Secure channels and access control require mechanisms to distribute cryptographic keys, but also mechanisms to add and remove users from a system. These topics are covered by what is known as security management. In a separate section, we discuss issues dealing with managing cryptographic keys, secure group management, and handing out certificates that prove the owner is entitled to access specified resources.

9.1 INTRODUCTION TO SECURITY

We start our description of security in distributed systems by taking a look at some general security issues. First, it is necessary to define what a secure system is. We distinguish security *policies* from security *mechanisms*, and take a look at the Globus wide-area system for which a security policy has been explicitly formulated. Our second concern is to consider some general design issues for secure systems. Finally, we briefly discuss some cryptographic algorithms, which play a key role in the design of security protocols.

9.1.1 Security Threats, Policies, and Mechanisms

Security in a computer system is strongly related to the notion of dependability. Informally, a dependable computer system is one that we justifiably trust to deliver its services (Laprie, 1995). As mentioned in Chap. 7, dependability includes availability, reliability, safety, and maintainability. However, if we are to put our trust in a computer system, then confidentiality and integrity should also be taken into account. Confidentiality refers to the property of a computer system whereby its information is disclosed only to authorized parties. Integrity is the characteristic that alterations to a system's assets can be made only in an authorized way. In other words, improper alterations in a secure computer system should be detectable and recoverable. Major assets of any computer system are its hardware, software, and data.

Another way of looking at security in computer systems is that we attempt to protect the services and data it offers against **security threats**. There are four types of security threats to consider (Pfleeger, 2003):

1. Interception
2. Interruption
3. Modification
4. Fabrication

The concept of interception refers to the situation that an unauthorized party has gained access to a service or data. A typical example of interception is where

communication between two parties has been overheard by someone else. Interception also happens when data are illegally copied, for example, after breaking into a person's private directory in a file system.

An example of interruption is when a file is corrupted or lost. More generally interruption refers to the situation in which services or data become unavailable, unusable, destroyed, and so on. In this sense, denial of service attacks by which someone maliciously attempts to make a service inaccessible to other parties is a security threat that classifies as interruption.

Modifications involve unauthorized changing of data or tampering with a service so that it no longer adheres to its original specifications. Examples of modifications include intercepting and subsequently changing transmitted data, tampering with database entries, and changing a program so that it secretly logs the activities of its user.

Fabrication refers to the situation in which additional data or activity are generated that would normally not exist. For example, an intruder may attempt to add an entry into a password file or database. Likewise, it is sometimes possible to break into a system by replaying previously sent messages. We shall come across such examples later in this chapter.

Note that interruption, modification, and fabrication can each be seen as a form of data falsification.

Simply stating that a system should be able to protect itself against all possible security threats is not the way to actually build a secure system. What is first needed is a description of security requirements, that is, a security policy. A **security policy** describes precisely which actions the entities in a system are allowed to take and which ones are prohibited. Entities include users, services, data, machines, and so on. Once a security policy has been laid down, it becomes possible to concentrate on the **security mechanisms** by which a policy can be enforced. Important security mechanisms are:

1. Encryption
2. Authentication
3. Authorization
4. Auditing

Encryption is fundamental to computer security. Encryption transforms data into something an attacker cannot understand. In other words, encryption provides a means to implement data confidentiality. In addition, encryption allows us to check whether data have been modified. It thus also provides support for integrity checks.

Authentication is used to verify the claimed identity of a user, client, server, host, or other entity. In the case of clients, the basic premise is that before a service starts to perform any work on behalf of a client, the service must learn the

client's identity (unless the service is available to all). Typically, users are authenticated by means of passwords, but there are many other ways to authenticate clients.

After a client has been authenticated, it is necessary to check whether that client is authorized to perform the action requested. Access to records in a medical database is a typical example. Depending on who accesses the database, permission may be granted to read records, to modify certain fields in a record, or to add or remove a record.

Auditing tools are used to trace which clients accessed what, and which way. Although auditing does not really provide any protection against security threats, audit logs can be extremely useful for the analysis of a security breach, and subsequently taking measures against intruders. For this reason, attackers are generally keen not to leave any traces that could eventually lead to exposing their identity. In this sense, logging accesses makes attacking sometimes a riskier business.

Example: The Globus Security Architecture

The notion of security policy and the role that security mechanisms play in distributed systems for enforcing such policies is often best explained by taking a look at a concrete example. Consider the security policy defined for the Globus wide-area system (Chervenak et al., 2000). Globus is a system supporting large-scale distributed computations in which many hosts, files, and other resources are simultaneously used for doing a computation. Such environments are also referred to as computational grids (Foster and Kesselman, 2003). Resources in these grids are often located in different administrative domains that may be located in different parts of the world.

Because users and resources are vast in number and widely spread across different administrative domains, security is essential. To devise and properly use security mechanisms, it is necessary to understand what exactly needs to be protected, and what the assumptions are with respect to security. Simplifying matters somewhat, the security policy for Globus entails the following eight statements, which we explain below (Foster et al., 1998):

1. The environment consists of multiple administrative domains.
2. Local operations (i.e., operations that are carried out only within a single domain) are subject to a local domain security policy only.
3. Global operations (i.e., operations involving several domains) require the initiator to be known in each domain where the operation is carried out.
4. Operations between entities in different domains require mutual authentication.
5. Global authentication replaces local authentication.

6. Controlling access to resources is subject to local security only.
7. Users can delegate rights to processes.
8. A group of processes in the same domain can share credentials.

Globus assumes that the environment consists of multiple administrative domains, where each domain has its own local security policy. It is assumed that local policies cannot be changed just because the domain participates in Globus, nor can the overall policy of Globus override local security decisions. Consequently, security in Globus will restrict itself to operations that affect multiple domains.

Related to this issue is that Globus assumes that operations that are entirely local to a domain are subject only to that domain's security policy. In other words, if an operation is initiated and carried out within a single domain, all security issues will be carried out using local security measures only. Globus will not impose additional measures.

The Globus security policy states that requests for operations can be initiated either globally or locally. The initiator, be it a user or process acting on behalf of a user, must be locally known within each domain where that operation is carried out. For example, a user may have a global name that is mapped to domain-specific local names. How exactly that mapping takes place is left to each domain.

An important policy statement is that operations between entities in different domains require mutual authentication. This means, for example, that if a user in one domain makes use of a service from another domain, then the identity of the user will have to be verified. Equally important is that the user will have to be assured that he is using a service he thinks he is using. We return to authentication, extensively, later in this chapter.

The above two policy issues are combined into the following security requirement. If the identity of a user has been verified, and that user is also known locally in a domain, then he can act as being authenticated for that local domain. This means that Globus requires that its systemwide authentication measures are sufficient to consider that a user has already been authenticated for a remote domain (where that user is known) when accessing resources in that domain. Additional authentication by that domain should not be necessary.

Once a user (or process acting on behalf of a user) has been authenticated, it is still necessary to verify the exact access rights with respect to resources. For example, a user wanting to modify a file will first have to be authenticated, after which it can be checked whether or not that user is actually permitted to modify the file. The Globus security policy states that such access control decisions are made entirely local within the domain where the accessed resource is located.

To explain the seventh statement, consider a mobile agent in Globus that carries out a task by initiating several operations in different domains, one after another. Such an agent may take a long time to complete its task. To avoid having

to communicate with the user on whose behalf the agent is acting. Globus requires that processes can be delegated a subset of the user's rights. As a consequence, by authenticating an agent and subsequently checking its rights, Globus should be able to allow an agent to initiate an operation without having to contact the agent's owner.

As a final policy statement, Globus requires that groups of processes running with a single domain and acting on behalf of the same user may share a single set of credentials. As will be explained below, credentials are needed for authentication. This statement essentially opens the road to scalable solutions for authentication by not demanding that each process carries its own unique set of credentials.

The Globus security policy allows its designers to concentrate on developing an overall solution for security. By assuming each domain enforces its own security policy, Globus concentrates only on security threats involving multiple domains. In particular, the security policy indicates that the important design issues are the representation of a user in a remote domain, and the allocation of resources from a remote domain to a user or his representative. What Globus therefore primarily needs, are mechanisms for cross-domain authentication, and making a user known in remote domains.

For this purpose, two types of representatives are introduced. A user proxy is a process that is given permission to act on behalf of a user for a limited period of time. Resources are represented by resource proxies. A resource proxy is a process running within a specific domain that is used to translate global operations on a resource into local operations that comply with that particular domain's security policy. For example, a user proxy typically communicates with a resource proxy when access to that resource is required.

The Globus security architecture esseatially consists of entities such as users, user proxies, resource proxies, and general processes. These entities are located in domains and interact with each other. In particular, the security architecture defines four different protocols, as illustrated in Fig. 9-1 [see also Foster et al. (1998)].

The first protocol describes precisely how a user can create a user proxy and delegate rights to that proxy. In particular, in order to let the user proxy act on behalf of its user, the user gives the proxy an appropriate set of credentials.

The second protocol specifies how a user proxy can request the allocation of a resource in a remote domain. In essence, the protocol tells a resource proxy to create a process in the remote domain after mutual authentication has taken place. That process represents the user (just as the user proxy did), but operates in the same domain as the requested resource. The process is given access to the resource subject to the access control decisions local to that domain.

A process created in a remote domain may initiate additional computations in other domains. Consequently, a protocol is needed to allocate resources in a remote domain as requested by a process other than a user proxy. In the Globus system, this type of allocation is done via the user proxy, by letting a process have its

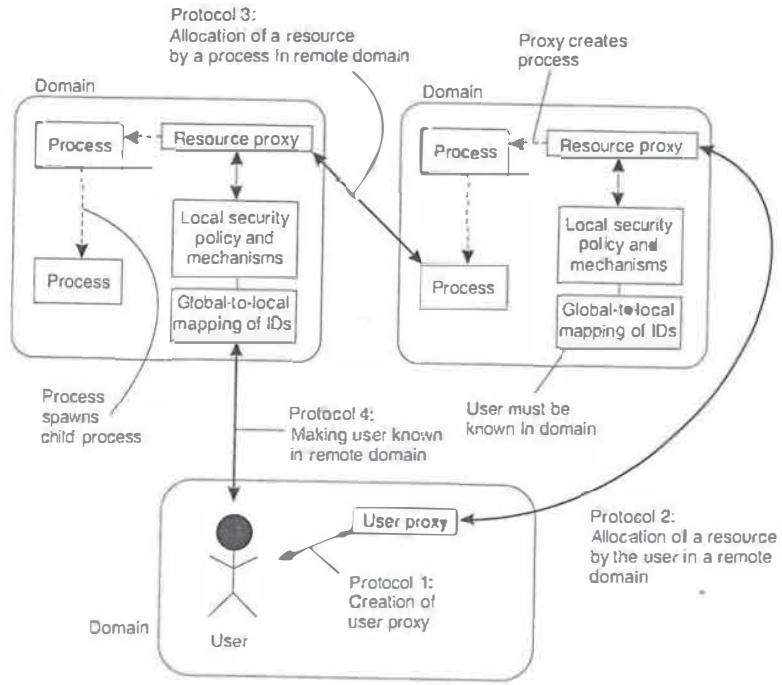

Figure 9-1. The Globus security architecture.

associated user proxy request the allocation of resources, essentially following the second protocol.

The fourth and last protocol in the Globus security architecture is the way a user can make himself known in a domain. Assuming that a user has an account in a domain, what needs to be established is that the systemwide credentials as held by a user proxy are automatically converted to credentials that are recognized by the specific domain. The protocol prescribes how the mapping between the global credentials and the local ones can be registered by the user in a mapping table local to that domain.

Specific details of each protocol are described in Foster et al. (1998). The important issue here is that the Globus security architecture reflects its security policy as stated above. The mechanisms used to implement that architecture, in particular the above mentioned protocols, are common to many distributed systems, and are discussed extensively in this chapter. The main difficulty in designing secure distributed systems is not so much caused by security mechanisms, but by

deciding on how those mechanisms are to be used to enforce a security policy. In the next section, we consider some of these design decisions.

9.1.2 Design Issues

A distributed system, or any computer system for that matter, must provide security services by which a wide range of security policies can be implemented. There are a number of important design issues that need to be taken into account when implementing general-purpose security services. In the following pages, we discuss three of these issues: focus of control, layering of security mechanisms, and simplicity [see also Gollmann (2006)].

Focus of Control

When considering the protection of a (possibly distributed) application, there are essentially three different approaches that can be followed, as shown in Fig. 9-2. The first approach is to concentrate directly on the protection of the data that is associated with the application. By direct, we mean that irrespective of the various operations that can possibly be performed on a data item, the primary concern is to ensure data integrity. Typically, this type of protection occurs in database systems in which various integrity constraints can be formulated that are automatically checked each time a data item is modified [see, for example, Doorn and Rivero (2002)].

The second approach is to concentrate on protection by specifying exactly which operations may be invoked, and by whom, when certain data or resources are to be accessed. In this case, the focus of control is strongly related to access control mechanisms, which we discuss extensively later in this chapter. For example, in an object-based system, it may be decided to specify for each method that is made available to clients which clients are permitted to invoke that method. Alternatively, access control methods can be applied to an entire interface offered by an object, or to the entire object itself. This approach thus allows for various granularities of access control.

A third approach is to focus directly on users by taking measures by which only specific people have access to the application, irrespective of the operations they want to carry out. For example, a database in a bank may be protected by denying access to anyone except the bank's upper management and people specifically authorized to access it. As another example, in many universities, certain data and applications are restricted to be used by faculty and staff members only, whereas access by students is not allowed. In effect, control is focused on defining roles that users have, and once a user's role has been verified, access to a resource is either granted or denied. As part of designing a secure system, it is thus necessary to define roles that people may have, and provide mechanisms to support role-based access control. We return to roles later in this chapter.

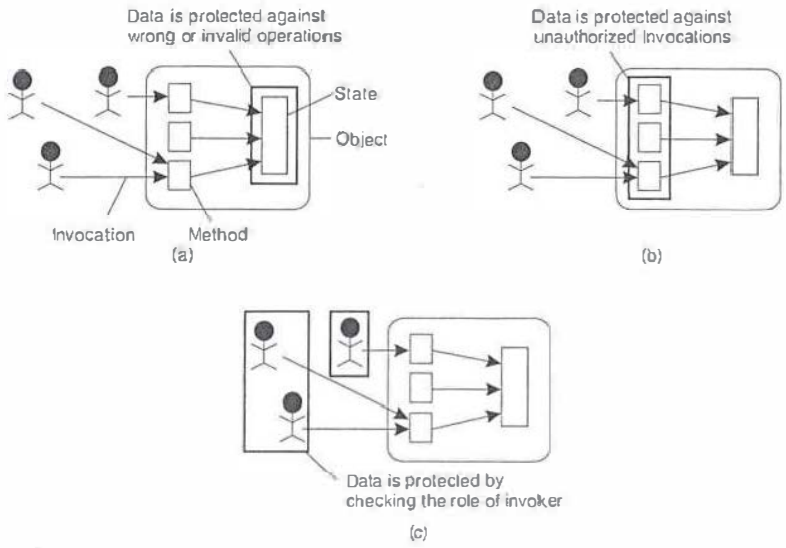

Figure 9-2. Three approaches for protection against security threats. (a) Protection against invalid operations (b) Protection against unauthorized invocations. (c) Protection against unauthorized users.

Layering of Security Mechanisms

An important issue in designing secure systems is to decide at which level security mechanisms should be placed. A level in this context is related to the logical organization of a system into a number of layers. For example, computer networks are often organized into layers following some reference model, as we discussed in Chap. 4. In Chap. 1, we introduced the organization of distributed systems consisting of separate layers for applications, middleware, operating system services, and the operating system kernel. Combining the layered organization of computer networks and distributed systems, leads roughly to what is shown in Fig. 9-3.

In essence, Fig. 9-3 separates general-purpose services from communication services. This separation is important for understanding the layering of security in distributed systems and, in particular, the notion of trust. The difference between trust and security is important. A system is either secure or it is not (taking various probabilistic measures into account), but whether a client considers a system to be secure is a matter of trust (Bishop, 2003). Security is technical; trust is emotional. In which layer security mechanisms are placed depends on the trust a client has in how secure the services are in a particular layer.

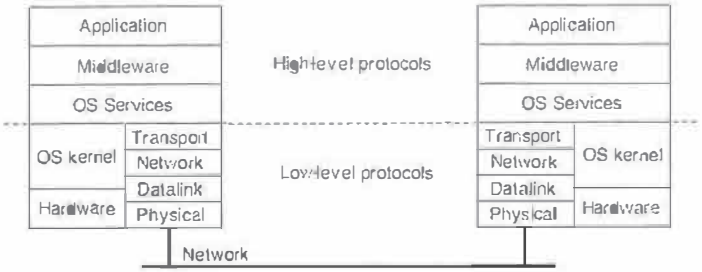

Figure 9-3. The logical organization of a distributed system into several layers.

As an example, consider an organization located at different sites that are connected through a communication service such as **Switched Multi-megabit Data Service (SMDS)**. An SMDS network can be thought of as a link-level backbone connecting various local-area networks at possibly geographically dispersed sites, as shown in Fig. 9-4.

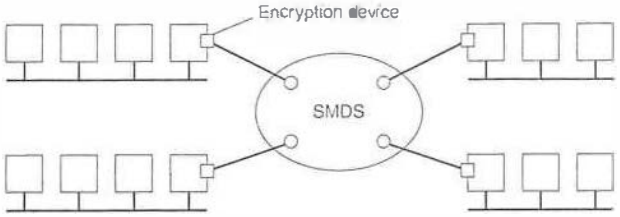

Figure 9-4. Several sites connected through a wide-area backbone service.

Security can be provided by placing encryption devices at each SMDS router, as also shown in Fig. 9-4. These devices automatically encrypt and decrypt packets that are sent between sites, but do not otherwise provide secure communication between hosts at the same site. If Alice at site *A* sends a message to Bob at site *B*, and she is worried about her message being intercepted, she must at least trust the encryption of intersite traffic to work properly. This means, for example, that she must trust the system administrators at both sites to have taken the proper measures against tampering with the devices.

Now suppose that Alice does not trust the security of intersite traffic. She may then decide to take her own measures by using a transport-level security service such as SSL. **SSL** stands for **Secure Sockets Layer** and can be used to securely send messages across a TCP connection. We will discuss the details of SSL later Chap. 12 when discussing Web-based systems. The important thing to observe here is that SSL allows Alice to set up a secure connection to Bob. All transport-level messages will be encrypted—and at the SMDS level as well, but that is of no concern to Alice. In this case, Alice will have to put her trust into SSL. In other words, she believes that SSL is secure.

