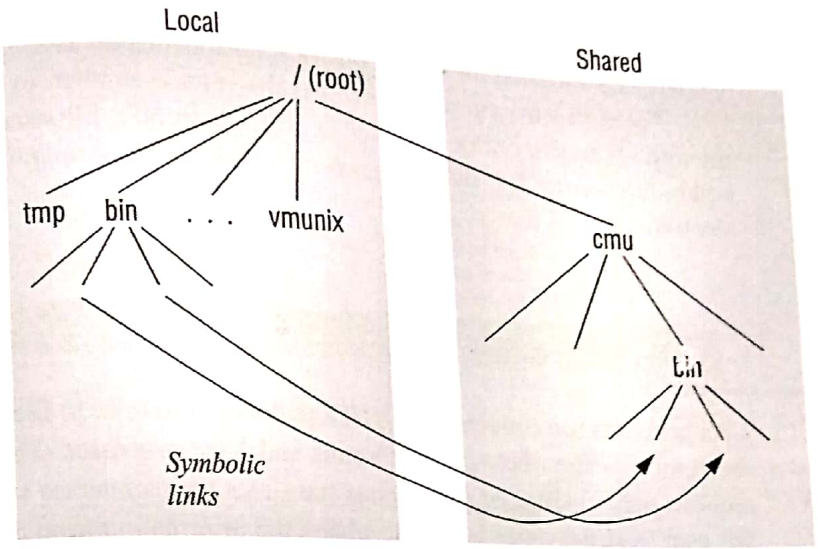
AFS is implemented as two software components that exist as UNIX processes called *Vice* and *Venus*. Figure 12.11 shows the distribution of Vice and Venus processes. Vice is the name given to the server software that runs as a user-level UNIX process in each server computer, and Venus is a user-level process that runs in each client computer and corresponds to the client module in our abstract model.

The files available to user processes running on workstations are either *local* or *shared*. Local files are handled as normal UNIX files. They are stored on a workstation's disk and are available only to local user processes. Shared files are stored on servers, and copies of them are cached on the local disks of workstations. The name space seen by user processes is illustrated in Figure 12.12. It is a conventional UNIX directory hierarchy, with a specific subtree (called *cmu*) containing all of the shared files. This splitting of the file name space into local and shared files leads to some loss of location transparency, but this is hardly noticeable to users other than system administrators. Local files are used only for temporary files (*/tmp*) and processes that are essential for workstation startup. Other standard UNIX files (such as those normally found in */bin*, */lib* and so on) are implemented as symbolic links from local directories to files held in the shared space. Users' directories are in the shared space, enabling users to access their files from any workstation.

The UNIX kernel in each workstation and server is a modified version of BSD UNIX. The modifications are designed to intercept *open*, *close* and some other file system calls when they refer to files in the shared name space and pass them to the Venus process in the client computer (illustrated in Figure 12.13). One other kernel modification is included for performance reasons, and this is described later.

One of the file partitions on the local disk of each workstation is used as a cache, holding the cached copies of files from the shared space. Venus manages the cache, removing the least recently used files when a new file is acquired from a server to make

