7

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 SSL (Secure Sockets Layer) 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.

7.1 Introduction

Security measures must be incorporated into computer systems whenever they are potential targets for malicious or mischievous attacks. This is especially so for systems that handle financial transactions or confidential, classified or other information whose secrecy and integrity are critical. In Figure 7.1, we summarize the evolution of security needs in computer systems since they first arose with the advent of shared data in multi-user timesharing systems of the 1960s and 70s. Today the advent of wide-area, open distributed systems has resulted in a wide range of security issues.

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 for its employees and for 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 remains 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 shall describe do not in themselves ensure the security of a system. In Section 7.1.2, we outline the requirements for security in various simple electronic commerce scenarios, illustrating the need for policies in that

Figure 7.1 Historical context: the evolution of security needs

	1965–75	*1975–89*	*1990–99*	*Current*
Platforms	Multi-user timesharing computers	Distributed systems based on local networks	The Internet, wide-area services	The Internet + mobile devices
Shared resources	Memory, files	Local services (e.g. NFS), local networks	Email, web sites, Internet commerce	Distributed objects, mobile code
Security requirements	User identification and authentication	Protection of services	Strong security for commercial transactions	Access control for individual objects, secure mobile code
Security management environment	Single authority, single authorization database (e.g. /etc/passwd)	Single authority, delegation, replicated authorization databases (e.g. NIS)	Many authorities, no network-wide authorities	Per-activity authorities, groups with shared responsibilities

context. As an initial example, consider the security of a networked file server whose interface is accessible to clients. To ensure that access control to files is maintained, there would need to be a policy that all requests must include an authenticated user identity.

The provision of mechanisms for the protection of data and other computer-based resources and for securing networked transactions is the concern of this chapter. We shall describe the mechanisms that enable security policies to be enforced in distributed systems. The mechanisms we shall describe are strong enough to resist the most determined attacks.

The distinction between security policies and security mechanisms is helpful when designing secure systems, but it is often difficult to be confident that a given set of security mechanisms fully implements the desired security policies. In Section 2.3.3, we introduced a security model that is designed to help in analysing the potential security threats in a distributed system. We can summarize the security model of Chapter 2 as follows:

- Processes encapsulate resources (such as programming language-level objects and other system-defined resources) and allow clients to access them through their interfaces. Principals (users or other processes) can be explicitly authorized to operate on resources. Resources must be protected against unauthorized access.
- 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.

That security model identifies the features of distributed systems that expose them to attacks. In this chapter, we shall detail these attacks and the security techniques that are available for defeating them.

The emergence of cryptography into the public domain ◊ Cryptography provides the basis for most computer security mechanisms. 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 [1967, 1983, 1991] and Simon Singh [1999]. Whitfield Diffie, one of the inventors of public-key cryptography, has written with first-hand knowledge on the recent history and politics of cryptography [Diffie 1988, Diffie and Landau 1998] and in the preface to Schneier's book [1996].

But 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 published in many books, journals and conferences. The publication of Schneier's book *Applied Cryptography* [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 remains the

Figure 7.2 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

definitive reference on most aspects of modern cryptography. Menezes *et al.* [1997] also provides a good practical handbook with a strong theoretical basis.

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. This resulted in 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 standard encryption algorithm 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 familiar names for principals and attackers helps to clarify and bring to life descriptions of security protocols and potential attacks on them, which is an important step towards identifying their weaknesses. The names shown in Figure 7.2 are used extensively in the security literature and we shall 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].

7.1.1 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 – the acquisition of information by unauthorized recipients;

Tampering – the unauthorized alteration of information;

Vandalism – 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 him 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 forty-two 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 have focused 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 the 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 analyse 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].

7.1.2 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 using the Web and they are delivered to them through an appropriate delivery mechanism. Software and other digital products (such as recordings and videos) can be delivered by downloading through the Internet. 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. For example, most web pages are not currently charged for, but the development of the Web as a high-quality publishing medium surely depends upon the extent to which information suppliers can obtain payments from consumers of the information. The use of the Internet for voice and videoconferencing provides another example of a service that is likely to

be supplied only when it is paid for by end-users. 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 a fraudulent vendor who obtains 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 the customer obtains 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 that they intended to deal with.
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 in the case that 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 account holder and the bank as the vendor, but here there certainly *is* the need to:

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 7.6 the *de facto* standard security protocol used in most electronic commerce – Secure Sockets Layer (SSL) – and Millicent, a protocol specifically designed for micro-transactions.