In distributed systems, security mechanisms are often placed in the middleware layer. If Alice does not trust SSL, she may want to use a local secure RPC service. Again, she will have to trust this RPC service to do what it promises, such as not leaking information or properly authenticating clients and servers.

Security services that are placed in the middleware layer of a distributed system can be trusted only if the services they rely on to be secure are indeed secure. For example, if a secure RPC service is partly implemented by means of SSL, then trust in the RPC service depends on how much trust one has in SSL. If SSL is not trusted, then there can be no trust in the security of the RPC service.

Distribution of Security Mechanisms

Dependencies between services regarding trust lead to the notion of a **Trusted Computing Base (TCB)**. A TCB is the set of all security mechanisms in a (distributed) computer system that are needed to enforce a security policy, and that thus need to be trusted. The smaller the TCB, the better. If a distributed system is built as middleware on an existing network operating system, its security may depend on the security of the underlying local operating systems. In other words, the TCB in a distributed system may include the local operating systems at various hosts.

Consider a file server in a distributed file system. Such a server may need to rely on the various protection mechanisms offered by its local operating system. Such mechanisms include not only those for protecting files against accesses by processes other than the file server, but also mechanisms to protect the file server from being maliciously brought down.

Middleware-based distributed systems thus require trust in the existing local operating systems they depend on. If such trust does not exist, then part of the functionality of the local operating systems may need to be incorporated into the distributed system itself. Consider a microkernel operating system, in which most operating-system services run as normal user processes. In this case, the file system, for instance, can be entirely replaced by one tailored to the specific needs of a distributed system, including its various security measures.

Consistent with this approach is to separate security services from other types of services by distributing services across different machines depending on amount of security required. For example, for a secure distributed file system, it may be possible to isolate the file server from clients by placing the server on a machine with a trusted operating system, possibly running a dedicated secure file system. Clients and their applications are placed on untrusted machines.

This separation effectively reduces the TCB to a relatively small number of machines and software components. By subsequently protecting those machines

against security attacks from the outside, overall trust in the security of the distributed system can be increased. Preventing clients and their applications direct access to critical services is followed in the **Reduced Interfaces for Secure System Components (RISSC)** approach, as described in Neumann (1995). In the RISSC approach, any security-critical server is placed on a separate machine isolated from end-user systems using low-level secure network interfaces, as shown in Fig. 9-5. Clients and their applications run on different machines and can access the secured server only through these network interfaces.

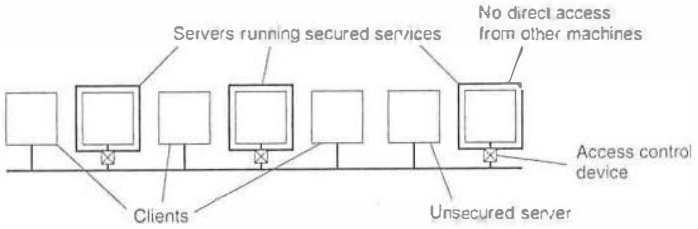

Figure 9-5. The principle of RISSC as applied to secure distributed systems.

Simplicity

Another important design issue related to deciding in which layer to place security mechanisms is that of simplicity. Designing a secure computer system is generally considered a difficult task. Consequently, if a system designer can use a few, simple mechanisms that are easily understood and trusted to work, the better it is.

Unfortunately, simple mechanisms are not always sufficient for implementing security policies. Consider once again the situation in which Alice wants to send a message to Bob as discussed above. Link-level encryption is a simple and easy-to-understand mechanism to protect against interception of intersite message traffic. However, much more is needed if Alice wants to be sure that only Bob will receive her messages. In that case, user-level authentication services are needed, and Alice may need to be aware of how such services work in order to put her trust in it. User-level authentication may therefore require at least a notion of cryptographic keys and awareness of mechanisms such as certificates, despite the fact that many security services are highly automated and hidden from users.

In other cases, the application itself is inherently complex and introducing security only makes matters worse. An example application domain involving complex security protocols (as we discuss later in this chapter) is that of digital payment systems. The complexity of digital paymem protocols is often caused by the fact that multiple parties need to communicate to make a payment. In these cases,

it is important that the underlying mechanisms that are used to implemem the protocols are relatively simple and easy to understand. Simplicity will contribute to the trust that end users will put into the application and, more importantly, will contribute to convincing the designers that the system has no security holes.

9.1.3 Cryptography

Fundamental to security in distributed systems is the use of cryptographic techniques. The basic idea of applying these techniques is simple. Consider a sender S wanting to transmit message m to a receiver R. To protect the message against security threats, the sender first encrypts it into an unintelligible message m', and subsequently sends m' to R. R, in turn, must decrypt the received message into its original form m.

Encryption and decryption are accomplished by using cryptographic methods parameterized by keys, as shown in Fig. 9-6. The original form of the message that is sent is called the **plaintext**, shown as P in Fig. 9-6; the encrypted form is referred to as the **ciphertext**, illustrated as C.

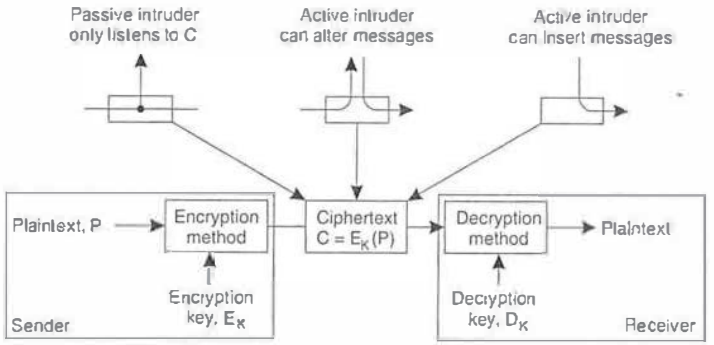

Figure 9-6. Intruders and eavesdroppers in communication.

To describe the various security protocols that are used in building security services for distributed systems, it is useful to have a notation to relate plaintext, ciphertext, and keys. Following the common notational conventions, we will use $C = E_K(P)$ to denote that the ciphertext C is obtained by encrypting the plaintext P using key K. Likewise, $P = D_K(C)$ is used to express the decryption of the ciphertext C using key K, resulting in the plaintext P.

Returning to our example shown in Fig. 9-6, while transferring a message as ciphertext C, there are three different attacks that we need to protect against, and for which encryption helps. First, an intruder may intercept the message without either the sender or receiver being aware that eavesdropping is happening. Of

course, if the transmitted message has been encrypted in such a way that it cannot be easily decrypted without having the proper key, interception is useless; the intruder will see only unintelligible data. (By the way, the fact alone that a message is being transmitted may sometimes be enough for an intruder to draw conclusions. For example, if during a world crisis the amount of traffic into the White House suddenly drops dramatically while the amount of traffic going into a certain mountain in Colorado increases by the same amount, there may be useful information in knowing that.)

The second type of attack that needs to be dealt with is that of modifying the message. Modifying plaintext is easy; modifying ciphertext that has been properly encrypted is much more difficult because the intruder will first have to decrypt the message before he can meaningfully modify it. In addition, he will also have to properly encrypt it again or otherwise the receiver may notice that the message has been tampered with.

The third type of attack is when an intruder inserts encrypted messages into the communication system, attempting to make R believe these messages came from S. Again, as we shall see later in this chapter, encryption can help protect against such attacks. Note that if an intruder can modify messages, he can also insert messages.

There is a fundamental distinction between different cryptographic systems, based on whether or not the encryption and decryption key are the same. In a **symmetric cryptosystem**, the same key is used to encrypt and decrypt a message. In other words,

$$P = D_K(E_K(P))$$

Symmetric cryptosystems are also referred to as secret-key or shared-key systems, because the sender and receiver are required to share the same key, and to ensure that protection works, this shared key must be kept secret: no one else is allowed to see the key. We will use the notation $K_{A,B}$ to denote a key shared by A and B.

In an **asymmetric cryptosystem**, the keys for encryption and decryption are different, but together form a unique pair. In other words, there is a separate key K_E for encryption and one for decryption, K_D, such that

$$P = D_{K_D}(E_{K_E}(P))$$

One of the keys in an asymmetric cryptosystem is kept private; the other is made public. For this reason, asymmetric cryptosystems are also referred to as **public-key systems**. In what follows, we use the notation K_A^+ to denote a public key belonging to A, and K_A^- as its corresponding private key.

Anticipating the detailed discussions on security protocols later in this chapter, which one of the encryption or decryption keys that is actually made public depends on how the keys are used. For example, if Alice wants to send a confidential message to Bob, she should use Bob's public key to encrypt the message. Because Bob is the only one holding the private decryption key, he is also the only person that can decrypt the message.

On the other hand, suppose that Bob wants to know for sure that the message he just received actually came from Alice. In that case, Alice can keep her encryption key private to encrypt the messages she sends. If Bob can successfully decrypt a message using Alice's public key (and the plaintext in the message has enough information to make it meaningful to Bob), he knows that message must have come from Alice, because the decryption key is uniquely tied to the encryption key. We return to such algorithms in detail below.

One final application of cryptography in distributed systems is the use of **hash functions**. A hash function H takes a message m of arbitrary length as input and produces a bit string h having a fixed length as output:

$$h = H(m)$$

A hash h is somewhat comparable to the extra bits that are appended to a message in communication systems to allow for error detection, such a cyclic-redundancy check (CRC).

Hash functions that are used in cryptographic systems have a number of essential properties. First, they are **one-way functions**, meaning that it is computationally infeasible to find the input m that corresponds to a known output h. On the other hand, computing h from m is easy. Second, they have the **weak collision resistance** property, meaning that given an input m and its associated output $h = H(m)$, it is computationally infeasible to find another, different input $m' \neq m$, such that $H(m) = H(m')$. Finally, cryptographic hash functions also have the **strong collision resistance** property, which means that, when given only H, it is computationally infeasible to find any two different input values m and m', such that $H(m) = H(m')$.

Similar properties must apply to any encryption function E and the keys that are used. Furthermore, for any encryption function E, it should be computationally infeasible to find the key K when given the plaintext P and associated ciphertext $C = E_K(P)$. Likewise, analogous to collision resistance, when given a plaintext P and a key K, it should be effectively impossible to find another key K' such that $E_K(P) = E_{K'}(P)$.

The art and science of devising algorithms for cryptographic systems has a long and fascinating history (Kahn, 1967), and building secure systems is often surprisingly difficult, or even impossible (Schneier, 2000). It is beyond the scope of this book to discuss any of these algorithms in detail. However, to give some impression of cryptography in computer systems, we will now briefly present three representative algorithms. Detailed information on these and other cryptographic algorithms can be found in Ferguson and Schneier (2003), Menezes et al., (1996), and Schneier (1996).

Before we go into the details of the various protocols, Fig. 9-7 summarizes the notation and abbreviations we use in the mathematical expressions to follow.

Notation	Description
$K_{A,B}$	Secret key shared by A and B
K_A^+	Public key of A
K_A^-	Private key of A

Figure 9-7. Notation used in this chapter.

Symmetric Cryptosystems: DES

Our first example of a cryptographic algorithm is the **Data Encryption Standard** (DES), which is used for symmetric cryptosystems. DES is designed to operate on 64-bit blocks of data. A block is transformed into an encrypted (64 bit) block of output in 16 rounds, where each round uses a different 48-bit key for encryption. Each of these 16 keys is derived from a 56-bit master key, as shown in Fig. 9-8(a). Before an input block starts its 16 rounds of encryption, it is first subject to an initial permutation, of which the inverse is later applied to the encrypted output leading to the final output block.

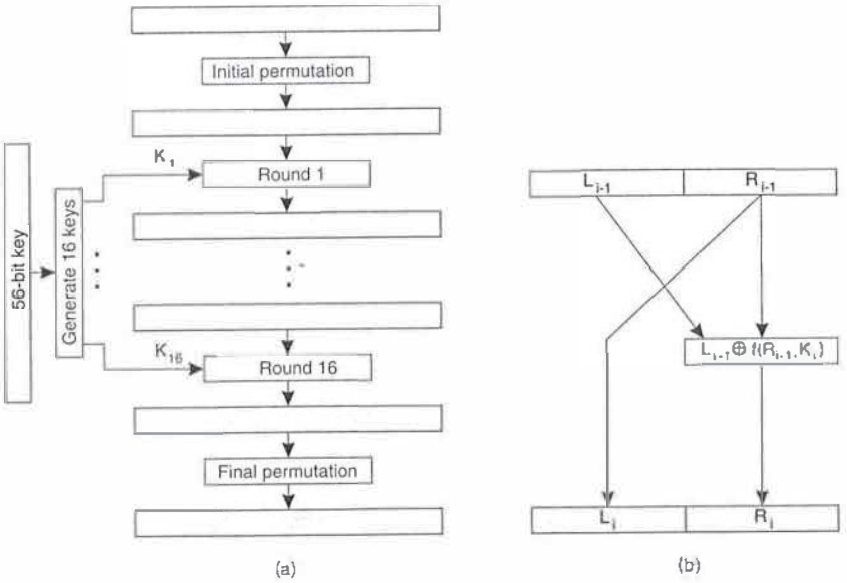

Figure 9-8. (a) The principle of DES. (b) Outline of one encryption round.

Each encryption round i takes the 64-bit block produced by the previous round $i-1$ as its input, as shown in Fig. 9-8(b). The 64 bits are split into a left part L_{i-1} and a right part R_{i-1}, each containing 32 bits. The right part is used for the left part in the next round, that is, $L_i = R_{i-1}$.

The hard work is done in the mangler function f. This function takes a 32-bit block R_{i-1} as input, together with a 48-bit key K_i, and produces a 32-bit block that is XORed with L_{i-1} to produce R_i. (XOR is an abbreviation for the exclusive or operation.) The mangler function first expands R_{i-1} to a 48-bit block and XORs it with K_i. The result is partitioned into eight chunks of six bits each. Each chunk is then fed into a different **S-box**, which is an operation that substitutes each of the 64 possible 6-bit inputs into one of 16 possible 4-bit outputs. The eight output chunks of four bits each are then combined into a 32-bit value and permuted again.

The 48-bit key K_i for round i is derived from the 56-bit master key as follows. First, the master key is permuted and divided into two 28-bit halves. For each round, each half is first rotated one or two bits to the left, after which 24 bits are extracted. Together with 24 bits from the other rotated half, a 48-bit key is constructed. The details of one encryption round are shown in Fig. 9-9.

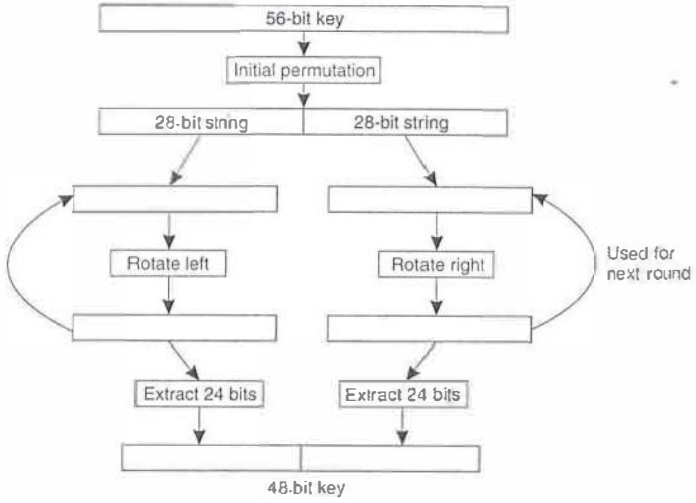

Figure 9-9. Details of per-round key generation in DES.

The principle of DES is quite simple, but the algorithm is difficult to break using analytical methods. Using a brute-force attack by simply searching for a key that will do the job has become easy as has been demonstrated a number of times. However, using DES three times in a special encrypt-decrypt-encrypt mode with

different keys, also known as Triple DES is much more safe and is still often used [see also Barker (2004)].

What makes DES difficult to attack by analysis is that the rationale behind the design has never been explained in public. For example, it is known that taking other S-boxes than are currently used in the standard, makes the algorithm substantially easier to break (see Pfleeger, 2003 for a brief analysis of DES). A rationale for the design and use of the S-boxes was published only after "new" attack models had been devised in the 1990s. DES proved to be quite resistant to these attacks, and its designers revealed that the newly devised models were already known to them when they developed DES in 1974 (Coppersmith, 1994).

DES has been used as a standard encryption technique for years, but is currently in the process of being replaced by the Rijndael algorithm blocks of 128 bits. There are also variants with larger keys and larger data blocks. The algorithm has been designed to be fast enough so that it can even be implemented on smart cards, which form an increasingly important application area for cryptography.

Public-Key Cryptosystems: RSA

Our second example of a cryptographic algorithm is very widely used for public-key systems: RSA, named after its inventors: Rivest, Shamir, and Adleman (1978). The security of RSA comes from the fact that no methods are known to efficiently find the prime factors of large numbers. It can be shown that each integer can be written as the product of prime numbers. For example, 2100 can be written as

$$2100 = 2 \times 2 \times 3 \times 5 \times 5 \times 7$$

making 2, 3, 5, and 7 the prime factors in 2100. In RSA, the private and public keys are constructed from very large prime numbers (consisting of hundreds of decimal digits). As it turns out, breaking RSA is equivalent to finding those two prime numbers. So far, this has shown to be computationally infeasible despite mathematicians working on the problem for centuries.

Generating the private and public keys requires four steps:

1. Choose two very large prime numbers, p and q.
2. Compute $n = p \times q$ and $z = (p - 1) \times (q - 1)$.
3. Choose a number d that is relatively prime to z.
4. Compute the number e such that $e \times d = 1 \ mod \ z$.

One of the numbers, say d, can subsequently be used for decryption, whereas e is used for encryption. Only one of these two is made public, depending on what the algorithm is being used for.

Let us consider the case that Alice wants to keep the messages she sends to Bob confidential. In other words, she wants to ensure that no one but Bob can intercept and read her messages to him. RSA considers each message m to be just a string of bits. Each message is first divided into fixed-length blocks, where each block m_i, interpreted as a binary number, should lie in the interval $0 \leq m_i < n$.

To encrypt message m, the sender calculates for each block m_i the value $c_i = m_i^e \ (mod \ n)$, which is then sent to the receiver. Decryption at the receiver's side takes place by computing $m_i = c_i^d \ (mod \ n)$. Note that for the encryption, both e and n are needed, whereas decryption requires knowing the values d and n.