Figure 12.12 File name space seen by clients of AFS

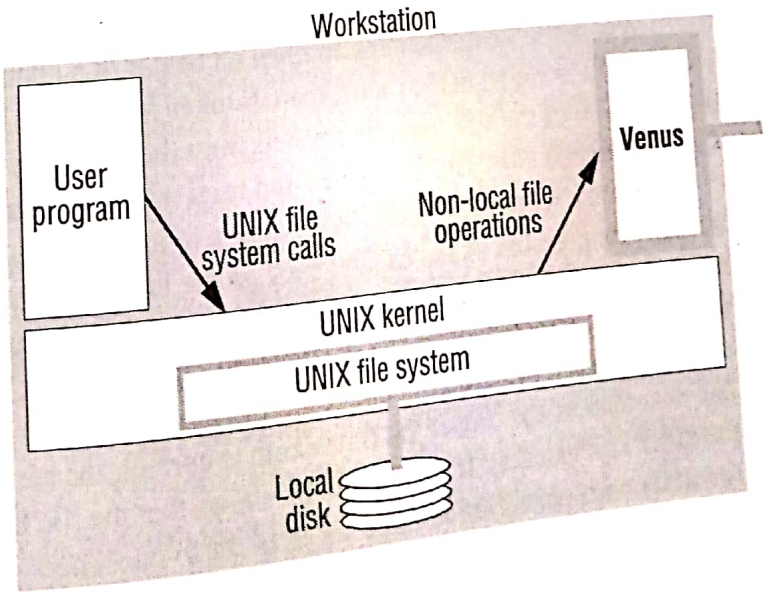


the required space if the partition is full. The workstation cache is usually large enough to accommodate several hundred average-sized files, rendering the workstation largely independent of the Vice servers once a working set of the current user's files and frequently used system files has been cached.

AFS resembles the abstract file service model described in Section 12.2 in these respects:

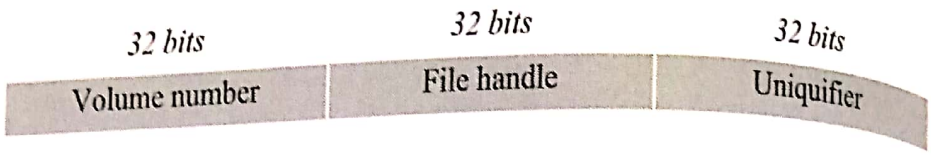
- A flat file service is implemented by the Vice servers, and the hierarchic directory structure required by UNIX user programs is implemented by the set of Venus processes in the workstations.
- Each file and directory in the shared file space is identified by a unique, 96-bit file identifier (*fid*) similar to a UFID. The Venus processes translate the pathnames issued by clients to *fids*.

Figure 12.13 System call interception in AFS



Files are grouped into *volumes* for ease of location and movement. Volumes are generally smaller than the UNIX filesystems, which are the unit of file grouping in NFS. For example, each user's personal files are generally located in a separate volume. Other volumes are allocated for system binaries, documentation and library code.

The representation of *fids* includes the volume number for the volume containing the file (*cf.* the *file group identifier* in UFIDs), an NFS file handle identifying the file within the volume (*cf.* the *file number* in UFIDs) and a *uniquifier* to ensure that file identifiers are not reused:



User programs use conventional UNIX pathnames to refer to files, but AFS uses *fids* in the communication between the Venus and Vice processes. The Vice servers accept requests only in terms of *fids*. Venus translates the pathnames supplied by clients into *fids* using a step-by-step lookup to obtain the information from the file directories held in the Vice servers.

Figure 12.14 describes the actions taken by Vice, Venus and the UNIX kernel when a user process issues each of the system calls mentioned in our outline scenario above. The *callback promise* mentioned here is a mechanism for ensuring that cached copies of files are updated when another client closes the same file after updating it. This mechanism is discussed in the next section.

12.4.2 Cache consistency

When Vice supplies a copy of a file to a Venus process it also provides a *callback promise* – a token issued by the Vice server that is the custodian of the file, guaranteeing that it will notify the Venus process when any other client modifies the file. Callback promises are stored with the cached files on the workstation disks and have two states: *valid* or *cancelled*. When a server performs a request to update a file it notifies all of the Venus processes to which it has issued callback promises by sending a *callback* to each – a callback is a remote procedure call from a server to a Venus process. When the Venus process receives a callback, it sets the *callback promise* token for the relevant file to *cancelled*.

Whenever Venus handles an *open* on behalf of a client, it checks the cache. If the required file is found in the cache, then its token is checked. If its value is *cancelled*, then a fresh copy of the file must be fetched from the Vice server, but if the token is *valid*, then the cached copy can be opened and used without reference to Vice.