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 7.5.2.

7.1.3 Designing secure systems

Immense strides have been made in recent years in the development of cryptographic techniques and their application, yet the design of 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 must be to exclude all bugs from his program. In neither case is there a concrete method to ensure the goals 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.cdk3.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 260 shows a set of useful assumptions and design guidelines. These assumptions underly the thinking behind the techniques that we shall 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 informal argument, or better, it can take the form of 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 is strong enough to withstand almost any attack, but their use incurs costs and inconvenience:

- a cost (in computational effort and in 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 can be quantified and selected based on the estimated cost of attacking them. The relatively low-cost techniques employed in the Millicent protocol for small commercial transactions described in Section 7.6.4 provide an example.

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 7.3 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

7.2 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 7.3 and 7.4. We shall use the familiar names for principals introduced in Figure 7.2 and the notations for encrypted and signed items shown in Figure 7.3.

7.2.1 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 a 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* – 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 he has 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.

7.2.2 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 they 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 reads the encrypted messages using the corresponding decryption function $D(K_{AB}, M)$.

Bob can now read the original message M. If the message makes sense when it is decrypted by Bob, 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 shall 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 7.3, 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 7.6.2. 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, described in Section 7.6.1.

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 *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 7.2.3.)
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). So Alice sends $keyname, \{K_{AB}\}_{K_{Bpub}}$ to Bob.
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 conventional signatures, 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

Figure 7.4 Alice's bank account certficate

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

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; the signature can be decrypted by any recipient using the corresponding 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 7.2.3.

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.

7.2.3 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 7.4). Alice could use this certificate when shopping to certify that she has an account with Bob's Bank. The certificate is signed in Bob's private key K_{Bpriv}. A vendor Carol can accept such a certificate for charging items to Alice's account provided 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

Figure 7.5 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*:	$\{Digest(field\ 2 + field\ 3)\}_{K_{Fpriv}}$

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.

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. Then Fred would issue a *public-key certificate* for Bob (Figure 7.5).

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 am 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 Alice 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 shall return to these requirements in Section 7.4.4.

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 or 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 a revoked certificate. 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.

7.2.4 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 [Stallings 1998b].

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, her identity is authenticated and a protection domain is created for the processes that she will run. Conceptually, the domain includes all of the access rights that the principal possesses, including any rights that she acquires

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. These are *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 act 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 (1) to put the illicit key holder in jail – not always feasible on a timescale that will prevent them doing damage, or (2) 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 the capability.

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 7.2.5).

It is interesting to note the similarity between capabilities and certificates. Consider Alice's certificate of ownership of her bank account introduced in Section 7.2.3. 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 to perform all the operations on Alice's bank account permitted to account holders, provided that the requester can be proven to be Alice.

Access control lists: A list is stored with each resource, with 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 1998b] with a model for ORBs to provide secure communication, authentication, access control with credentials, ACLs and auditing; these are described further in Section 17.3.4.

7.2.5 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 7.2.4). 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 its 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 a role certificate associating a principle with a named role in a specific task or organization [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 that the print server's code is subsequently 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.

7.2.6 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 SSL) 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.

7.3 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 into the original plaintext. Other principals are unable to decipher the message unless they 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:

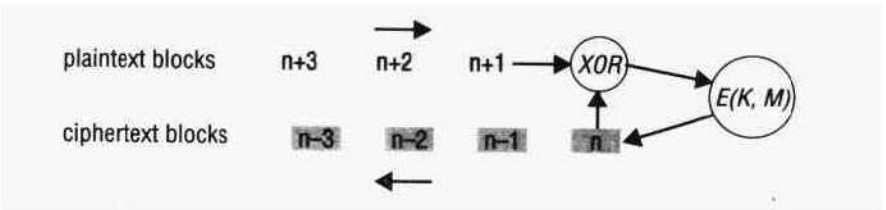
$$D(K, E(K, M)) = M$$

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 shall 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.

Figure 7.6 Cipher block chaining



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 its inverse unless a secret is known. It was the possibility of finding such functions and using them 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 functions of large numbers as trap-door functions.

The pair of keys needed for asymmetric algorithms is derived from a common root. For the RSA algorithm, described in Section 7.3.2, 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.