When comparing RSA to symmetric cryptosystems such as DES, RSA has the drawback of being computationally more complex. As it turns out, encrypting messages using RSA is approximately 100–1000 times slower than DES, depending on the implementation technique used. As a consequence, many cryptographic systems use RSA to exchange only shared keys in a secure way, but much less for actually encrypting "normal" data. We will see examples of the combination of these two techniques later in succeeding sections.

Hash Functions: MD5

As a last example of a widely-used cryptographic algorithm, we take a look at MD5 (Rivest, 1992). MD5 is a hash function for computing a 128-bit, fixed length message digest from an arbitrary length binary input string. The input string is first padded to a total length of 448 bits (modulo 512), after which the length of the original bit string is added as a 64-bit integer. In effect, the input is converted to a series of 512-bit blocks.

The structure of the algorithm is shown in Fig. 9-10. Starting with some constant 128-bit value, the algorithm proceeds in k phases, where k is the number of 512-bit blocks comprising the padded message. During each phase, a 128-bit digest is computed out of a 512-bit block of data coming from the padded message, and the 128-bit digest computed in the preceding phase.

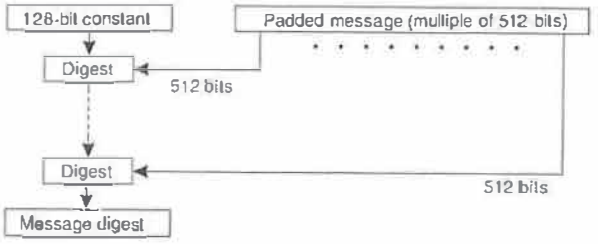

Figure 9-10. The structure of MD5.

A phase in MD5 consists of four rounds of computations, where each round uses one of the following four functions:

$$F(x,y,z) = (x\ AND\ y)\ OR\ ((NOT\ x)\ AND\ z)$$
$$G(x,y,z) = (x\ AND\ z)\ OR\ (y\ AND\ (NOT\ z))$$
$$H(x,y,z) = x\ XOR\ y\ XOR\ z$$
$$I(x,y,z) = y\ XOR\ (x\ OR\ (NOT\ z))$$

Each of these functions operates on 32-bit variables x, y, and z. To illustrate how these functions are used, consider a 512-bit block b from the padded message that is being processed during phase k. Block b is divided into 16 32-bit subblocks $b_0, b_1, \ldots, b_{15}$. During the first round, function F is used to change four variables (denoted as p, q, r, and s, respectively) in 16 iterations as shown in Fig. 9-11. These variables are carried to each next round, and after a phase has finished, passed on to the next phase. There are a total of 64 predefined constants C_i. The notation $x \lll n$ is used to denote a *left rotate*: the bits in x are shifted n positions to the left, where the bit shifted off the left is placed in the rightmost position.

Iterations 1–8	Iterations 9–16
$p \leftarrow (p + F(q,r,s) + b_0 + C_1) \lll 7$	$p \leftarrow (p + F(q,r,s) + b_8 + C_9) \lll 7$
$s \leftarrow (s + F(p,q,r) + b_1 + C_2) \lll 12$	$s \leftarrow (s + F(p,q,r) + b_9 + C_{10}) \lll 12$
$r \leftarrow (r + F(s,p,q) + b_2 + C_3) \lll 17$	$r \leftarrow (r + F(s,p,q) + b_{10} + C_{11}) \lll 17$
$q \leftarrow (q + F(r,s,p) + b_3 + C_4) \lll 22$	$q \leftarrow (q + F(r,s,p) + b_{11} + C_{12}) \lll 22$
$p \leftarrow (p + F(q,r,s) + b_4 + C_5) \lll 7$	$p \leftarrow (p + F(q,r,s) + b_{12} + C_{13}) \lll 7$
$s \leftarrow (s + F(p,q,r) + b_5 + C_6) \lll 12$	$s \leftarrow (s + F(p,q,r) + b_{13} + C_{14}) \lll 12$
$r \leftarrow (r + F(s,p,q) + b_6 + C_7) \lll 17$	$r \leftarrow (r + F(s,p,q) + b_{14} + C_{15}) \lll 17$
$q \leftarrow (q + F(r,s,p) + b_7 + C_8) \lll 22$	$q \leftarrow (q + F(r,s,p) + b_{15} + C_{16}) \lll 22$

Figure 9-11. The 16 iterations during the first round in a phase in MD5.

The second round uses the function G in a similar fashion, whereas H and I are used in the third and fourth round, respectively. Each step thus consists of 64 iterations, after which the next phase is started, but now with the values that p, q, r, and s have at that point.

9.2 SECURE CHANNELS

In the preceding chapters, we have frequently used the client-server model as a convenient way to organize a distributed system. In this model, servers may possibly be distributed and replicated, but also act as clients with respect to other servers. When considering security in distributed systems, it is once again useful to think in terms of clients and servers. In particular, making a distributed system secure essentially boils down to two predominant issues. The first issue is how to make the communication between clients and servers secure. Secure communication requires authentication of the communicating parties. In many cases it also requires ensuring message integrity and possibly confidentiality as well. As part of this problem, we also need to consider protecting the communication within a group of servers.

The second issue is that of authorization: once a server has accepted a request from a client, how can it find out whether that client is authorized to have that request carried out? Authorization is related to the problem of controlling access to resources, which we discuss extensively in the next section. In this section, we concentrate on protecting the communication within a distributed system.

The issue of protecting communication between clients and servers, can be thought of in terms of setting up a **secure channel** between communicating parties (Voydock and Kent, 1983). A secure channel protects senders and receivers against interception, modification, and fabrication of messages. It does not also necessarily protect against interruption. Protecting messages against interception is done by ensuring confidentiality: the secure channel ensures that its messages cannot be eavesdropped by intruders. Protecting against modification and fabrication by intruders is done through protocols for mutual authentication and message integrity. In the following pages, we first discuss various protocols that can be used for authentication, using symmetric as well as public-key cryptosystems. A detailed description of the logics underlying authentication can be found in Lampson et al. (1992). We discuss confidemiality and message integrity separately.

9.2.1 Authentication

Before going into the details of various authentication protocols, it is worthwhile noting that authentication and message integrity cannot do without each other. Consider, for example, a distributed system that supports authentication of two communicating parties, but does not provide mechanisms to ensure message integrity. In such a system, Bob may know for sure that Alice is the sender of a message m. However, if Bob cannot be given guarantees that m has not been modified during transmission, what use is it to him to know that Alice sem (the original version of) m?

Likewise, suppose that only message integrity is supported, but no mechanisms exist for authentication. When Bob receives a message stating that he has just won $1,000,000 in the lottery, how happy can he be if he cannot verify that the message was sent by the organizers of that lottery?

Consequently, authentication and message integrity should go together. In many protocols, the combination works roughly as follows. Again, assume that Alice and Bob want to communicate, and that Alice takes the initiative in setting up a channel. Alice starts by sending a message to Bob, or otherwise to a trusted third party who will help set up the channel. Once the channel has been set up,

Alice knows for sure that she is talking to Bob, and Bob knows for sure he is talking to Alice, they can exchange messages.

To subsequently ensure integrity of the data messages that are exchanged after authentication has taken place, it is common practice to use secret-key cryptography by means of session keys. A session key is a shared (secret) key that is used to encrypt messages for integrity and possibly also confidentiality. Such a key is generally used only for as long as the channel exists. When the channel is closed, its associated session key is discarded (or actually, securely destroyed). We return to session keys below.

Authentication Based on a Shared Secret Key

Let us start by taking a look at an authentication protocol based on a secret key that is already shared between Alice and Bob. How the two actually managed to obtain a shared key in a secure way is discussed later in this chapter. In the description of the protocol, Alice and Bob are abbreviated by A and B, respectively, and their shared key is denoted as $K_{A,B}$. The protocol takes a common approach whereby one party challenges the other to a response that can be correct only if the other knows the shared secret key. Such solutions are also known as **challenge-response protocols**.

In the case of authentication based on a shared secret key, the protocol proceeds as shown in Fig. 9-12. First, Alice sends her identity to Bob (message 1), indicating that she wants to set up a communication channel between the two. Bob subsequently sends a challenge R_B to Alice, shown as message 2. Such a challenge could take the form of a random number. Alice is required to encrypt the challenge with the secret key $K_{A,B}$ that she shares with Bob, and return the encrypted challenge to Bob. This response is shown as message 3 in Fig. 9-12 containing $K_{A,B}(R_B)$.

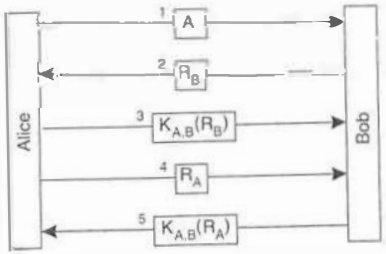

Figure 9-12. Authentication based on a shared secret key.

When Bob receives the response $K_{A,B}(R_B)$ to his challenge R_B, he can decrypt the message using the shared key again to see if it contains R_B. If so, he then knows that Alice is on the other side, for who else could have encrypted R_B with

$K_{A,B}$ in the first place? In other words, Bob has now verified that he is indeed talking to Alice. However, note that Alice has not yet verified that it is indeed Bob on the other side of the channel. Therefore, she sends a challenge R_A (message 4), which Bob responds to by returning $K_{A,B}(R_A)$, shown as message 5. When Alice decrypts it with $K_{A,B}$ and sees her R_A, she knows she is talking to Bob.

One of the harder issues in security is designing protocols that actually work. To illustrate how easily things can go wrong, consider an "optimization" of the authentication protocol in which the number of messages has been reduced from five to three, as shown in Fig. 9-13. The basic idea is that if Alice eventually wants to challenge Bob anyway, she might as well send a challenge along with her identity when setting up the channel. Likewise, Bob returns his response to that challenge, along with his own challenge in a single message.

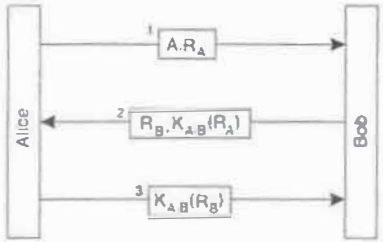

Figure 9-13. Authentication based on a shared secret key, but using three instead of five messages.

Unfortunately, this protocol no longer works. It can easily be defeated by what is known as a **reflection attack**. To explain how such an attack works, consider an intruder called Chuck, whom we denote as C in our protocols. Chuck's goal is to set up a channel with Bob so that Bob believes he is talking to Alice. Chuck can establish this if he responds correctly to a challenge sent by Bob, for instance, by returning the encrypted version of a number that Bob sent. Without knowledge of $K_{A,B}$, only Bob can do such an encryption, and this is precisely what Chuck tricks Bob into doing.

The attack is illustrated in Fig. 9-14. Chuck starts out by sending a message containing Alice's identity A, along with a challenge R_C. Bob returns his challenge R_B and the response $K_{A,B}(R_C)$ in a single message. At that point, Chuck would need to prove he knows the secret key by returning $K_{A,B}(R_B)$ to Bob. Unfortunately, he does not have $K_{A,B}$. Instead, what he does is attempt to set up a second channel to let Bob do the encryption for him.

Therefore, Chuck sends A and R_B in a single message as before, but now pretends that he wants a second channel. This is shown as message 3 in Fig. 9-14. Bob, not recognizing that he, himself, had used R_B before as a challenge, responds with $K_{A,B}(R_B)$ and another challenge R_{B2}, shown as message 4. At that point,

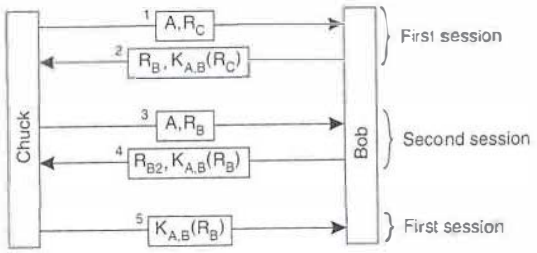

Figure 9-14. The reflection attack.

Chuck has $K_{A,B}(R_B)$ and finishes setting up the first session by returning message 5 containing the response $K_{A,B}(R_B)$, which was originally requested from the challenge sent in message 2.

As explained in Kaufman et al. (2003), one of the mistakes made during the adaptation of the original protocol was that the two parties in the new version of the protocol were using the same challenge in two different runs of the protocol. A better design is to always use different challenges for the initiator and for the responder. For example, if Alice always uses an odd number and Bob an even number, Bob would have recognized that something fishy was going on when receiving R_B in message 3 in Fig. 9-14. (Unfortunately, this solution is subject to other attacks, notably the one known as the "man-in-the-middle attack," which is explained in Ferguson and Schneier, 2003). In general, letting the two parties setting up a secure channel do a number of things identically is not a good idea.

Another principle that is violated in the adapted protocol is that Bob gave away valuable information in the form of the response $K_{A,B}(R_C)$ without knowing for sure to whom he was giving it. This principle was not violated in the original protocol, in which Alice first needed to prove her identity, after which Bob was willing to pass her encrypted information.

There are other principles that developers of cryptographic protocols have gradually come to learn over the years, and we will present some of them when discussing other protocols below. One important lesson is that designing security protocols that do what they are supposed to do is often much harder than it looks. Also, tweaking an existing protocol to improve its performance, can easily affect its correctness as we demonstrated above. More on design principles for protocols can be found in Abadi and Needham (1996).

Authentication Using a Key Distribution Center

One of the problems with using a shared secret key for authentication is scalability. If a distributed system contains N hosts, and each host is required to share a secret key with each of the other $N-1$ hosts, the system as a whole needs to manage $N(N-1)/2$ keys, and each host has to manage $N-1$ keys. For large N, this will lead to problems. An alternative is to use a centralized approach by means of a **Key Distribution Center (KDC)**. This KDC shares a secret key with each of the hosts, but no pair of hosts is required to have a shared secret key as well. In other words, using a KDC requires that we manage N keys instead of $N(N-1)/2$, which is clearly an improvement.

If Alice wants to set up a secure channel with Bob, she can do so with the help of a (trusted) KDC. The whole idea is that the KDC hands out a key to both Alice and Bob that they can use for communication, shown in Fig. 9-15.

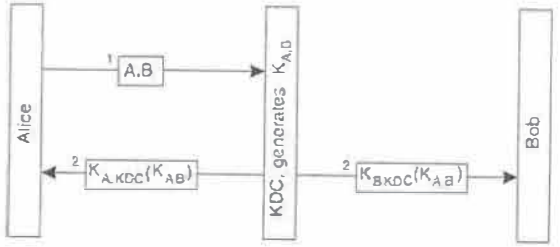

Figure 9-15. The principle of using a KDC.

Alice first sends a message to the KDC, telling it that she wants to talk to Bob. The KDC returns a message containing a shared secret key $K_{A,B}$ that she can use. The message is encrypted with the secret key $K_{A,KDC}$ that Alice shares with the KDC. In addition, the KDC sends $K_{A,B}$ also to Bob, but now encrypted with the secret key $K_{B,KDC}$ it shares with Bob.

The main drawback of this approach is that Alice may want to start setting up a secure channel with Bob even before Bob had received the shared key from the KDC. In addition, the KDC is required to get Bob into the loop by passing him the key. These problems can be circumvented if the KDC just passes $K_{B,KDC}(K_{A,B})$ back to Alice, and lets her take care of connecting to Bob. This leads to the protocol shown in Fig. 9-16. The message $K_{B,KDC}(K_{A,B})$ is also known as a **ticket**. It is Alice's job to pass this ticket to Bob. Note that Bob is still the only one that can make sensible use of the ticket, as he is the only one besides the KDC who knows how to decrypt the information it contains.

The protocol shown in Fig. 9-16 is actually a variant of a well-known example of an authentication protocol using a KDC, known as the **Needham-Schroeder authentication protocol**, named after its inventors (Needham and Schroeder, 1978). A different variant of the protocol is being used in the Kerberos system, which we describe later. The Needham-Schroeder protocol, shown in Fig. 9-17, is a multiway challenge-response protocol and works as follows.

When Alice wants to set up a secure channel with Bob, she sends a request to the KDC containing a challenge R_1, along with her identity A and, of course, that

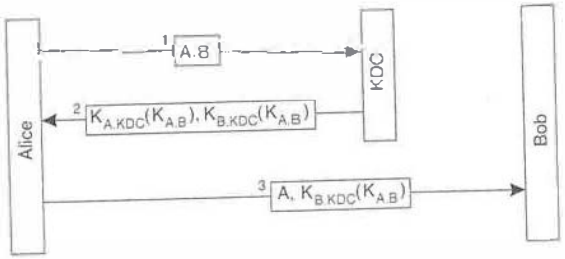

Figure 9-16. Using a ticket and letting Alice set up a connection to Bob.

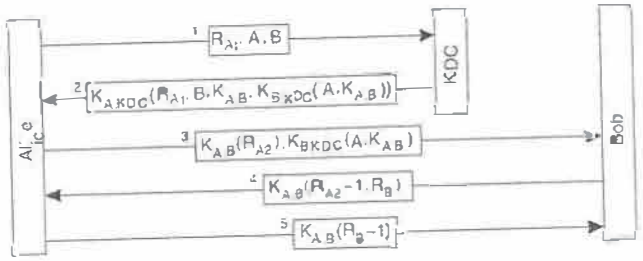

Figure 9-17. The Needham-Schroeder authentication protocol.

of Bob. The KDC responds by giving her the ticket $K_{B,KDC}(K_{A,B})$, along with the secret key $K_{A,B}$ that she can subsequently share with Bob.

The challenge R_{A1} that Alice sends to the KDC along with her request to set up a channel to Bob is also known as a nonce. A nonce is a random number that is used only once, such as one chosen from a very large set. The main purpose of a nonce is to uniquely relate two messages to each other, in this case message 1 and message 2. In particular, by including R_{A1} again in message 2, Alice will know for sure that message 2 is sent as a response to message 1, and that it is not, for example, a replay of an older message.

To understand the problem at hand, assume that we did not use nonces, and that Chuck has stolen one of Bob's old keys, say $K^{old}_{B,KDC}$. In addition, Chuck has intercepted an old response $K_{A,KDC}(B, K_{A,B}, K^{old}_{B,KDC}(A, K_{A,B}))$ that the KDC had returned to a previous request from Alice to talk to Bob. Meanwhile, Bob will have negotiated a new shared secret key with the KDC. However, Chuck patiently waits until Alice again requests to set up a secure channel with Bob. At that point, he replays the old response, and fools Alice into making her believe she is talking

to Bob, because he can decrypt the ticket and prove he knows the shared secret key $K_{A,B}$. Clearly this is unacceptable and must be defended against.