When a workstation is restarted after a failure or a shutdown, Venus aims to retain as many as possible of the cached files on the local disk, but it cannot assume that the callback promise tokens are correct, since some callbacks may have been missed. Before the first use of each cached file or directory after a restart, Venus therefore generates a cache validation request containing the file modification timestamp to the server that is the custodian of the file. If the timestamp is current, the server responds with *valid* and the token is reinstated. If the timestamp shows that the file is out of date, then the server responds with *cancelled* and the token is set to *cancelled*. Callbacks must be renewed

Figure 12.14 Implementation of file system calls in AFS

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus. Open the local file and return the file descriptor to the application.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file. Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	→ ←	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

before an *open* if a time T (typically on the order of a few minutes) has elapsed since the file was cached without communication from the server. This is to deal with possible communication failures, which can result in the loss of callback messages.

This callback-based mechanism for maintaining cache consistency was adopted as offering the most scalable approach, following the evaluation in the prototype (AFS-1) of a timestamp-based mechanism similar to that used in NFS. In AFS-1, a Venus process holding a cached copy of a file interrogates the Vice process on each *open* to determine whether the timestamp on the local copy agrees with that on the server. The callback-based approach is more scalable because it results in communication between client and server and activity in the server only when the file has been updated, whereas the timestamp approach results in a client-server interaction on each *open*, even when there is a valid local copy. Since the majority of files are not accessed concurrently, and *read* operations predominate over *writes* in most applications, the *callback* mechanism results in a dramatic reduction in the number of client-server interactions.

The callback mechanism used in AFS-2 and later versions of AFS requires Vice servers to maintain some state on behalf of their Venus clients, unlike AFS-1, NFS and our file service model. The client-dependent state required consists of a list of the Venus

Figure 12.15 The main components of the Vice service interface

Fetch(fid) → *attr, data*	Returns the attributes (status) and, optionally, the contents of the file identified by *fid* and records a callback promise on it.
Store(fid, attr, data)	Updates the attributes and (optionally) the contents of a specified file.
Create() → *fid*	Creates a new file and records a callback promise on it.
Remove(fid)	Deletes the specified file.
SetLock(fid, mode)	Sets a lock on the specified file or directory. The mode of the lock may be shared or exclusive. Locks that are not removed expire after 30 minutes.
ReleaseLock(fid)	Unlocks the specified file or directory.
RemoveCallback(fid)	Informs the server that a Venus process has flushed a file from its cache.
BreakCallback(fid)	Call made by a Vice server to a Venus process; cancels the callback promise on the relevant file.

Note: Directory and administrative operations (*Rename*, *Link*, *Makedir*, *Removedir*, *GetTime*, *CheckToken* and so on) are not shown.

processes to which callback promises have been issued for each file. These callback lists must be retained over server failures – they are held on the server disks and are updated using atomic operations.

Figure 12.15 shows the RPC calls provided by AFS servers for operations on files (that is, the interface provided by AFS servers to Venus processes).

Update semantics • The goal of this cache-consistency mechanism is to achieve the best approximation to one-copy file semantics that is practicable without serious performance degradation. A strict implementation of one-copy semantics for UNIX file access primitives would require that the results of each *write* to a file be distributed to all sites holding the file in their cache before any further accesses can occur. This is not practicable in large-scale systems; instead, the callback promise mechanism maintains a well-defined approximation to one-copy semantics.

For AFS-1, the update semantics can be formally stated in very simple terms. For a client C operating on a file F whose custodian is a server S, the following guarantees of currency for the copies of F are maintained:

after a successful *open*:	*latest*(F, S)
after a failed *open*:	*failure*(S)
after a successful *close*:	*updated*(F, S)
after a failed *close*:	*failure*(S)

where *latest*(F, S) denotes a guarantee that the current value of F at C is the same as the value at S, *failure*(S) denotes that the *open* or *close* operation has not been performed at

S (and the failure can be detected by *C*), and *updated*(*F*, *S*) denotes that *C*'s value of *F* has been successfully propagated to *S*.