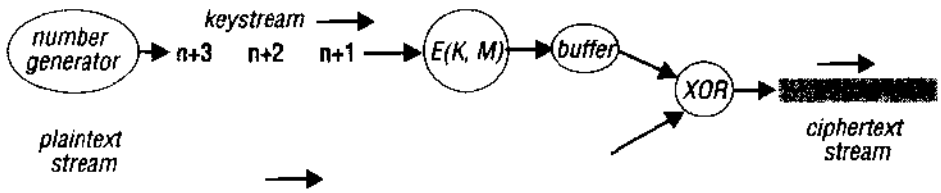
One of the pair of keys is used for encryption. For RSA, the encryption function obscures the plaintext by treating each block of bits as a binary number and raising it to the power of the key, modulo N. The resulting number is the corresponding ciphertext block.

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 shall discuss safe sizes for N in Section 7.3.2.

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.

Figure 7.7 Stream cipher



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 7.6). 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 *idempotent* – two applications of it produce the original value.

CBC is intended to prevent identical portions of plaintext encrypting to identical pieces of cipher text. But there is a weakness at the start of each sequence of blocks – if 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 a 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 Chapter15, 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 7.7). 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 the 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 this could be 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 and 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 describe 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 7.5.1.

Readers who do not require an understanding of the operation of cryptographic algorithms may omit Sections 7.3.1 and 7.3.2.

Figure 7.8 TEA encryption function

```
void encrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];                          1
    unsigned long delta = 0x9e3779b9, sum = 0; int n;                2
    for (n= 0; n < 32; n++) {                                        3
        sum += delta;                                                4
        y += ((z << 4) + k[0]) ^ (z+sum) ^ ((z >> 5) + k[1]);        5
        z += ((y << 4) + k[2]) ^ (y+sum) ^ ((y >> 5) + k[3]);        6
    }
    text[0] = y; text[1] = z;                                        7
}
```

7.3.1 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 has been 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 and is probably the most effective symmetric block encryption algorithm 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 be adopted as a new Advanced Encryption Standard (AES). We give some information on the progress of this activity below.

TEA ◊ The design principles for symmetric algorithms outlined above are illustrated well in the Tiny Encryption Algorithm developed at Cambridge University [Wheeler

Figure 7.9 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;
}
```

Figure 7.10 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 */
    }
}
```

and Needham 1994]. The encryption function, programmed in C, is given in its entirety in Figure 7.8.

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 7.9.

This short program provides secure and reasonably fast secret-key encryption. Its performance has been measured at approximately three times that of 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 they have addressed in a subsequent note [Wheeler and Needham 1997].

To illustrate its use, Figure 7.10 shows a simple procedure that uses TEA to encrypt or decrypt between 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 80s, but it was implemented in fast VLSI hardware and can easily be incorporated into network interface and other communication chips.

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 I]. A consortium of Internet users ran a client application program on a number of computers (PCs and other workstations) that grew from 1000 to 14,000 [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, which 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 twelve 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, providing adequate strength 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 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 analysed, and no significant weaknesses have been found. It performs encryption and decryption at approximately three times the speed of DES.

AES ◊ In 1997, the US National Institute for Standards and Technology (NIST) issued an invitation for proposals of a secure and efficient algorithm to be adopted as a new Advanced Encryption Standard (AES) [NIST 1999]. The evaluation will continue until May 2000, when a preliminary standard will be selected, and the final standard will be published in 2001.

Fifteen algorithms were submitted by the cryptographic research community in response to the initial AES invitation. Following an intensive technical review, five of

them were selected to proceed to a second phase of evaluation. All of the candidates support key sizes of 128, 192 and 256 bit, and the final five are all of high performance. Once it is announced, the AES seems likely to become the most widely used symmetric encryption algorithm.

7.3.2 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 trap-door.

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 shall 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 to compute.

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 e, d:

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 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. Factorization 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.

Much faster computers and better factorization methods have been developed since the 1978 claim of Rivest *et al.* The RSA Corporation issued a series of challenges to perform factorizations of numbers of more than 100 decimal digits [www.rsasecurity.com II]. Numbers up to 155 decimal digits in length have been successfully factored by a distributed consortium of Internet users working in a similar manner to the DES cracking effort described in the preceding section. At the time of writing, the status is that numbers of 155 digits (or ~ 500 binary digits) have been successfully factorized, casting doubt on the security of 512-bit keys. The processing effort needed for factoring a 155-digit number hasn't been published, but a 140-digit number took in the order of 2000 MIPS-years in February 1999. A MIPS-year is the processing effort generated by a processor with a speed of a million instructions per second running for a year. Today's single-processor chips have speeds of the order of 200 MIPS, enabling them to complete this task in about ten years, but distributed factorizing applications and parallel processors could reduce this figure to days. 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.