By including a nonce, such an attack is impossible, because replaying an older message will immediately be discovered. In particular, the nonce in the response message will not match the nonce in the original request.

Message 2 also contains B, the identity of Bob. By including B, the KDC protects Alice against the following attack. Suppose that B was left out of message 2. In that case, Chuck could modify message 1 by replacing the identity of Bob with his own identity, say C. The KDC would then think Alice wants to set up a secure channel to Chuck, and responds accordingly. As soon as Alice wants to contact Bob, Chuck intercepts the message and fools Alice into believing she is talking to Bob. By copying the identity of the other party from message 1 to message 2, Alice will immediately detect that her request had been modified.

After the KDC has passed the ticket to Alice, the secure channel between Alice and Bob can be set up. Alice starts with sending message 3, which contains the ticket to Bob, and a challenge R_{A2} encrypted with the shared key $K_{A,B}$ that the KDC had just generated. Bob then decrypts the ticket to find the shared key, and returns a response $R_{A2} - 1$ along with a challenge R_B for Alice.

The following remark regarding message 4 is in order. In general, by returning $R_{A2} - 1$ and not just R_{A2}, Bob not only proves he knows the shared secret key, but also that he has actually decrypted the challenge. Again, this ties message 4 to message 3 in the same way that the nonce R_A tied message 2 to message 1. The protocol is thus more protected against replays.

However, in this special case, it would have been sufficient to just return $K_{A,B}(R_{A2}, R_B)$, for the simple reason that this message has not yet been used anywhere in the protocol before. $K_{A,B}(R_{A2}, R_B)$ already proves that Bob has been capable of decrypting the challenge sent in message 3. Message 4 as shown in Fig. 9-17 is due to historical reasons.

The Needham-Schroeder protocol as presented here still has the weak point that if Chuck ever got a hold of an old key $K_{A,B}$, he could replay message 3 and get Bob to set up a channel. Bob will then believe he is talking to Alice, while, in fact, Chuck is at the other end. In this case, we need to relate message 3 to message 1, that is, make the key dependent on the initial request from Alice to set up a channel with Bob. The solution is shown in Fig. 9-18.

The trick is to incorporate a nonce in the request sent by Alice to the KDC. However, the nonce has to come from Bob: this assures Bob that whoever wants to set up a secure channel with him, will have gotten the appropriate information from the KDC. Therefore, Alice first requests Bob to send her a nonce R_{B1}, encrypted with the key shared between Bob and the KDC. Alice incorporates this nonce in her request to the KDC, which will then subsequently decrypt it and put the result in the generated ticket. In this way, Bob will know for sure that the session key is tied to the original request from Alice to talk to Bob.

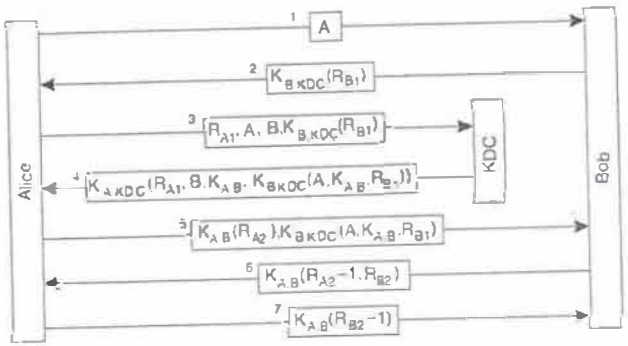

Figure 9-18. Protection against malicious reuse of a previously generated session key in the Needham-Schroeder protocol.

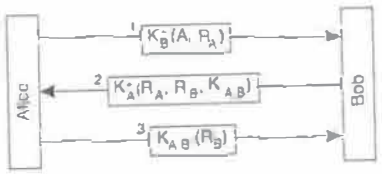

Figure 9-19. Mutual authentication in a public-key cryptosystem.

Authentication Using Public-Key Cryptography

Let us now look at authentication with a public-key cryptosystem that does not require a KDC. Again, consider the situation that Alice wants to set up a secure channel to Bob, and that both are in the possession of each other's public key. A typical authentication protocol based on public-key cryptography is shown in Fig. 9-19, which we explain next.

Alice starts with sending a challenge R_A to Bob encrypted with his public key K_B^+. It is Bob's job to decrypt the message and return the challenge to Alice. Because Bob is the only person that can decrypt the message (using the private key that is associated with the public key Alice used), Alice will know that she is talking to Bob. Note that it is important that Alice is guaranteed to be using Bob's public key, and not the public key of someone impersonating Bob. How such guarantees can be given is discussed later in this chapter.

When Bob receives Alice's request to set up a channel, he returns the decrypted challenge, along with his own challenge R_B to authenticate Alice. In addition, he generates a session key $K_{A,B}$ that can be used for further communication. Bob's response to Alice's challenge, his own challenge, and the session key

are put into a message encrypted with the public key K_A^+ belonging to Alice, shown as message 2 in Fig. 9-19. Only Alice will be capable of decrypting this message using the private key K_A^- associated with K_A^+.

Alice, finally, returns her response to Bob's challenge using the session key $K_{A,B}$ generated by Bob. In that way, she will have proven that she could decrypt message 2, and thus that she is actually Alice to whom Bob is talking.

9.2.2 Message Integrity and Confidentiality

Besides authentication, a secure channel should also provide guarantees for message integrity and confidentiality. Message integrity means that messages are protected against surreptitious modification; confidentiality ensures that messages cannot be intercepted and read by eavesdroppers. Confidentiality is easily established by simply encrypting a message before sending it. Encryption can take place either through a secret key shared with the receiver or alternatively by using the receiver's public key. However, protecting a message against modifications is somewhat more complicated, as we discuss next.

Digital Signatures

Message integrity often goes beyond the actual transfer through a secure channel. Consider the situation in which Bob has just sold Alice a collector's item of some phonograph record for $500. The whole deal was done through e-mail. In the end, Alice sends Bob a message confirming that she will buy the record for $500. In addition to authentication, there are at least two issues that need to be taken care of regarding the integrity of the message.

1. Alice needs to be assured that Bob will not maliciously change the $500 mentioned in her message into something higher, and claim she promised more than $500.
2. Bob needs to be assured that Alice cannot deny ever having sent the message, for example, because she had second thoughts.

These two issues can be dealt with if Alice digitally signs the message in such a way that her signature is uniquely tied to its content. The unique association between a message and its signature prevents that modifications to the message will go unnoticed. In addition, if Alice's signature can be verified to be genuine, she cannot later repudiate the fact that she signed the message.

There are several ways to place digital signatures. One popular form is to use a public-key cryptosystem such as RSA, as shown in Fig. 9-20. When Alice sends a message m to Bob, she encrypts it with her *private* key K_A^-, and sends it

off to Bob. If she also wants to keep the message content a secret, she can use Bob's public key and send $K_B^+(m, K_A^-(m))$, which combines m and the version signed by Alice.

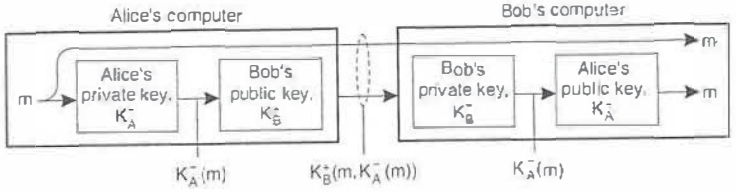

Figure 9-20. Digital signing a message using public-key cryptography.

When the message arrives at Bob, he can decrypt it using Alice's public key. If he can be assured that the public key is indeed owned by Alice, then decrypting the signed version of m and successfully comparing it to m can mean only that it came from Alice. Alice is protected against any malicious modifications to m by Bob, because Bob will always have to prove that the modified version of m was also signed by Alice. In other words, the decrypted message alone essentially never counts as proof. It is also in Bob's own interest to keep the signed version of m to protect himself against repudiation by Alice.

There are a number of problems with this scheme, although the protocol in itself is correct. First, the validity of Alice's signature holds only as long as Alice's private key remains a secret. If Alice wants to bail out of the deal even after sending Bob her confirmation, she could claim that her private key was stolen before the message was sent.

Another problem occurs when Alice decides to change her private key. Doing so may in itself be not such a bad idea, as changing keys from time to time generally helps against intrusion. However, once Alice has changed her key, her statement sent to Bob becomes worthless. What may be needed in such cases is a central authority that keeps track of when keys are changed, in addition to using timestamps when signing messages.

Another problem with this scheme is that Alice encrypts the entire message with her private key. Such an encryption may be costly in terms of processing requirements (or even mathematically infeasible as we assume that the message interpreted as a binary number is bounded by a predefined maximum), and is actually unnecessary. Recall that we need to uniquely associate a signature with a only specific message. A cheaper and arguably more elegant scheme is to use a message digest.

As we explained, a message digest is a fixed-length bit string h that has been computed from an arbitrary-length message m by means of a cryptographic hash function H. If m is changed to m', its hash $H(m')$ will be different from $h = H(m)$ so that it can easily be detected that a modification has taken place.

To digitally sign a message, Alice can first compute a message digest and subsequently encrypt the digest with her private key, as shown in Fig. 9-21. The encrypted digest is sent along with the message to Bob. Note that the message itself is sent as plaintext: everyone is allowed to read it. If confidentiality is required, then the message should also be encrypted with Bob's public key.

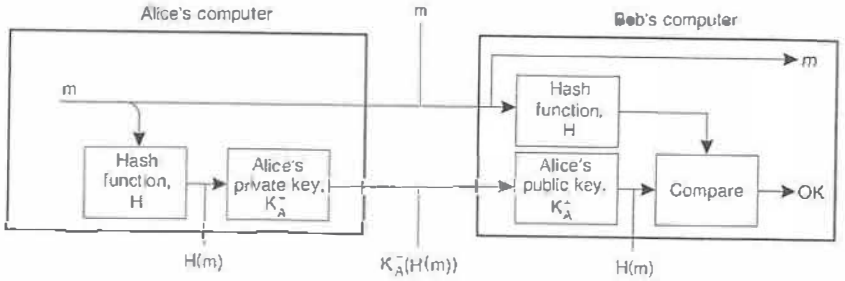

Figure 9-21. Digitally signing a message using a message digest.

When Bob receives the message and its encrypted digest, he need merely decrypt the digest with Alice's public key, and separately calculate the message digest. If the digest calculated from the received message and the decrypted digest match, Bob knows the message has been signed by Alice.

Session Keys

During the establishment of a secure channel, after the authentication phase has completed, the communicating parties generally use a unique shared session key for confidentiality. The session key is safely discarded when the channel is no longer used. An alternative would have been to use the same keys for confidentiality as those that are used for setting up the secure channel. However, there are a number of important benefits to using session keys (Kaufman et al., 2003).

First, when a key is used often, it becomes easier to reveal it. In a sense, cryptographic keys are subject to "wear and tear" just like ordinary keys. The basic idea is that if an intruder can intercept a lot of data that have been encrypted using the same key, it becomes possible to mount attacks to find certain characteristics of the keys used, and possibly reveal the plaintext or the key itself. For this reason, it is much safer to use the authentication keys as little as possible. In addition, such keys are often exchanged using some relatively time-expensive out-of-band mechanism such as regular mail or telephone. Exchanging keys that way should be kept to a minimum.

Another important reason for generating a unique key for each secure channel is to ensure protection against replay attacks as we have come across previously a

number of times. By using a unique session key each time a secure channel is set up, the communicating parties are at least protected against replaying an entire session. To protect replaying individual messages from a previous session, additional measures are generally needed such as including timestamps or sequence numbers as part of the message content.

Suppose that message integrity and confidentiality were achieved by using the same key used for session establishment. In that case, whenever the key is compromised, an intruder may be able to decrypt messages transferred during an old conversation, clearly not a desirable feature. Instead, it is much safer to use per-session keys, because if such a key is compromised, at worst, only a single session is affected. Messages sent during other sessions stay confidential.

Related to this last point is that Alice may want to exchange some confidential data with Bob, but she does not trust him so much that she would give him information in the form of data that have been encrypted with long-lasting keys. She may want to reserve such keys for highly-confidential messages that she exchanges with parties she really trusts. In such cases, using a relatively cheap session key to talk to Bob is sufficient.

By and large, authentication keys are often established in such a way that replacing them is relatively expensive. Therefore, the combination of such long-lasting keys with the much cheaper and more temporary session keys is often a good choice for implementing secure channels for exchanging data.

9.2.3 Secure Group Communication

So far, we have concentrated on setting up a secure communication channel between two parties. In distributed systems, however, it is often necessary to enable secure communication between more than just two parties. A typical example is that of a replicated server for which all communication between the replicas should be protected against modification, fabrication, and interception, just as in the case of two-party secure channels. In this section, we take a closer look at secure group communication.

Confidential Group Communication

First, consider the problem of protecting communication between a group of N users against eavesdropping. To ensure confidentiality, a simple scheme is to let all group members share the same secret key, which is used to encrypt and decrypt all messages transmitted between group members. Because the secret key in this scheme is shared by all members, it is necessary that all members are trusted to indeed keep the key a secret. This prerequisite alone makes the use of a single shared secret key for confidential group communication more vulnerable to attacks compared to two-party secure channels.

An alternative solution is to use a separate shared secret key between each pair of group members. As soon as one member turns out to be leaking information, the others can simply stop sending messages to that member, but still use the keys they were using to communicate with each other. However, instead of having to maintain one key, it is now necessary to maintain $N(N-1)/2$ keys, which may be a difficult problem by itself.

Using a public-key cryptosystem can improve matters. In that case, each member has its own *(public key, private key)* pair, in which the public key can be used by all members for sending confidential messages. In this case, a total of N key pairs are needed. If one member ceases to be trustworthy, it is simply removed from the group without having been able to compromise the other keys.

Secure Replicated Servers

Now consider a completely different problem: a client issues a request to a group of replicated servers. The servers may have been replicated for reasons of fault tolerance or performance, but in any case, the client expects the response to be trustworthy. In other words, regardless of whether the group of servers is subject to Byzantine failures as we discussed in the previous chapter, a client expects that the returned response has not been subject to a security attack. Such an attack could happen if one or more servers had been successfully corrupted by an intruder.

A solution to protect the client against such attacks is to collect the responses from all servers and authenticate each one of them. If a majority exists among the responses from the noncorrupted (i.e., authenticated) servers, the client can trust the response to be correct as well. Unfortunately, this approach reveals the replication of the servers, thus violating replication transparency.

Reiter et al. (1994) proposes a solution to a secure, replicated server in which replication transparency is maintained. The advantage of their scheme is that because clients are unaware of the actual replicas, it becomes much easier to add or remove replicas in a secure way. We return to managing secure groups below when discussing key management.

The essence of secure and transparent replicated servers lies in what is known as secret sharing. When multiple users (or processes) share a secret, none of them knows the entire secret. Instead, the secret can be revealed only if they all get together. Such schemes can be extremely useful. Consider, for example, launching a nuclear missile. Such an act generally requires the authorization of at least two people. Each of them holds a private key that should be used in combination with the other to actually launch a missile. Using only a single key will not do.

In the case of secure, replicated servers, what we are seeking is a solution by which at most k out of the N servers can produce an incorrect answer, and of those k servers, at most $c \leq k$ have actually been corrupted by an intruder. Note that this

requirement makes the service itself k fault tolerant as discussed in the previous chapter. The difference lies in the fact that we now classify a maliciously corrupted server as being faulty.

Now consider the situation in which the servers are actively replicated. In other words, a request is sent to all servers simultaneously, and subsequently handled by each of them. Each server produces a response that it returns to the client. For a securely replicated group of servers, we require that each server accompanies its response with a digital signature. If r_i is the response from server S_i, let $md(r_i)$ denote the message digest computed by server S_i. This digest is signed with server S_i's private key K_i^-.

Suppose that we want to protect the client against at most c corrupted servers. In other words, the server group should be able to tolerate corruption by at most c servers, and still be capable of producing a response that the client can put its trust in. If the signatures of the individual servers could be combined in such a way that at least $c + 1$ signatures are needed to construct a *valid* signature for the response, then this would solve our problem. In other words, we want to let the replicated servers generate a secret valid signature with the property that c corrupted servers alone are not enough to produce that signature.

As an example, consider a group of five replicated servers that should be able to tolerate two corrupted servers, and still produce a response that a client can trust. Each server S_i sends its response r_i to the client, along with its signature $sig(S_i,r_i) = K_i^-(md(r_i))$. Consequently, the client will eventually have received five triplets $<r_i, md(r_i), sig(S_i,r_i)>$ from which it should derive the correct response. This situation is shown in Fig. 9-22.

Each digest $md(r_i)$ is also calculated by the client. If r_i is incorrect, then normally this can be detected by computing $K_i^+(K_i^-(md(r_i)))$. However, this method can no longer be applied, because no individual server can be trusted. Instead, the client uses a special, publicly-known decryption function D, which takes a set $V = \{sig(S,r), sig(S',r'), sig(S'',r'')\}$ of *three* signatures as input, and produces a single digest as output:

$$d_{out} = D(V) = D(sig(S,r), sig(S',r'), sig(S'',r''))$$

For details on D, see Reiter (1994). There are 5!/(3!2!)=10 possible combinations of three signatures that the client can use as input for D. If one of these combinations produces a correct digest $md(r_i)$ for some response r_i, then the client can consider r_i as being correct. In particular, it can trust that the response has been produced by at least three honest servers.

To improve replication transparency, Reiter and Birman let each server S_i broadcast a message containing its response r_i to the other servers, along with the associated signature $sig(S_i,r_i)$. When a server has received at least $c + 1$ of such messages, including its own message, it attempts to compute a valid signature for one of the responses. If this succeeds for, say, response r and the set V of $c + 1$ signatures, the server sends r and V as a single message to the client. The client

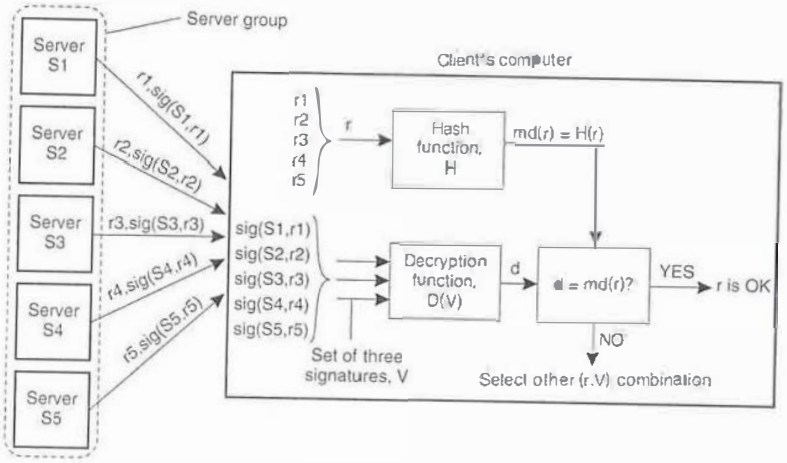

Figure 9-22. Sharing a secret signature in a group of replicated servers.

can subsequently verify the correctness of r by checking its signature, that is, whether $md(r) = D(V)$.