For AFS-2, the currency guarantee for *open* is slightly weaker, and the corresponding formal statement of the guarantee is more complex. This is because a client may open an old copy of a file after it has been updated by another client. This occurs if a callback message is lost, for example as a result of a network failure. But there is a maximum time, *T*, for which a client can remain unaware of a newer version of a file. Hence we have the following guarantee:

after a successful *open*: *latest*(*F*, *S*, *0*)
or (*lostCallback*(*S*, *T*) *and inCache*(*F*) *and* *latest*(*F*, *S*, *T*))

where *latest*(*F*, *S*, *T*) denotes that the copy of *F* seen by the client is no more than *T* seconds out of date, *lostCallback*(*S*, *T*) denotes that a callback message from *S* to *C* has been lost at some time during the last *T* seconds, and *inCache*(*F*) indicates that the file *F* was in the cache at *C* before the *open* operation was attempted. The above formal statement expresses the fact that the cached copy of *F* at *C* after an *open* operation is the most recent version in the system or a callback message has been lost (due to a communication failure) and the version that was already in the cache has been used; the cached version will be no more than *T* seconds out of date. (*T* is a system constant representing the interval at which callback promises must be renewed. At most installations, the value of *T* is about 10 minutes.)

In line with its goal – to provide a large-scale, UNIX-compatible distributed file service – AFS does not provide any further mechanism for the control of concurrent updates. The cache consistency algorithm described above comes into action only on *open* and *close* operations. Once a file has been opened, the client may access and update the local copy in any way it chooses without the knowledge of any processes on other workstations. When the file is closed, a copy is returned to the server, replacing the current version.

If clients in different workstations *open*, *write* and *close* the same file concurrently, all but the update resulting from the last *close* will be silently lost (no error report is given). Clients must implement concurrency control independently if they require it. On the other hand, when two client processes in the same workstation open a file, they share the same cached copy and updates are performed in the normal UNIX fashion – block by block.

Although the update semantics differ depending on the locations of the concurrent processes accessing a file and are not precisely the same as those provided by the standard UNIX file system, they are sufficiently close for the vast majority of existing UNIX programs to operate correctly.

12.4.3 Other aspects

AFS introduces several other interesting design developments and refinements that we outline here, together with a summary of performance evaluation results:

UNIX kernel modifications • We have noted that the Vice server is a user-level process running in the server computer and the server host is dedicated to the provision of an

AFS service. The UNIX kernel in AFS hosts is altered so that Vice can perform file operations in terms of file handles instead of the conventional UNIX file descriptors. This is the only kernel modification required by AFS, and it is necessary if Vice is not to maintain any client state (such as file descriptors).

Location database • Each server contains a copy of a fully replicated location database giving a mapping of volume names to servers. Temporary inaccuracies in this database may occur when a volume is moved, but they are harmless because forwarding information is left behind in the server from which the volume is moved.

Threads • The implementations of Vice and Venus make use of a non-preemptive threads package to enable requests to be processed concurrently at both the client (where several user processes may have file access requests in progress concurrently) and the server. In the client, the tables describing the contents of the cache and the volume database are held in memory that is shared between the Venus threads.

Read-only replicas • Volumes containing files that are frequently read but rarely modified, such as the UNIX */bin* and */usr/bin* directories of system commands and */man* directory of manual pages, can be replicated as read-only volumes at several servers. When this is done, there is only one read-write replica and all updates are directed to it. The propagation of the changes to the read-only replicas is performed after the update by an explicit operational procedure. Entries in the location database for volumes that are replicated in this way are one-to-many, and the server for each client request is selected on the bases of server loads and accessibility.

Bulk transfers • AFS transfers files between clients and servers in 64-kilobyte chunks. The use of such a large packet size is an important aid to performance, minimizing the effect of network latency. Thus the design of AFS enables the use of the network to be optimized.