All of 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 would be vulnerable if and when 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.

7.3.3 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 SSL case study in Section 7.6.3.

8

DISTRIBUTED FILE SYSTEMS

Distributed file systems support the sharing of information in the form of files 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. 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 in an intranet.

We describe the designs of two distributed file systems that have been in widespread use for a decade or more:

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

These case studies illustrate a range of design solutions to the emulation of a UNIX file system interface with differing degrees of scalability and fault tolerance. Each necessitates some deviation from the *strict* emulation of UNIX *one-copy file update semantics*.

Recent design advances for distributed file systems have exploited the higher-bandwidth connectivity of switched local networks and new modes of data organization on disk to achieve very high-performance, fault-tolerant and highly scalable file systems.

8.1 Introduction

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. It is achieved in the large scale on the Internet primarily by the use of web servers, but 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 *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.

We begin with a brief overview of the spectrum of distributed and non-distributed storage systems. 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 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 becomes impractical for rapidly changing objects, and several more direct approaches

Figure 8.1 Storage systems and their properties

	Sharing	*Persis-tence*	*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 (Ch. 16)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Persistent distributed object store	✓	✓	✓	✓	PerDiS, Khazana

have therefore 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.

Recent developments for the distribution of stored information include distributed shared memory (DSM) systems and persistent object stores. DSM is described in detail in Chapter 16. It provides an emulation of a shared memory by the replication of memory pages or segments at each host. 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 Object Service (see Chapter 17) and persistent extensions to Java [Jordan 1996, java.sun.com IV]. Some recent research has resulted 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]).

Figure 8.1 provides an overview of the properties of the various types of storage system we have mentioned here. 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 8.1) – 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 8.1 – we discuss these mechanisms and the degree to which they deviate from strict consistency in Sections 8.3 and 8.4.

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

Figure 8.2 File system modules

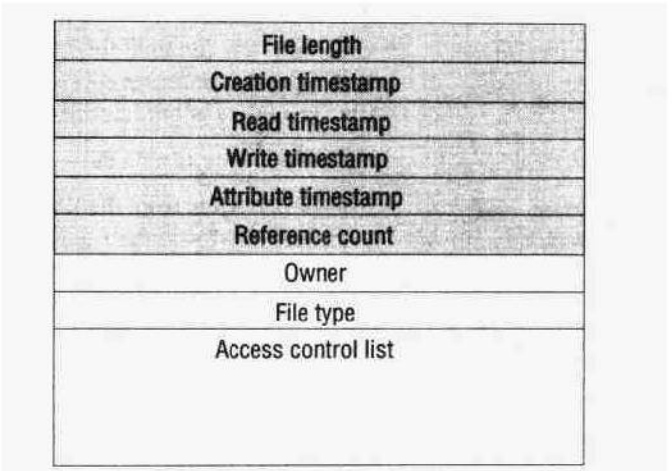
Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and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

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 detail in Chapter 16. Persistent object systems vary considerably in their approach to the use of 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 have 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 8.1.1 and the requirements for distributed file systems in Section 8.1.2. Section 8.1.3 introduces the case studies that will be used throughout the chapter. In Section 8.2, we define an abstract model for a basic distributed file service, including a set of programming interfaces. The Sun NFS system is described in Section 8.3; it shares many of the features of the abstract model. In Section 8.4, we describe the Andrew File System – a widely used system that employs substantially different caching and consistency mechanisms. Section 8.5 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 14 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 which provides a weakly consistent form of replication for high availability is also covered in Chapter 14. Chapter 15 covers the Tiger video file server, which is designed to provide timely delivery of streams of data to large numbers of clients.

Figure 8.3 File attribute record structure



8.1.1 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 8.3. The shaded attributes are managed by the file system and are not normally updatable by user programs.

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 other persistent information used by the file system.

Figure 8.2 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.

Figure 8.4 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)	Gets the file attributes for file *name* into *buffer*.

File system operations ◊ Figure 8.4 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 library procedures such as the C Standard Input-Output Library or 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.