What we have just described is also known as an (m,n)-threshold scheme with, in our example, $m = c + 1$ and $n = N$, the number of servers. In an (m,n)-threshold scheme, a message has been divided into n pieces, known as shadows, since any m shadows can be used to reconstruct the original message, but using $m - 1$ or fewer messages cannot. There are several ways to construct (m,n)-threshold schemes. Details can be found in Schneier (1996).

9.2.4 Example: Kerberos

It should be clear by now that incorporating security into distributed systems is not trivial. Problems are caused by the fact that the entire system must be secure; if some part is insecure, the whole system may be compromised. To assist the construction of distributed systems that can enforce a myriad of security policies, a number of supporting systems have been developed that can be used as a basis for further development. An important system that is widely used is **Kerberos** (Steiner et al., 1988; and Kohl and Neuman, 1994).

Kerberos was developed at M.I.T. and is based on the Needham-Schroeder authentication protocol we described earlier. There are currently two different versions of Kerberos in use, version 4 (V4) and version 5 (V5). Both versions are conceptually similar, with V5 being much more flexible and scalable. A detailed

description of V5 can be found in Neuman et al. (2005), whereas practical information on running Kerberos is described by Garman (2003).

Kerberos can be viewed as a security system that assists clients in setting up a secure channel with any server that is part of a distributed system. Security is based on shared secret keys. There are two different components. The **Authentication Server (AS)** is responsible for handling a login request from a user. The AS authenticates a user and provides a key that can be used to set up secure channels with servers. Setting up secure channels is handled by a **Ticket Granting Service (TGS)**. The TGS hands out special messages, known as tickets, that are used to convince a server that the client is really who he or she claims to be. We give concrete examples of tickets below.

Let us take a look at how Alice logs onto a distributed system that uses Kerberos and how she can set up a secure channel with server Bob. For Alice to log onto the system, she can use any workstation available. The workstation sends her name in plaintext to the AS, which returns a session key $K_{A,TGS}$ and a ticket that she will need to hand over to the TGS.

The ticket that is returned by the AS contains the identity of Alice, along with a generated secret key that Alice and the TGS can use to communicate with each other. The ticket itself will be handed over to the TGS by Alice. Therefore, it is important that no one but the TGS can read it. For this reason, the ticket is encrypted with the secret key $K_{AS,TGS}$ shared between the AS and the TGS.

This part of the login procedure is shown as messages 1, 2, and 3 in Fig. 9-23. Message 1 is not really a message, but corresponds to Alice typing in her login name at a workstation. Message 2 contains that name and is sent to the AS. Message 3 contains the session key $K_{A,TGS}$ and the ticket $K_{AS,TGS}(A, K_{A,TGS})$. To ensure privacy, message 3 is encrypted with the secret key $K_{A,AS}$ shared between Alice and the AS.

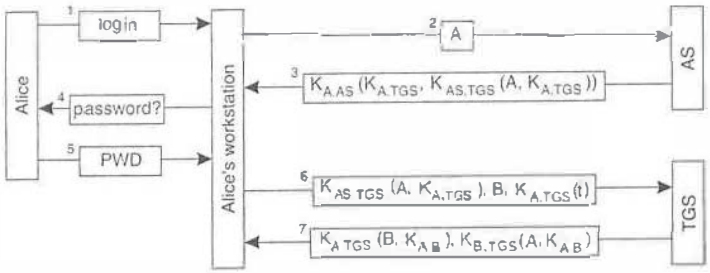

Figure 9-23. Authentication in Kerberos.

When the workstation receives the response from the AS, it prompts Alice for her password (shown as message 4), which it uses to subsequently generate the shared key $K_{A,AS}$. (It is relatively simple to take a character string password, apply a cryptographic hash, and then take the first 56 bits as the secret key.) Note that this approach not only has the advantage that Alice's password is never sent as plaintext across the network, but also that the workstation does not even have to temporarily store it. Moreover, as soon as it has generated the shared key $K_{A,AS}$, the workstation will find the session key $K_{A,TGS}$, and can forget about Alice's password and use only the shared secret $K_{A,AS}$.

After this part of the authentication has taken place, Alice can consider herself logged into the system through the current workstation. The ticket received from the AS is stored temporarily (typically for 8–24 hours), and will be used for accessing remote services. Of course, if Alice leaves her workstation, she should destroy any cached tickets. If she wants to talk to Bob, she requests the TGS to generate a session key for Bob, shown as message 6 in Fig. 9-23. The fact that Alice has the ticket $K_{AS,TGS}(A, K_{A,TGS})$ proves that she is Alice. The TGS responds with a session key $K_{A,B}$, again encapsulated in a ticket that Alice will later have to pass to Bob.

Message 6 also contains a timestamp, t, encrypted with the secret key shared between Alice and the TGS. This timestamp is used to prevent Chuck from maliciously replaying message 6 again, and trying to set up a channel to Bob. The TGS will verify the timestamp before returning a ticket to Alice. If it differs more than a few minutes from the current time, the request for a ticket is rejected.

This scheme establishes what is known as **single sign-on**. As long as Alice does not change workstations, there is no need for her to authenticate herself to any other server that is part of the distributed system. This feature is important when having to deal with many different services that are spread across multiple machines. In principle, servers in a way have delegated client authentication to the AS and TGS, and will accept requests from any client that has a valid ticket. Of course, services such as remote login will require that the associated user has an account, but this is independent from authentication through Kerberos.

Setting up a secure channel with Bob is now straightforward, and is shown in Fig. 9-24. First, Alice sends to Bob a message containing the ticket she got from the TGS, along with an encrypted timestamp. When Bob decrypts the ticket, he notices that Alice is talking to him, because only the TGS could have constructed the ticket. He also finds the secret key $K_{A,B}$, allowing him to verify the timestamp. At that point, Bob knows he is talking to Alice and not someone maliciously replaying message 1. By responding with $K_{A,B}(t + 1)$, Bob proves to Alice that he is indeed Bob.

9.3 ACCESS CONTROL

In the client-server model, which we have used so far, once a client and a server have set up a secure channel, the client can issue requests that are to be carried out by the server. Requests involve carrying out operations on resources that

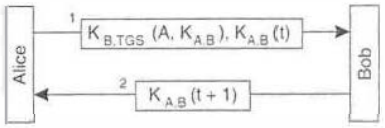

Figure 9-24. Setting up a secure channel in Kerberos.

are controlled by the server. A general situation is that of an object server that has a number of objects under its control. A request from a client generally involves invoking a method of a specific object. Such a request can be carried out only if the client has sufficient access rights for that invocation.

Formally, verifying access rights is referred to as access control, whereas authorization is about granting access rights. The two terms are strongly related to each other and are often used in an interchangeable way. There are many ways to achieve access control. We start with discussing some of the general issues, concentrating on different models for handling access control. One important way of actually controlling access to resources is to build a firewall that protects applications or even an entire network. Firewalls are discussed separately. With the advent of code mobility, access control could no longer be done using only the traditional methods. Instead, new techniques had to be devised, which are also discussed in this section.

9.3.1 General Issues in Access Control

In order to understand the various issues involved in access control, the simple model shown in Fig. 9-25 is generally adopted. It consists of subjects that issue a request to access an object. An object is very much like the objects we have been discussing so far. It can be thought of as encapsulating its own state and implementing the operations on that state. The operations of an object that subjects can request to be carried out are made available through interfaces. Subjects can best be thought of as being processes acting on behalf of users, but can also be objects that need the services of other objects in order to carry out their work.

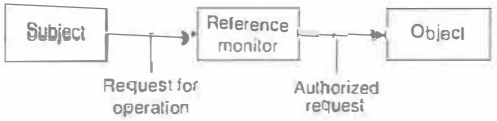

Figure 9-25. General model of controlling access to objects.

Controlling the access to an object is all about protecting the object against invocations by subjects that are not allowed to have specific (or even any) of the

methods carried out. Also, protection may include object management issues, such as creating, renaming, or deleting objects. Protection is often enforced by a program called a reference monitor. A reference monitor records which subject may do what, and decides whether a subject is allowed to have a specific operation carried out. This monitor is called (e.g., by the underlying trusted operating system) each time an object is invoked. Consequently, it is extremely important that the reference monitor is itself tamperproof: an attacker must not be able to fool around with it.

Access Control Matrix

A common approach to modeling the access rights of subjects with respect to objects is to construct an access control matrix. Each subject is represented by a row in this matrix; each object is represented by a column. If the matrix is denoted M, then an entry $M[s,o]$ lists precisely which operations subject s can request to be carried out on object o. In other words, whenever a subject s requests the invocation of method m of object o, the reference monitor should check whether m is listed in $M[s,o]$. If m is not listed in $M[s,o]$, the invocation fails.

Considering that a system may easily need to support thousands of users and millions of objects that require protection, implementing an access control matrix as a true matrix is not the way to go. Many entries in the matrix will be empty: a single subject will generally have access to relatively few objects. Therefore, other, more efficient ways are followed to implement an access control matrix.

One widely-applied approach is to have each object maintain a list of the access rights of subjects that want to access the object. In essence, this means that the matrix is distributed column-wise across all objects, and that empty entries are left out. This type of implementation leads to what is called an Access Control List (ACL). Each object is assumed to have its own associated ACL.

Another approach is to distribute the matrix row-wise by giving each subject a list of capabilities it has for each object. In other words, a capability corresponds to an entry in the access control matrix. Not having a capability for a specific object means that the subject has no access rights for that object.

A capability can be compared to a ticket: its holder is given certain rights that are associated with that ticket. It is also clear that a ticket should be protected against modifications by its holder. One approach that is particularly suited in distributed systems and which has been applied extensively in Amoeba (Tanenbaum et al., 1990), is to protect (a list of) capabilities with a signature. We return to these and other matters later when discussing security management.

The difference between how ACLs and capabilities are used to protect the access to an object is shown in Fig. 9-26. Using ACLs, when a client sends a request to a server, the server's reference monitor will check whether it knows the client and if that client is known and allowed to have the requested operation carried out, as shown in Fig. 9-26(a).

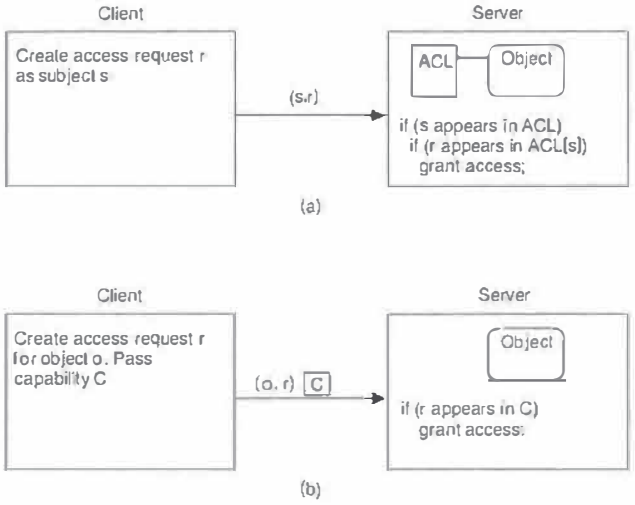

Figure 9-26. Comparison between ACLs and capabilities for protecting objects. (a) Using an ACL. (b) Using capabilities.

However, when using capabilities, a client simply sends its request to the server. The server is not interested in whether it knows the client; the capability says enough. Therefore, the server need only check whether the capability is valid and whether the requested operation is listed in the capability. This approach to protecting objects by means of capabilities is shown in Fig. 9-26(b).

Protection Domains

ACLs and capabilities help in efficiently implementing an access control matrix by ignoring all empty entries. Nevertheless, an ACL or a capability list can still become quite large if no further measures are taken.

One general way of reducing ACLs is to make use of protection domains. Formally, a **protection domain** is a set of *(object, access rights)* pairs. Each pair specifies for a given object exactly which operations are allowed to be carried out (Saltzer and Schroeder, 1975). Requests for carrying out an operation are always issued within a domain. Therefore, whenever a subject requests an operation to be carried out at an object, the reference monitor first looks up the protection domain associated with that request. Then, given the domain, the reference monitor can subsequently check whether the request is allowed to be carried out. Different uses of protection domains exist.

One approach is to construct groups of users. Consider, for example, a Web page on a company's internal intranet. Such a page should be available to every employee, but otherwise to no one else. Instead of adding an entry for each possible employee to the ACL for that Web page, it may be decided to have a separate group *Employee* containing all current employees. Whenever a user accesses the Web page, the reference monitor need only check whether that user is an employee. Which users belong to the group *Employee* is kept in a separate list (which, of course, is protected against unauthorized access).

Matters can be made more flexible by introducing hierarchical groups. For example, if an organization has three different branches at, say, Amsterdam, New York, and San Francisco, it may want to subdivide its *Employee* group into subgroups, one for each city, leading to an organization as shown in Fig. 9-27.

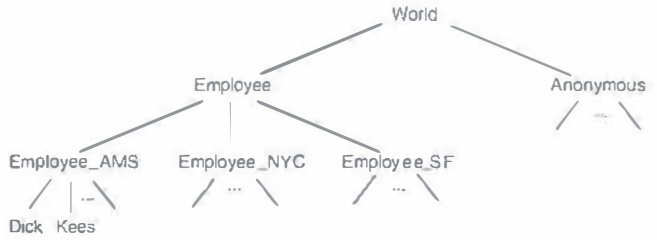

Figure 9-27. The hierarchical organization of protection domains as groups of users.

Accessing Web pages of the organization's intranet should be permitted by all employees. However, changing for example Web pages associated with the Amsterdam branch should be permitted only by a subset of employees in Amsterdam. If user Dick from Amsterdam wants to read a Web page from the intranet, the reference monitor needs to first look up the subsets *Employee_AMS*, *Employee_NYC*, and *Employee_SF* that jointly comprise the set *Employee*. It then has to check if one of these sets contains Dick. The advantage of having hierarchical groups is that managing group membership is relatively easy, and that very large groups can be constructed efficiently. An obvious disadvantage is that looking up a member can be quite costly if the membership database is distributed.

Instead of letting the reference monitor do all the work, an alternative is to let each subject carry a certificate listing the groups it belongs to. So, whenever Dick wants to read a Web page from the company's intranet, he hands over his certificate to the reference monitor stating that he is a member of *Employee_AMS*. To guarantee that the certificate is genuine and has not been tampered with, it should be protected by means of, for example, a digital signature. Certificates are seen to be comparable to capabilities. We return to these issues later.

Related to having groups as protection domains, it is also possible to implement protection domains as roles. In role-based access control, a user always logs into the system with a specific role, which is often associated with a function the user has in an organization (Sandhu et al., 1996). A user may have several functions. For example, Dick could simultaneously be head of a department, manager of a project, and member of a personnel search committee. Depending on the role he takes when logging in, he may be assigned different privileges. In other words, his role determines the protection domain (i.e., group) in which he will operate.

When assigning roles to users and requiring that users take on a specific role when logging in, it should also be possible for users to change their roles if necessary. For example, it may be required to allow Dick as head of the department to occasionally change to his role of project manager. Note that such changes are difficult to express when implementing protection domains only as groups.

Besides using protection domains, efficiency can be further improved by (hierarchically) grouping objects based on the operations they provide. For example, instead of considering individual objects, objects are grouped according to the interfaces they provide, possibly using subtyping [also referred to as interface inheritance, see Gamma et al. (1994)] to achieve a hierarchy. In this case, when a subject requests an operation to be carried out at an object, the reference monitor looks up to which interface the operation for that object belongs. It then checks whether the subject is allowed to call an operation belonging to that interface, rather than if it can call the operation for the specific object.

Combining protection domains and grouping of objects is also possible. Using both techniques, along with specific data structures and restricted operations on objects, Gladney (1997) describes how to implement ACLs for very large collections of objects that are used in digital libraries.

9.3.2 Firewalls

So far, we have shown how protection can be established using cryptographic techniques, combined with some implementation of an access control matrix. These approaches work fine as long as all communicating parties play according to the same set of rules. Such rules may be enforced when developing a standalone distributed system that is isolated from the rest of the world. However, matters become more complicated when outsiders are allowed to access the resources controlled by a distributed system. Examples of such accesses including sending mail, downloading files, uploading tax forms, and so on.

To protect resources under these circumstances, a different approach is needed. In practice, what happens is that external access to any part of a distributed system is controlled by a special kind of reference monitor known as a firewall (Cheswick and Bellovin, 2000; and Zwicky et al., 2000). Essentially, a firewall disconnects any part of a distributed system from the outside world, as shown in Fig. 9-28. All outgoing, but especially all incoming packets are routed through a special computer and inspected before they are passed. Unauthorized traffic is discarded and not allowed to continue. An important issue is that the firewall itself should be heavily protected against any kind of security threat: it should never fail.

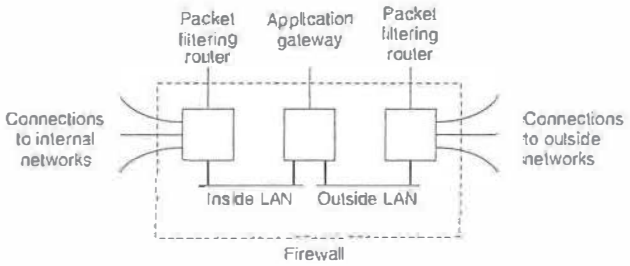

Figure 9-28. A common implementation of a firewall.

Firewalls essentially come in two different flavors that are often combined. An important type of firewall is a **packet-filtering gateway**. This type of firewall operates as a router and makes decisions as to whether or not to pass a network packet based on the source and destination address as contained in the packet's header. Typically, the packet-filtering gateway shown on the outside LAN in Fig. 9-28 would protect against incoming packets, whereas the one on the inside LAN would filter outgoing packets.

For example, to protect an internal Web server against requests from hosts that are not on the internal network, a packet-filtering gateway could decide to drop all incoming packets addressed to the Web server.