Partial file caching • The need to transfer the entire contents of files to clients even when the application requirement is to read only a small portion of the file is an obvious source of inefficiency. Version 3 of AFS removed this requirement, allowing file data to be transferred and cached in 64-kbyte blocks while still retaining the consistency semantics and other features of the AFS protocol.

Performance • The primary goal of AFS is scalability, so its performance with large numbers of users is of particular interest. Howard *et al.* [1988] give details of extensive comparative performance measurements, which were undertaken using a specially developed *AFS benchmark* that has subsequently been widely used for the evaluation of distributed file systems. Not surprisingly, whole-file caching and the callback protocol led to dramatically reduced loads on the servers. Satyanarayanan [1989a] states that a server load of 40% was measured with 18 client nodes running a standard benchmark, against a load of 100% for NFS running the same benchmark. Satyanarayanan attributes much of the performance advantage of AFS to the reduction in server load deriving from the use of callbacks to notify clients of updates to files, compared with the timeout mechanism used in NFS for checking the validity of pages cached at clients.

Wide area support: • Version 3 of AFS supports multiple administrative cells, each with its own servers, clients, system administrators and users. Each cell is a completely autonomous environment, but a federation of cells can cooperate in presenting users

with a uniform, seamless file name space. The resulting system was widely deployed by the Transarc Corporation, and a detailed survey of the resulting performance usage patterns was published [Spasojevic and Satyanarayanan 1996]. The system was installed on over 1000 servers at over 150 sites. The survey showed cache hit ratios in the range of 96–98% for accesses to a sample of 32,000 file volumes holding 200 Gbytes of data.

12.5 Enhancements and further developments

Several advances have been made in the design of distributed file systems since the emergence of NFS and AFS. In this section, we describe advances that enhance the performance, availability and scalability of conventional distributed file systems. More radical advances are described elsewhere in the book, including the maintenance of consistency in replicated read-write filesystems to support disconnected operation and high availability in the Bayou and Coda systems (Sections 18.4.2 and 18.4.3) and a highly scalable architecture for the delivery of streams of real-time data with quality guarantees in the Tiger video file server (Section 20.6.1).

NFS enhancements • Several research projects have addressed the need for one-copy update semantics by extending the NFS protocol to include *open* and *close* operations and adding a callback mechanism to enable the server to notify clients of the need to invalidate cache entries. We describe two such efforts here; their results seem to indicate that these enhancements can be accommodated without undue complexity or extra communication costs.

Some recent efforts by Sun and other NFS developers have been directed at making NFS servers more accessible and useful in wide-area networks. While the HTTP protocol supported by web servers offers an effective and highly scalable method for making whole files available to clients throughout the Internet, it is less useful to application programs that require access to portions of large files or those that update portions of files. The WebNFS development (described below) makes it possible for application programs to become clients of NFS servers anywhere in the Internet (using the NFS protocol directly instead of indirectly through a kernel module). This, together with appropriate libraries for Java and other network programming languages, should offer the possibility of implementing Internet applications that share data directly, such as multi-user games or clients of large dynamic databases.

Achieving one-copy update semantics: The stateless server architecture of NFS brought great advantages in terms of robustness and ease of implementation, but it precluded the achievement of precise one-copy update semantics (the effects of concurrent writes by different clients to the same file are not guaranteed to be the same as they would be in a single UNIX system when multiple processes write to a local file). It also prevents the use of callbacks notifying clients of changes to files, and this results in frequent *getattr* requests from clients to check for file modification.

Two research systems have been developed that address these drawbacks. Spritely NFS [Srinivasan and Mogul 1989, Mogul 1994] is a version of the file system developed for the Sprite distributed operating system at Berkeley [Nelson *et al.* 1988]. Spritely NFS is an implementation of the NFS protocol with the addition of *open* and *close* calls.