8.1.2 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 1.4.7. 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 12. 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 14, 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 5.2.4). 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 14.

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 shall discuss the impact of these requirements in our case study descriptions.

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.

8.1.3 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 which emulates a conventional file system interface for application programs and server modules, which 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 Microsystem'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 successfully 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 considerable success both technically and commercially.

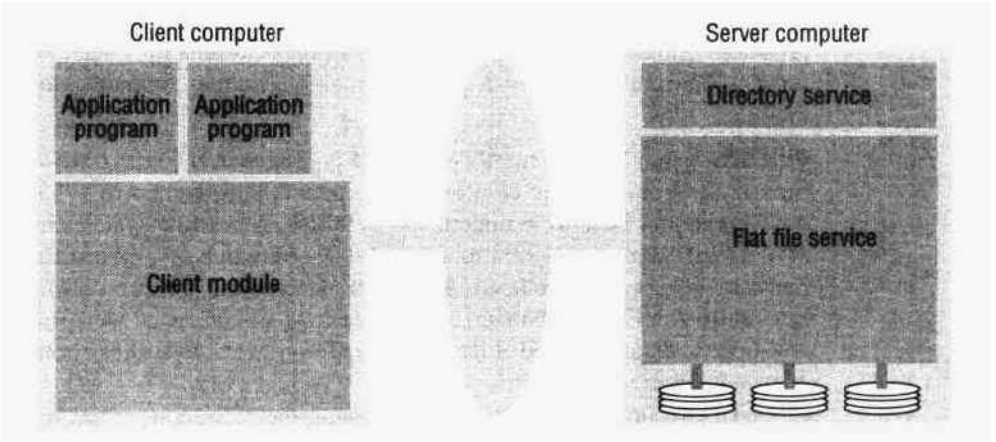
To encourage its adoption as a standard, the definitions of the key interfaces were placed in the public domain [Sun 1989], enabling other vendors to produce implementations, and the source code for a reference implementation was made available to other computer vendors under licence. It is now supported 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. Typically, every computer has NFS client and server modules installed in its system kernel, at least in the case of UNIX 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 current operating systems and platforms, including Windows 95, Windows NT, MacOS and VMS as well as Linux and almost 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 the 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 (henceforth abbreviated AFS) reflects an intention to support information sharing on a large scale by minimizing client–server communication. This was achieved by transferring whole files (or for large files, 64-kbyte chunks) between server and client computers and caching them at clients until the server receives a more up-to-date version. The campus computing network that Andrew served was expected to grow to include between 5000 and 10,000 workstations during the lifetime of the system. We shall describe AFS-2, the first 'production' implementation, following the descriptions by Satyanarayanan

Figure 8.5 File service architecture



[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. More recently, a public-domain implementation of AFS has become available in the Linux operating system [Linux AFS]. In 1991, AFS was supporting approximately 800 workstations serviced by approximately forty servers at CMU, with additional clients and servers at remote sites connected via the Internet. 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 8.5.

8.2 File service architecture

The scope for open, configurable systems is enhanced if the file service is structured as three components – a *flat file service*, a *directory service* and a *client module*. The relevant modules and their relationships are shown in Figure 8.5. 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

Figure 8.6 Flat file service operations

Read(FileId, i, n) → *Data* — throws *BadPosition*	If $1 \le i \le Length(File)$: 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 $1 \le i \le Length(File)+1$: 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 8.3).

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 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 8.6 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 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 8.4 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 the result of 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 7.2.4), 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. We have seen in Chapter 7 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 8.7 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 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 three operations for altering directories: *AddName*, *ReName* 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

Figure 8.7 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, File) — 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: the entry containing *Name* is removed 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*.

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 sub-strings and symbols denoting variable characters or repeated occurrences of characters or sub-strings.

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.

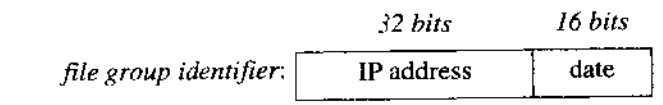
A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services that we have defined. A tree-structured network of directories is constructed with files at the leaves and directories at the other nodes of the tree. The root of the tree is a directory 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 grouping ◊ 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. File groups were originally introduced to support facilities for moving collections of files stored on removable disk cartridges 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.

8.3 Sun Network File System

Figure 8.8 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 calling. Sun's RPC system, described in Section 5.3.1, 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