More subtle is the situation in which a company's network consists of multiple local-area networks connected, for example, through an SMDS network as we discussed before. Each LAN can be protected by means of a packet-filtering gateway, which is configured to pass incoming traffic only if it originated from a host on one of the other LANs. In this way, a private virtual network can be set up.

The other type of firewall is an **application-level gateway**. In contrast to a packet-filtering gateway, which inspects only the header of network packets, this type of firewall actually inspects the content of an incoming or outgoing message. A typical example is a mail gateway that discards incoming or outgoing mail exceeding a certain size. More sophisticated mail gateways exist that are, for example, capable of filtering spam e-mail.

Another example of an application-level gateway is one that allows external access to a digital library server, but will supply only abstracts of documents. If an external user wants more, an electronic payment protocol is started. Users inside the firewall have direct access to the library service.

A special kind of application-level gateway is what is known as a proxy gateway. This type of firewall works as a front end to a specific kind of application, and ensures that only those messages are passed that meet certain criteria. Consider, for example, surfing the Web. As we discuss in the next section, many Web pages contain scripts or applets that are to be executed in a user's browser. To prevent such code to be downloaded to the inside LAN, all Web traffic could be directed through a Web proxy gateway. This gateway accepts regular HTTP requests, either from inside or outside the firewall. In other words, it appears to its users as a normal Web server. However, it filters all incoming and outgoing traffic, either by discarding certain requests and pages, or modifying pages when they contain executable code.

9.3.3 Secure Mobile Code

As we discussed in Chap. 3, an important development in modern distributed systems is the ability to migrate code between hosts instead of just migrating passive data. However, mobile code introduces a number of serious security threats. For one thing, when sending an agent across the Internet, its owner will want to protect it against malicious hosts that try to steal or modify information carried by the agent.

Another issue is that hosts need to be protected against malicious agents. Most users of distributed systems will not be experts in systems technology and have no way of telling whether the program they are fetching from another host can be trusted not to corrupt their computer. In many cases it may be difficult even for an expert to detect that a program is actually being downloaded at all.

Unless security measures are taken, once a malicious program has settled itself in a computer, it can easily corrupt its host. We are faced with an access control problem: the program should not be allowed unauthorized access to the host's resources. As we shall see, protecting a host against downloaded malicious programs is not always easy. The problem is not so much as to avoid downloading of programs. Instead, what we are looking for is supporting mobile code that we can allow access to local resources in a flexible, yet fully controlled manner.

Protecting an Agent

Before we take a look at protecting a computer system against downloaded malicious code, let us first take a look at the opposite situation. Consider a mobile agent that is roaming a distributed system on behalf of a user. Such an agent may be searching for the cheapest airplane ticket from Nairobi to Malindi, and has been authorized by its owner to make a reservation as soon as it found a flight. For this purpose, the agent may carry an electronic credit card.

Obviously, we need protection here. Whenever the agent moves to a host, that host should not be allowed to steal the agent's credit card information. Also, the agent should be protected against modifications that make the owner pay much more than actually is needed. For example, if Chuck's Cheaper Charters can see that the agent has not yet visited its cheaper competitor Alice Airlines, Chuck should be prevented from changing the agent so that it will not visit Alice Airlines' host. Other examples that require protection of an agent against attacks from a hostile host include maliciously destroying an agent, or tampering with an agent such that it will attack or steal from its owner when it returns.

Unfortunately, fully protecting an agent against all kinds of attacks is impossible (Farmer et al., 1996). This impossibility is primarily caused by the fact that no hard guarantees can be given that a host will do what it promises. An alternative approach is therefore to organize agents in such a way that modifications can at least be detected. This approach has been followed in the Ajanta system (Karnik and Tripathi, 2001). Ajanta provides three mechanisms that allow an agent's owner to detect that the agent has been tampered with: read-only state, append-only logs, and selective revealing of state to certain servers.

The read-only state of an Ajanta agent consists of a collection of data items that is signed by the agent's owner. Signing takes place when the agent is constructed and initialized before it is sent off to other hosts. The owner first constructs a message digest, which it subsequently encrypts with its private key. When the agent arrives at a host, that host can easily detect whether the read-only state has been tampered with by verifying the state against the signed message digest of the original state.

To allow an agent to collect information while moving between hosts, Ajanta provides secure append-only logs. These logs are characterized by the fact that data can only be appended to the log; there is no way that data can be removed or modified without the owner being able to detect this. Using an append-only log works as follows. Initially, the log is empty and has only an associated checksum C_{init} calculated as $C_{init} = K^{+}_{owner}(N)$, where K^{+}_{owner} is the public key of the agent's owner, and N is a secret nonce known only to the owner.

When the agent moves to a server S that wants to hand it some data X, S appends X to the log, then signs X with its signature $sig(S,X)$, and calculates a checksum:

$$C_{new} = K^{+}_{owner}(C_{old}, sig(S,X), S)$$

where C_{old} is the checksum that was used previously.

When the agent comes back to its owner, the owner can easily verify whether the log has been tampered with. The owner starts reading the log at the end by successively computing $K^{-}_{owner}(C)$ on the checksum C. Each iteration returns a checksum C_{next} for the next iteration, along with $sig(S,X)$ and S for some server S. The owner can then verify whether or not the then-last element in the log matches $sig(S,X)$. If so, the element is removed and processed, after which the next iteration step is taken. The iteration stops when the initial checksum is reached, or when the owner notices that the log as been tampered with because a signature does not match.

Finally, Ajanta supports selective revealing of state by providing an array of data items, where each entry is intended for a designated server. Each entry is encrypted with the designated server's public key to ensure confidentiality. The entire array is signed by the agent's owner to ensure integrity of the array as a whole. In other words, if *any* entry is modified by a malicious host, any of the designated servers will notice and can take appropriate action.

Besides protecting an agent against malicious hosts, Ajanta also provides various mechanisms to protect hosts against malicious agents. As we discuss next, many of these mechanisms are also supplied by other systems that support mobile code. Further information on Ajanta can be found in Tripathi et al. (1999).

Protecting the Target

Although protecting mobile code against a malicious host is important, more attention has been paid to protecting hosts against malicious mobile code. If sending an agent into the outside world is considered too dangerous, a user will generally have alternatives to get the job done for which the agent was intended. However, there are often no alternatives to letting an agent into your system, other than locking it out completely. Therefore, if it is once decided that the agent can come in, the user needs full control over what the agent can do.

As we just discussed, although protecting an agent from modification may be impossible, at least it is possible for the agent's owner to detect that modifications have been made. At worst, the owner will have to discard the agent when it returns, but otherwise no harm will have been done. However, when dealing with malicious incoming agents, simply detecting that your resources have been harassed is too late. Instead, it is essential to protect all resources against unauthorized access by downloaded code.

One approach to protection is to construct a sandbox. A **sandbox** is a technique by which a downloaded program is executed in such a way that each of its instructions can be fully controlled. If an attempt is made to execute an instruction that has been forbidden by the host, execution of the program will be stopped. Likewise, execution is halted when an instruction accesses certain registers or areas in memory that the host has not allowed.

Implementing a sandbox is not easy. One approach is to check the executable code when it is downloaded, and to insert additional instructions for situations that can be checked only at runtime (Wahbe et al., 1993). Fortunately, matters become much simpler when dealing with interpreted code. Let us briefly consider the approach taken in Java [see also MacGregor et al. (1998)]. Each Java program consists of a number of classes from which objects are created. There are no global variables and functions; everything has to be declared as part of a class. Program execution starts at a method called main. A Java program is compiled to a set of instructions that are interpreted by what is called the Java Virtual Machine (JVM). For a client to download and execute a compiled Java program, it is therefore necessary that the client process is running the JVM. The JVM will subsequently handle the actual execution of the downloaded program by interpreting each of its instructions, starting at the instructions that comprise main.

In a Java sandbox, protection starts by ensuring that the component that handles the transfer of a program to the client machine can be trusted. Downloading in Java is taken care of by a set of **class loaders**. Each class loader is responsible for fetching a specified class from a server and installing it in the client's address space so that the JVM can create objects from it. Because a class loader is just another Java class, it is possible that a downloaded program contains its own class loaders. The first thing that is handled by a sandbox is that exclusively trusted class loaders are used. In particular, a Java program is not allowed to create its own class loaders by which it could circumvent the way class loading is normally handled.

The second component of a Java sandbox consists of a **byte code verifier**, which checks whether a downloaded class obeys the security rules of the sandbox. In particular, the verifier checks that the class contains no illegal instructions or instructions that could somehow corrupt the stack or memory. Not all classes are checked, as shown in Fig. 9-29; only the ones that are downloaded from an external server to the client. Classes that are located on the client's machine are generally trusted, although their integrity could also be easily verified.

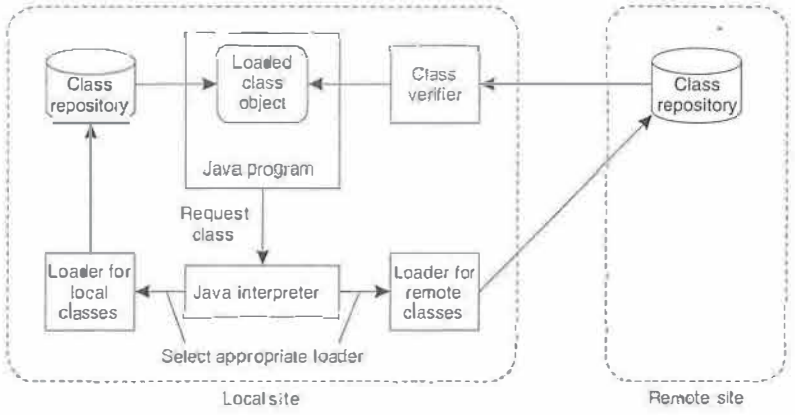

Figure 9-29. The organization of a Java sandbox.

Finally, when a class has been securely downloaded and verified, the JVM can instantiate objects from it and execute those object's methods. To further prevent objects from unauthorized access to the client's resources, a **security manager** is used to perform various checks at runtime. Java programs intended to be downloaded are forced to make use of the security manager; there is no way

they can circumvent it. This means, for example, that any I/O operation is vetted for validity and will not be carried out if the security manager says "no." The security manager thus plays the role of a reference monitor we discussed earlier.

A typical security manager will disallow many operations to be carried out. For example, virtually all security managers deny access to local files and allow a program only to set up a connection to the server from where it came. Manipulating the JVM is obviously not allowed as well. However, a program is permitted to access the graphics library for display purposes and to catch events such as moving the mouse or clicking its buttons.

The original Java security manager implemented a rather strict security policy in which it made no distinction between different downloaded programs, or even programs from different servers. In many cases, the initial Java sandbox model was overly restricted and more flexibility was required. Below, we discuss an alternative approach that is currently followed.

An approach in line with sandboxing, but which offers somewhat more flexibility, is to create a playground for downloaded mobile code (Malkhi and Reiter, 2000). A playground is a separate, designated machine exclusively reserved for running mobile code. Resources local to the playground, such as files or network connections to external servers are available to programs executing in the playground, subject to normal protection mechanisms. However, resources local to other machines are physically disconnected from the playground and cannot be accessed by downloaded code. Users on these other machines can access the playground in a traditional way, for example, by means of RPCs. However, no mobile code is ever downloaded to machines not in the playground. This distinction between a sandbox and a playground is shown in Fig. 9-30.

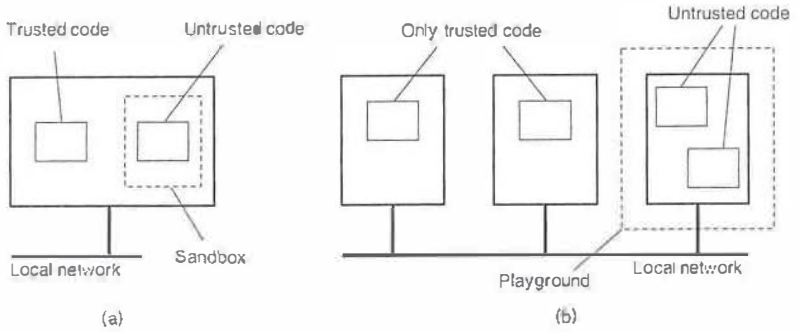

Figure 9-30. (a) A sandbox. (b) A playground.

A next step toward increased flexibility is to require that each downloaded program can be authenticated, and to subsequently enforce a specific security policy based on where the program came from. Demanding that programs can be

authenticated is relatively easy: mobile code can be signed, just like any other document. This code-signing approach is often applied as an alternative to sandboxing as well. In effect, only code from trusted servers is accepted.

However, the difficult part is enforcing a security policy. Wallach et al. (1997) propose three mechanisms in the case of Java programs. The first approach is based on the use of object references as capabilities. To access a local resource such as a file, a program must have been given a reference to a specific object that handles file operations when it was downloaded. If no such reference is given, there is no way that files can be accessed. This principle is shown in Fig. 9-31.

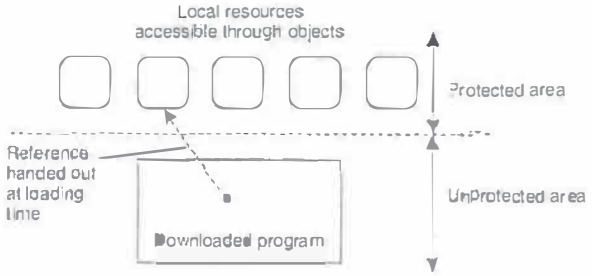

Figure 9-31. The principle of using Java object references as capabilities.

All interfaces to objects that implement the file system are initially hidden from the program by simply not handing out any references to these interfaces. Java's strong type checking ensures that it is impossible to construct a reference to one of these interfaces at runtime. In addition, we can use the property of Java to keep certain variables and methods completely internal to a class. In particular, a program can be prevented from instantiating its own file-handling objects, by essentially hiding the operation that creates new objects from a given class. (In Java terminology, a constructor is made private to its associated class.)

The second mechanism for enforcing a security policy is (extended) stack introspection. In essence, any call to a method m of a local resource is preceded by a call to a special procedure enable_privilege that checks whether the caller is authorized to invoke m on that resource. If the invocation is authorized, the caller is given temporary privileges for the duration of the call. Before returning control to the invoker when m is finished, the special procedure disable_privilege is invoked to disable these privileges.

To enforce calls to enable_privilege and disable_privilege, a developer of interfaces to local resources could be required to insert these calls in the appropriate places. However, it is much better to let the Java interpreter handle the calls automatically. This is the standard approach followed in, for example, Web browsers for dealing with Java applets. An elegant solution is as follows. Whenever an

invocation to a local resource is made, the Java interpreter automatically calls enable_privilege, which subsequently checks whether the call is permitted. If so, a call to disable_privilege is pushed on the stack to ensure that privileges are disabled when the method call returns. This approach prevents malicious programmers from circumventing the rules.

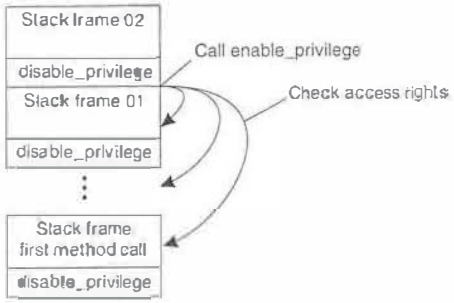

Figure 9-32. The principle of stack introspection.

Another important advantage of using the stack is that it enables a much better way of checking privileges. Suppose that a program invokes a local object $O1$, which, in turn, invokes object $O2$. Although $O1$ may have permission to invoke $O2$, if the invoker of $O1$ is not trusted to invoke a specific method belonging to $O2$, then this chain of invocations should not be allowed. Stack introspection makes it easy to check such chains, as the interpreter need merely inspect each stack frame starting at the top to see if there is a frame having the right privileges enabled (in which case the call is permitted), or if there is a frame that explicitly forbids access to the current resource (in which case the call is immediately terminated). This approach is shown in Fig. 9-32.

In essence, stack introspection allows for the attachment of privileges to classes or methods, and the checking of those privileges for each caller separately. In this way, it is possible to implement class-based protection domains, as is explained in detail in Gong and Schemers (1998).

The third approach to enforcing a security policy is by means of **name space management**. The idea is put forth below. To give programs access to local resources, they first need to attain access by including the apptopriate files that contain the classes implementing those resources. Inclusion requires that a name is given to the interpreter, which then resolves it to a class, which is subsequently loaded at runtime. To enforce a security policy for a specific downloaded program, the same name can be resolved to different classes, depending on where the downloaded program came from. Typically, name resolution is handled by class loaders, which need to be adapted to implement this approach. Details of how this can be done can be found in Wallach et al. (1997).

The approach described so far associates privileges with classes or methods based on where a downloaded program came from. By virtue of the Java interpreter, it is possible to enforce security policies through the mechanisms described above. In this sense, the security architecture becomes highly language dependent, and will need to be developed anew for other languages. Language-independent solutions, such as, for example, described in Jaeger et al. (1999), require a more general approach to enforcing security, and are also harder to implement. In these cases, support is needed from a secure operating system that is aware of downloaded mobile code and which enforces all calls to local resources to run through the kernel where subsequent checking is done.

9.3.4 Denial of Service

Access control is generally about carefully ensuring that resources are accessed only by authorized processes. A particularly annoying type of attack that is related to access control is maliciously preventing authorized processes from accessing resources. Defenses against such **denial-of-service (DoS) attacks** are becoming increasingly important as distributed systems are opened up through the Internet. Where DoS attacks that come from one or a few sources can often be handled quite effectively, matters become much more difficult when having to deal with **distributed denial of service (DDoS)**.

In DDoS attacks, a huge collection of processes jointly attempt to bring down a networked service. In these cases, we often see that the attackers have succeeded in hijacking a large group of machines which unknowingly participate in the attack. Specht and Lee (2004) distinguish two types of attacks: those aimed at bandwidth depletion and those aimed at resource depletion.

Bandwidth depletion can be accomplished by simply sending many messages to a single machine. The effect is that normal messages will hardly be able to reach the receiver. Resource depletion attacks concentrate on letting the receiver use up resources on otherwise useless messages. A well-known resource-depletion attack is TCP SYN-flooding. In this case, the attacker attempts to initiate a huge amount of connections (i.e., send SYN packets as part of the three-way handshake), but will otherwise never respond to acknowledgments from the receiver.

There is no single method to protect against DDoS attacks. One problem is that attackers make use of innocent victims by secretly installing software on their machines. In these cases, the only solution is to have machines continuously monitor their state by checking files for pollution. Considering the ease by which a virus can spread over the Internet, relying only on this countermeasure is not feasible.

Much better is to continuously monitor network traffic, for example, starting at the egress routers where packets leave an organization's network. Experience shows that by dropping packets whose source address does not belong to the

organization's network we can prevent a lot of havoc. In general, the more packets can be filtered close to the sources, the better.

Alternatively, it is also possible to concentrate on ingress routers, that is, where traffic flows into an organization's network. The problem is that detecting an attack at an ingress router is too late as the network will probably already be unreachable for regular traffic. Better is to have routers further in the Internet, such as in the networks of ISPs, start dropping packets when they suspect that an attack is going on. This approach is followed by Gil and Poletto (2001), where a router will drop packets when it notices that the rate between the number of packets *to* a specific node is disproportionate to the number of packets *from* that node.

In general, a myriad of techniques need to be deployed, whereas new attacks continue to emerge. A practical overview of the state-of-the-art in denial-of-service attacks and solutions can be found in Mirkovic et al. (2005); a detailed taxonomy is presented in Mirkovic and Reiher (2004).

9.4 SECURITY MANAGEMENT

So far, we have considered secure channels and access control, but have hardly touched upon the issue how, for example, keys are obtained. In this section, we take a closer look at security management. In particular, we distinguish three different subjects. First, we need to consider the general management of cryptographic keys, and especially the means by which public keys are distributed. As it turns out, certificates play an important role here.

Second, we discuss the problem of securely managing a group of servers by concentrating on the problem of adding a new group member that is trusted by the current members. Clearly, in the face of distributed and replicated services, it is important that security is not compromised by admitting a malicious process to a group.

Third, we pay attention to authorization management by looking at capabilities and what are known as attribute certificates. An important issue in distributed systems with respect to authorization management is that one process can delegate some or all of its access rights to another process. Delegating rights in a secure way has its own subtleties as we also discuss in this section.

9.4.1 Key Management

So far, we have described various cryptographic protocols in which we (implicitly) assumed that various keys were readily available. For example, in the case of public-key cryptosystems, we assumed that a sender of a message had the public key of the receiver at its disposal so that it could encrypt the message to ensure confidentiality. Likewise, in the case of authentication using a key distribution center (KDC), we assumed each party already shared a secret key with the KDC.

However, establishing and distributing keys is not a trivial matter. For example, distributing secret keys by means of an unsecured channel is out of the question and in many cases we need to resort to out-of-band methods. Also, mechanisms are needed to revoke keys, that is, prevent a key from being used after it has been compromised or invalidated. For example, revocation is necessary when a key has been compromised.

Key Establishment

Let us start with considering how session keys can be established. When Alice wants to set up a secure channel with Bob, she may first use Bob's public key to initiate communication as shown in Fig. 9-19. If Bob accepts, he can subsequently generate the session key and return it to Alice encrypted with Alice's public key. By encrypting the shared session key before its transmission, it can be safely passed across the network.

A similar scheme can be used to generate and distribute a session key when Alice and Bob already share a secret key. However, both methods require that the communicating parties already have the means available to establish a secure channel. In other words, some form of key establishment and distribution must already have taken place. The same argument applies when a shared secret key is established by means of a trusted third party, such as a KDC.

An elegant and widely-applied scheme for establishing a shared key across an insecure channel is the **Diffie-Hellman key exchange** (Diffie and Hellman, 1976). The protocol works as follows. Suppose that Alice and Bob want to establish a shared secret key. The first requirement is that they agree on two large numbers, n and g that are subject to a number of mathematical properties (which we do not discuss here). Both n and g can be made public; there is no need to hide them from outsiders. Alice picks a large random number, say x, which she keeps secret. Likewise, Bob picks his own secret large number, say y. At this point there is enough information to construct a secret key, as shown in Fig. 9-33.

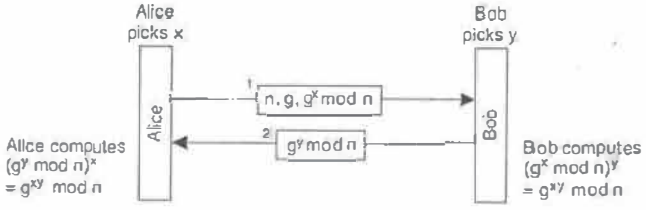

Figure 9-33. The principle of Diffie-Hellman key exchange.

Alice starts by sending $g^x \bmod n$ to Bob, along with n and g. It is important to note that this information can be sent as plaintext, as it is virtually impossible to

compute x given $g^x \bmod n$. When Bob receives the message, he subsequently calculates $(g^x \bmod n)^y$ which is mathematically equal to $g^{xy} \bmod n$. In addition, he sends $g^y \bmod n$ to Alice, who can then compute $(g^y \bmod n)^x = g^{xy} \bmod n$. Consequently, both Alice and Bob, and only those two, will now have the shared secret key $g^{xy} \bmod n$. Note that neither of them needed to make their private number (x and y, respectively), known to the other.

Diffie-Hellman can be viewed as a public-key cryptosystem. In the case of Alice, x is her private key, whereas $g^x \bmod n$ is her public key. As we discuss next, securely distributing the public key is essential to making Diffie-Hellman work in practice.

Key Distribution

One of the more difficult parts in key management is the actual distribution of initial keys. In a symmetric cryptosystem, the initial shared secret key must be communicated along a secure channel that provides authentication as well as confidentiality, as shown in Fig. 9-34(a). If there are no keys available to Alice and Bob to set up such a secure channel, it is necessary to distribute the key out-of-band. In other words, Alice and Bob will have to get in touch with each other using some other communication means than the network. For example, one of them may phone the other, or send the key on a floppy disk using snail mail.

In the case of a public-key cryptosystem, we need to distribute the public key in such a way that the receivers can be sure that the key is indeed paired to a claimed private key. In other words, as shown in Fig. 9-34(b), although the public key itself may be sent as plaintext, it is necessary that the channel through which it is sent can provide authentication. The private key, of course, needs to be sent across a secure channel providing authentication as well as confidentiality.

When it comes to key distribution, the authenticated distribution of public keys is perhaps the most interesting. In practice, public-key distribution takes place by means of **public-key certificates**. Such a certificate consists of a public key together with a string identifying the entity to which that key is associated. The entity could be a user, but also a host or some special device. The public key and identifier have together been signed by a **certification authority**, and this signature has been placed on the certificate as well. (The identity of the certification authority is naturally part of the certificate.) Signing takes place by means of a private key K_{CA}^{-} that belongs to the certification authority. The corresponding public key K_{CA}^{+} is assumed to be well known. For example, the public keys of various certification authorities are built into most Web browsers and shipped with the binaries.

Using a public-key certificate works as follows. Assume that a client wishes to ascertain that the public key found in the certificate indeed belongs to the identified entity. It then uses the public key of the associated certification authority to verify the certificate's signature. If the signature on the certificate matches the

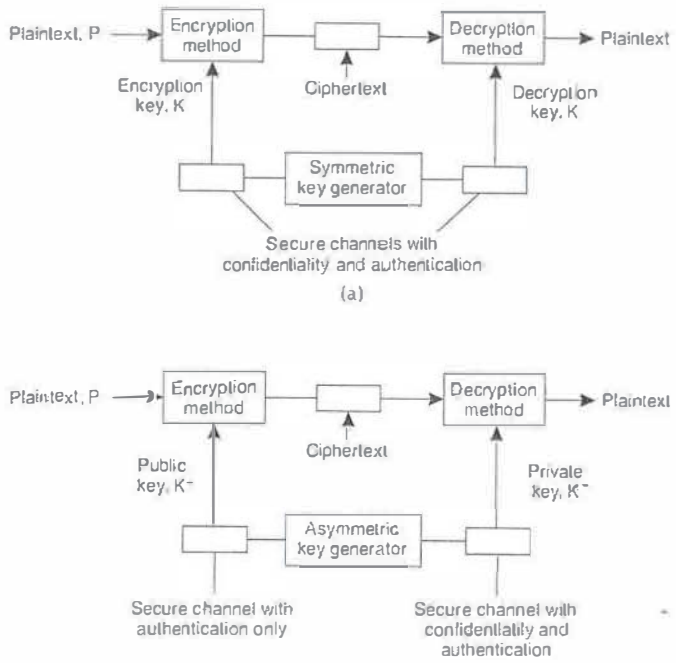

Figure 9-34. (a) Secret-key distribution. (b) Public-key distribution [see also Menezes et al. (1996)].

(public key, identifier)-pair, the client accepts that the public key indeed belongs to the identified entity.

It is important to note that by accepting the certificate as being in order, the client actually trusts that the certificate has not been forged. In particular, the client must assume that the public key K_{CA}^{+} indeed belongs to the associated certification authority. If in doubt, it should be possible to verify the validity of K_{CA}^{+} through another certificate coming from a different, possibly more trusted certification authority.

Such hierarchical trust models in which the highest-level certification authority must be trusted by everyone, are not uncommon. For example, **Privacy Enhanced Mail (PEM)** uses a three-level trust model in which lowest-level certification authorities can be authenticated by **Policy Certification Authorities (PCA)**, which in turn can be authenticated by the **Internet Policy Registration Authority (IPRA)**. If a user does not trust the IPRA, or does not think he can

safely talk to the IPRA, there is no hope he will ever trust e-mail messages to be sent in a secure way when using PEM. More information on this model can be found in Kent (1993). Other trust models are discussed in Menezes et al. (1996).

Lifetime of Certificates

An important issue concerning certificates is their longevity. First let us consider the situation in which a certification authority hands out lifelong certificates. Essentially, what the certificate then states is that the public key will always be valid for the entity identified by the certificate. Clearly, such a statement is not what we want. If the private key of the identified entity is ever compromised, no unsuspecting client should ever be able to use the public key (let alone malicious clients). In that case, we need a mechanism to revoke the certificate by making it publicly-known that the certificate is no longer valid.

There are several ways to revoke a certificate. One common approach is with a **Certificate Revocation List (CRL)** published regularly by the certification authority. Whenever a client checks a certificate, it will have to check the CRL to see whether the certificate has been revoked or not. This means that the client will at least have to contact the certification authority each time a new CRL is published. Note that if a CRL is published daily, it also takes a day to revoke a certificate. Meanwhile, a compromised certificate can be falsely used until it is published on the next CRL. Consequently, the time between publishing CRLs cannot be too long. In addition, getting a CRL incurs some overhead.

An alternative approach is to restrict the lifetime of each certificate. In essence, this approach is analogous to handing out leases as we discussed in Chap. 6. The validity of a certificate automatically expires after some time. If for whatever reason the certificate should be revoked before it expires, the certification authority can still publish it on a CRL. However, this approach will still force clients to check the latest CRL whenever verifying a certificate. In other words, they will need to contact the certification authority or a trusted database containing the latest CRL.

A final extreme case is to reduce the lifetime of a certificate to nearly zero. In effect, this means that certificates are no longer used; instead, a client will always have to contact the certification authority to check the validity of a public key. As a consequence, the certification authority must be continuously online.

In practice, certificates are handed out with restricted lifetimes. In the case of Internet applications, the expiration time is often as much as a year (Stein, 1998). Such an approach requires that CRLs are published regularly, but that they are also inspected when certificates are checked. Practice indicates that client applications hardly ever consult CRLs and simply assume a certificate to be valid until it expires. In this respect, when it comes to Internet security in practice, there is still much room for improvement, unfortunately.

9.4.2 Secure Group Management

Many security systems make use of special services such as Key Distribution Centers (KDCs) or Certification Authorities (CAs). These services demonstrate a difficult problem in distributed systems. In the first place, they must be trusted. To enhance the trust in security services, it is necessary to provide a high degree of protection against all kinds of security threats. For example, as soon as a CA has been compromised, it becomes impossible to verify the validity of a public key, making the entire security system completely worthless.

On the other hand, it is also necessary that many security services offer high availability. For example, in the case of a KDC, each time two processes want to set up a secure channel, at least one of them will need to contact the KDC for a shared secret key. If the KDC is not available, secure communication cannot be established unless an alternative technique for key establishment is available, such as the Diffie-Hellman key exchange.

The solution to high availability is replication. On the other hand, replication makes a server more vulnerable to security attacks. We already discussed how secure group communication can take place by sharing a secret among the group members. In effect, no single group member is capable of compromising certificates, making the group itself highly secure. What remains to consider is how to actually manage a group of replicated servers. Reiter et al. (199-) propose the following solution.

The problem that needs to be solved is to ensure that when a process asks to join a group G, the integrity of the group is not compromised. A group G is assumed to use a secret key CK_G shared by all group members for encrypting group messages. In addition, it also uses a public/private key pair (K_G^+, K_G^-) for communication with nongroup members.

Whenever a process P wants to join a group G, it sends a join request JR identifying G and P. P's local time T, a generated *reply pad* RP and a generated secret key $K_{P,G}$. RP and $K_{P,G}$ are jointly encrypted using the group's public key K_G^+, as shown as message 1 in Fig. 9-35. The use of RP and $K_{P,G}$ is explained in more detail below. The join request JR is signed by P, and is sent along with a certificate containing P's public key. We have used the widely-applied notation $[M]_A$ to denote that message M has been signed by subject A.

When a group member Q receives such a join request, it first authenticates P, after which communication with the other group members takes place to see whether P can be admitted as a group member. Authentication of P takes place in the usual way by means of the certificate. The timestamp T is used to make sure that the certificate was still valid at the time it was sent. (Note that we need to be sure that the time has not been tampered with as well.) Group member Q verifies the signature of the certification authority and subsequently extracts P's public key from the certificate to check the validity of JR. At that point, a group-specific protocol is followed to see whether all group members agree on admitting P.

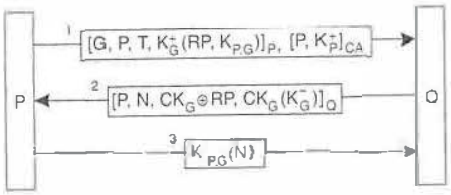

Figure 9-35. Securely admitting a new group member.

If P is allowed to join the group, Q returns a group admittance message GA, shown as message 2 in Fig. 9-35, identifying P and containing a nonce N. The reply pad RP is used to encrypt the group's communication key CK_G. In addition, P will also need the group's private key K_G^-, which is encrypted with CK_G. Message GA is subsequently signed by Q using key $K_{P,G}$.

Process P can now authenticate Q, because only a true group member can have discovered the secret key $K_{P,G}$. The nonce N in this protocol is not used for security; instead, when P sends back N encrypted with $K_{P,G}$ (message 3), Q then knows that P has received all the necessary keys, and has therefore now indeed joined the group.

Note that instead of using the reply pad RP, P and Q could also have encrypted CK_G using P's public key. However, because RP is used only once, namely for the encryption of the group's communication key in message GA, using RP is safer. If P's private key were ever to revealed, it would become possible to also reveal CK_G, which would compromise the secrecy of all group communication.

9.4.3 Authorization Management

Managing security in distributed systems is also concerned with managing access rights. So far, we have hardly touched upon the issue of how access rights are initially granted to users or groups of users, and how they are subsequently maintained in an unforgeable way. It is time to correct this omission.

In nondistributed systems, managing access rights is relatively easy. When a new user is added to the system, that user is given initial rights, for example, to create files and subdirectories in a specific directory, create processes, use CPU time, and so on. In other words, a complete account for a user is set up for one specific machine in which all rights have been specified in advance by the system administrators.

In a distributed system, matters are complicated by the fact that resources are spread across several machines. If the approach for nondistributed systems were to be followed, it would be necessary to create an account for each user on each machine. In essence, this is the approach followed in network operating systems.

Matters can be simplified a bit by creating a single account on a central server. That server is consulted each time a user accesses certain resources or machines.

Capabilities and Attribute Certificates

A much better approach that has been widely applied in distributed systems is the use of capabilities. As we explained briefly above, a capability is an unforgeable data structure for a specific resource, specifying exactly the access rights that the holder of the capability has with respect to that resource. Different implementations of capabilities exist. Here, we briefly discuss the implementation as used in the Amoeba operating system (Tanenbaum et al., 1986).

Amoeba was one of the first object-based distributed systems. Its model of distributed objects is that of remote objects. In other words, an object resides at a server while clients are offered transparent access to that object by means of a proxy. To invoke an operation on an object, a client passes a capability to its local operating system, which then locates the server where the object resides and subsequently does an RPC to that server.

A capability is a 128-bit identifier, internally organized as shown in Fig. 9-36. The first 48 bits are initialized by the object's server when the object is created and effectively form a machine-independent identifier of the object's server, referred to as the server port. Amoeba uses broadcasting to locate the machine where the server is currently located.

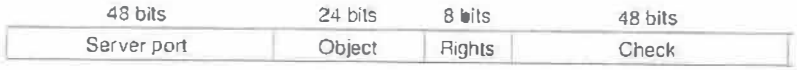

Figure 9-36. A capability in Amoeba.

The next 24 bits are used to identify the object at the given server. Note that the server port together with the object identifier form a 72-bit systemwide unique identifier for every object in Amoeba. The next 8 bits are used to specify the access rights of the holder of the capability. Finally, the 48-bits *check* field is used to make a capability unforgeable, as we explain in the following pages.

When an object is created, its server picks a random *check* field and stores it both in the capability as well as internally in its own tables. All the right bits in a new capability are initially on, and it is this owner capability that is returned to the client. When the capability is sent back to the server in a request to perform an operation, the *check* field is verified.

To create a restricted capability, a client can pass a capability back to the server, along with a bit mask for the new rights. The server takes the original *check* field from its tables, XORs it with the new rights (which must be a subset of the rights in the capability), and then runs the result through a one-way function.

The server then creates a new capability, with the same value in the *object* field, but with the new rights bits in the *rights* field and the output of the one-way function in the *check* field. The new capability is then returned to the caller. The client may send this new capability to another process, if it wishes.

The method of generating restricted capabilities is illustrated in Fig. 9-37. In this example, the owner has turned off all the rights except one. For example, the restricted capability might allow the object to be read, but nothing else. The meaning of the *rights* field is different for each object type since the legal operations themselves also vary from object type to object type.

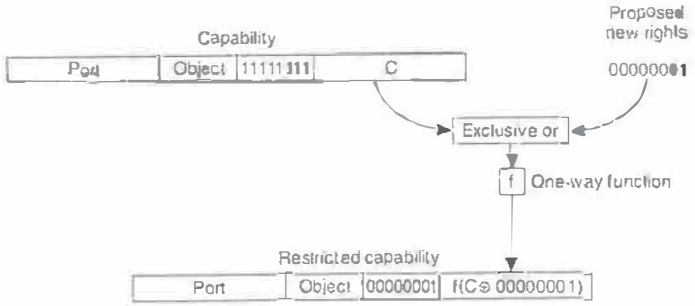

Figure 9-37. Generation of a restricted capability from an owner capability.

When the restricted capability comes back to the server, the server sees from the *rights* field that it is not an *owner capability* because at least one bit is turned off. The server then fetches the original random number from its tables, XORs it with the *rights* field from the capability, and runs the result through the one-way function. If the result agrees with the *check* field, the capability is accepted as valid.

It should be obvious from this algorithm that a user who tries to add rights that he does not have will simply invalidate the capability. Inverting the *check* field in a restricted capability to get the argument (*C* XOR 00000001 in Fig. 9-37) is impossible because the function f is a one-way function. It is through this cryptographic technique that capabilities are protected from tampering. Note that f essentially does the same as computing a message digest as discussed earlier. Changing anything in the original message (like inverting a bit), will immediately be detected.

A generalization of capabilities that is sometimes used in modern distributed systems is the **attribute certificate**. Unlike the certificates discussed above that are used to verify the validity of a public key, attribute certificates are used to list certain *(attribute, value)*-pairs that apply to an identified entity. In particular, attribute certificates can be used to list the access rights that the holder of a certificate has with respect to the identified resource.

Like other certificates, attribute certificates are handed out by special certification authorities, usually called **attribute certification authorities**. Compared to Amoeba's capabilities, such an authority corresponds to an object's server. In general, however, the attribute certification authority and the server managing the entity for which a certificate has been created need not be the same. The access rights listed in a certificate are signed by the attribute certification authority.

Delegation

Now consider the following problem. A user wants to have a large file printed for which he has read-only access rights. In order not to bother others too much, the user sends a request to the print server, asking it to start printing the file no earlier than 2 o'clock in the morning. Instead of sending the entire file to the printer, the user passes the file name to the printer so that it can copy it to its spooling directory, if necessary, when actually needed.

Although this scheme seems to be perfectly in order, there is one major problem: the printer will generally not have the appropriate access permissions to the named file. In other words, if no special measures are taken, as soon as the print server wants to read the file in order to print it, the system will deny the server access to the file. This problem could have been solved if the user had temporarily **delegated** his access rights for the file to the print server.

Delegation of access rights is an important technique for implementing protection in computer systems and distributed systems, in particular. The basic idea is simple: by passing certain access rights from one process to another, it becomes easier to distribute work between several processes without adversely affecting the protection of resources. In the case of distributed systems, processes may run on different machines and even within different administrative domains as we discussed for Globus. Delegation can avoid much overhead as protection can often be handled locally.

There are several ways to implement delegation. A general approach as described in Neuman (1993), is to make use of a proxy. A proxy in the context of security in computer systems is a token that allows its owner to operate with the same or restricted rights and privileges as the subject that granted the token. (Note that this notion of a proxy is different from a proxy as a synonym for a client-side stub. Although we try to avoid overloading terms, we make an exception here as the term "proxy" in the definition above is too widely used to ignore.) A process can create a proxy with at best the same rights and privileges it has itself. If a process creates a new proxy based on one it currently has, the derived proxy will have at least the same restrictions as the original one, and possibly more.

Before considering a general scheme for delegation, consider the following two approaches. First, delegation is relatively simple if Alice knows everyone. If she wants to delegate rights to Bob, she merely needs to construct a certificate

saying "Alice says Bob has rights R." such as $[A,B,R]_A$. If Bob wants to pass some of these rights to Charlie, he will ask Charlie to contact Alice and ask her for an appropriate certificate.

In a second simple case Alice can simply construct a certificate saying "The bearer of this certificate has rights R." However, in this case we need to protect the certificate against illegal copying, as is done with securely passing capabilities between processes. Neuman's scheme handles this case, as well as avoiding the issue that Alice needs to know everyone to whom rights need to be delegated.

A proxy in Neuman's scheme has two parts, as illustrated in Fig. 9-38. Let A be the process that created the proxy. The first part of the proxy is a set $C = \{R, S^+_{proxy}\}$, consisting of a set R of access rights that have been delegated by A, along with a publicly-known part of a secret that is used to authenticate the holder of the certificate. We will explain the use of S^+_{proxy} below. The certificate carries the signature $sig(A,C)$ of A, to protect it against modifications. The second part contains the other part of the secret, denoted as S^-_{proxy}. It is essential that S^-_{proxy} is protected against disclosure when delegating rights to another process.

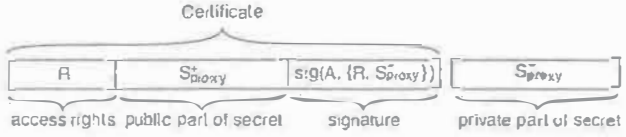

Figure 9-38. The general structure of a proxy as used for delegation.

Another way of looking at the proxy is as follows. If Alice wants to delegate some of her rights to Bob, she makes a list of rights (R) that Bob can exercise. By signing the list, she prevents Bob from tampering with it. However, having only a signed list of rights is often not enough. If Bob wants to exercise his rights, he may have to prove that he actually got the list from Alice and did not, for example, steal it from someone else. Therefore, Alice comes up with a very nasty question (S^+_{proxy}) that only she knows the answer to (S^-_{proxy}). Anyone can easily verify the correctness of the answer when given the question. The question is appended to the list before Alice adds her signature.

When delegating some of her rights, Alice gives the signed list of rights, along with the nasty question, to Bob. She also gives Bob the answer ensuring that no one can intercept it. Bob now has a list of rights, signed by Alice, which he can hand over to, say, Charlie, when necessary. Charlie will ask him the nasty question at the bottom of the list. If Bob knows the answer to it, Charlie will know for sure that Alice had indeed delegated the listed rights to Bob.

An important property of this scheme is that Alice need not be consulted. In fact, Bob may decide to pass on (some of) the rights on the list to Dave. In doing so, he will also tell Dave the answer to the question, so that Dave can prove the

list was handed over to him by someone entitled to it. Alice never needs to know about Dave at all.

A protocol for delegating and exercising rights is shown in Fig. 9-39. Assume that Alice and Bob share a secret key $K_{A,B}$ that can be used for encrypting messages they send to each other. Then, Alice first sends Bob the certificate $C = \{R, S^+_{proxy}\}$, signed with $sig(A,C)$ (and denoted again as $[R, S^-_{proxy}]_A$). There is no need to encrypt this message: it can be sent as plaintext. Only the private part of the secret needs to be encrypted, shown as $K_{A,B}(S^-_{proxy})$ in message 1.

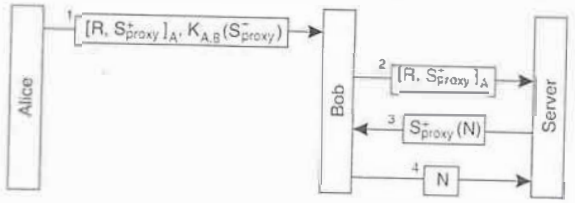

Figure 9-39. Using a proxy to delegate and prove ownership of access rights.

Now suppose that Bob wants an operation to be carried out at an object that resides at a specific server. Also, assume that Alice is authorized to have that operation carried out, and that she has delegated those rights to Bob. Therefore, Bob hands over his credentials to the server in the form of the signed certificate $[R, S^+_{proxy}]_A$.

At that point, the server will be able to verify that C has not been tampered with: any modification to the list of rights, or the nasty question will be noticed, because both have been jointly signed by Alice. However, the server does not know yet whether Bob is the rightful owner of the certificate. To verify this, the server must use the secret that came with C.

There are several ways to implement S^+_{proxy} and S^-_{proxy}. For example, assume S^-_{proxy} is a public key and S^+_{proxy} the corresponding private key. Z can then challenge Bob by sending him a nonce N, encrypted with S^+_{proxy}. By decrypting $S^+_{proxy}(N)$ and returning N, Bob proves he knows the secret and is thus the rightful holder of the certificate. There are other ways to implement secure delegation as well, but the basic idea is always the same: show you know a secret.

9.5 SUMMARY

Security plays an extremely important role in distributed systems. A distributed system should provide the mechanisms that allow a variety of different security policies to be enforced. Developing and properly applying those mechanisms generally makes security a difficult engineering exercise.

Three important issues can be distinguished. The first issue is that a distributed system should offer facilities to establish secure channels between processes. A secure channel, in principle, provides the means to mutually authenticate the communicating parties, and protect messages against tampering during their transmission. A secure channel generally also provides confidentiality so that no one but the communicating parties can read the messages that go through the channel.

An important design issue is whether to use only a symmetric cryptosystem (which is based on shared secret keys), or to combine it with a public-key system. Current practice shows the use of public-key cryptography for distributing short-term shared secret keys. The latter are known as session keys.

The second issue in secure distributed systems is access control, or authorization. Authorization deals with protecting resources in such a way that only processes that have the proper access rights can actual access and use those resources. Access control always take place after a process has been authenticated. Related to access control is preventing denial-of-service, which turns out to a difficult problem for systems that are accessible through the Internet.

There are two ways of implementing access control. First, each resource can maintain an access control list, listing exactly the access rights of each user or process. Alternatively, a process can carry a certificate stating precisely what its rights are for a particular set of resources. The main benefit of using certificates is that a process can easily pass its ticket to another process, that is, delegate its access rights. Certificates, however, have the drawback that they are often difficult to revoke.

Special attention is needed when dealing with access control in the case of mobile code. Besides being able to protect mobile code against a malicious host, it is generally more important to protect a host against malicious mobile code. Several proposals have been made, of which the sandbox is currently the most widely-applied one. However, sandboxes are rather restrictive, and more flexible approaches based on true protection domains have been devised as well.

The third issue in secure distributed systems concerns management. There are essentially two important subtopics: key management and authorization management. Key management includes the distribution of cryptographic keys, for which certificates as issued by trusted third parties play an important role. Important with respect to authorization management are attribute certificates and delegation.

PROBLEMS

1. Which mechanisms could a distributed system provide as security services to application developers that believe only in the end-to-end argument in system's design, as discussed in Chap. 6?

2. In the RISSC approach, can all security be concentrated on secure servers or not?
3. Suppose that you were asked to develop a distributed application that would allow teachers to set up exams. Give at least three statements that would be part of the security policy for such an application.
4. Would it be safe to join message 3 and message 4 in the authentication protocol shown in Fig. 9-12, into $K_{A,B}(R_B, R_A)$?
5. Why is it not necessary in Fig. 9-15 for the KDC to know for sure it was talking to Alice when it receives a request for a secret key that Alice can share with Bob?
6. What is wrong in implementing a nonce as a timestamp?
7. In message 2 of the Needham-Schroeder authentication protocol, the ticket is encrypted with the secret key shared between Alice and the KDC. Is this encryption necessary?
8. Can we safely adapt the authentication protocol shown in Fig. 9-19 such that message 3 consists only of R_B?
9. Devise a simple authentication protocol using signatures in a public-key cryptosystem.
10. Assume Alice wants to send a message m to Bob. Instead of encrypting m with Bob's public key K_B^+, she generates a session key $K_{A,B}$ and then sends $[K_{A,B}(m), K_B^+(K_{A,B})]$. Why is this scheme generally better? (*Hint*: consider performance issues.)
11. What is the role of the timestamp in message 6 in Fig. 9-23, and why does it need to be encrypted?
12. Complete Fig. 9-23 by adding the communication for authentication between Alice and Bob.
13. How can role changes be expressed in an access control matrix?
14. How are ACLs implemented in a UNIX file system?
15. How can an organization enforce the use of a Web proxy gateway and prevent its users to directly access external Web servers?
16. Referring to Fig. 9-31, to what extent does the use of Java object references as capabilities actually depend on the Java language?
17. Name three problems that will be encountered when developers of interfaces to local resources are required to insert calls to enable and disable privileges to protect against unauthorized access by mobile programs as explained in the text.
18. Name a few advantages and disadvantages of using centralized servers for key management.
19. The Diffie-Hellman key-exchange protocol can also be used to establish a shared secret key between three parties. Explain how.
20. There is no authentication in the Diffie-Hellman key-exchange protocol. By exploiting this property, a malicious third party, Chuck, can easily break into the key exchange taking place between Alice and Bob, and subsequently ruin the security. Explain how this would work.

21. Give a straightforward way how capabilities in Amoeba can be revoked.

22. Does it make sense to restrict the lifetime of a session key? If so, give an example how that could be established.

23. (Lab assignment) Install and configure a Kerberos v5 environment for a distributed system consisting of three different machines. One of these machines should be running the KDC. Make sure you can setup a (Kerberos) telnet connection between any two machines, but making use of only a single registered password at the KDC. Many of the details on running Kerberos are explained in Garman (2003).

10

DISTRIBUTED OBJECT-BASED SYSTEMS

With this chapter, we switch from our discussion of principles to an examination of various paradigms that are used to organize distributed systems. The first paradigm consists of distributed objects. In distributed object-based systems, the notion of an object plays a key role in establishing distribution transparency. In principle, everything is treated as an object and clients are offered services and resources in the form of objects that they can invoke.

Distributed objects form an important paradigm because it is relatively easy to hide distribution aspects behind an object's interface. Furthermore, because an object can be virtually anything, it is also a powerful paradigm for building systems. In this chapter, we will take a look at how the principles of distributed systems are applied to a number of well-known object-based systems. In particular, we cover aspects of CORBA, Java-based systems, and Globe.

10.1 ARCHITECTURE

Object orientation forms an important paradigm in software development. Ever since its introduction, it has enjoyed a huge popularity. This popularity stems from the natural ability to build software into well-defined and more or less independent components. Developers could concentrate on implementing specific functionality independent from other developers.

Object orientation began to be used for developing distributed systems in the 1980s. Again, the notion of an independent object hosted by a remote server while attaining a high degree of distribution transparency formed a solid basis for developing a new generation of distributed systems. In this section, we will first take a deeper look into the general architecture of object-based distributed systems, after which we can see how specific principles have been deployed in these systems.

10.1.1 Distributed Objects

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

This separation between interfaces and the objects implementing these interfaces is crucial for distributed systems. A strict separation allows us to place an interface at one machine, while the object itself resides on another machine. This organization, which is shown in Fig. 10-1, is commonly referred to as a **distributed object**.

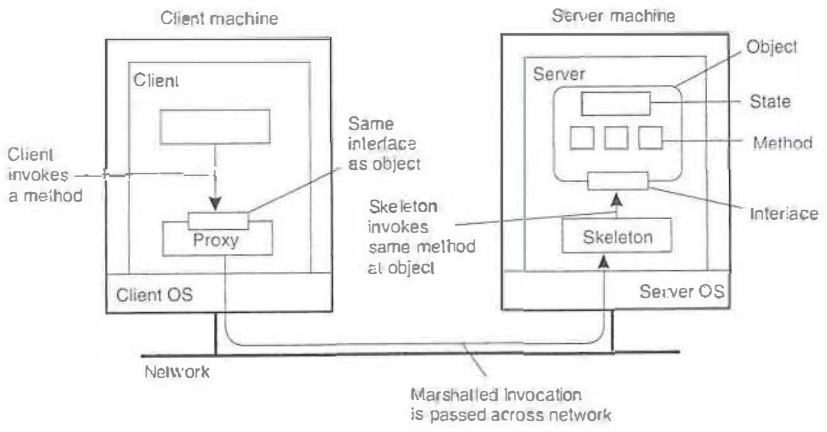

Figure 10-1. Common organization of a remote object with client-side proxy.

When a client binds to a distributed object, an implementation of the object's interface, called a proxy, is then loaded into the client's address space. A proxy is

analogous to a client stub in RPC systems. The only thing it does is marshal method invocations into messages and unmarshal reply messages to return the result of the method invocation to the client. The actual object resides at a server machine, where it offers the same interface as it does on the client machine. Incoming invocation requests are first passed to a server stub, which unmarshals them to make method invocations at the object's interface at the server. The server stub is also responsible for marshaling replies and forwarding reply messages to the client-side proxy.

The server-side stub is often referred to as a **skeleton** as it provides the bare means for letting the server middleware access the user-defined objects. In practice, it often contains incomplete code in the form of a language-specific class that needs to be further specialized by the developer.

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

Compile-Time versus Runtime Objects

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

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

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

When dealing with runtime objects, how objects are actually implemented is basically left open. For example, a developer may choose to write a C library containing a number of functions that can all work on a common data file. The

essence is how to let such an implementation appear to be an object whose methods can be invoked from a remote machine. A common approach is to use an **object adapter**, which acts as a *wrapper* around the implementation with the sole purpose to give it the appearance of an object. The term adapter is derived from a design pattern described in Gamma et al. (1994), which allows an interface to be converted into something that a client expects. An example object adapter is one that dynamically binds to the C library mentioned above and opens an associated data file representing an object's current state.

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

Persistent and Transient Objects

Besides the distinction between language-level objects and runtime objects, there is also a distinction between persistent and transient objects. A **persistent object** is one that continues to exist even if it is currently not contained in the address space of any server process. In other words, a persistent object is not dependent on its current server. In practice, this means that the server that is currently managing the persistent object can store the object's state on secondary storage and then exit. Later, a newly started server can read the object's state from storage into its own address space, and handle invocation requests. In contrast, a **transient object** is an object that exists only as long as the server that is hosting the object. As soon as that server exits, the object ceases to exist as well. There used to be much controversy about having persistent objects; some people believe that transient objects are enough. To take the discussion away from middleware issues, most object-based distributed systems simply support both types.

10.1.2 Example: Enterprise Java Beans

The Java programming language and associated model has formed the foundation for numerous distributed systems and applications. Its popularity can be attributed to the straightforward support for object orientation, combined with the inherent support for remote method invocation. As we will discuss later in this chapter, Java provides a high degree of access transparency, making it easier to use than, for example, the combination of C with remote procedure calling.

Ever since its introduction, there has been a strong incentive to provide facilities that would ease the development of distributed applications. These facilities go well beyond language support, requiring a runtime environment that supports

traditional multitiered client-server architectures. To this end, much work has been put into the development of **(Enterprise) Java Beans (EJB)**.

An EJB is essentially a Java object that is hosted by a special server offering different ways for remote clients to invoke that object. Crucial is that this server provides the support to separate application functionality from systems-oriented functionality. The latter includes functions for looking up objects, storing objects, letting objects be part of a transaction, and so on. How this separation can be realized is discussed below when we concentrate on object servers. How to develop EJBs is described in detail by Monson-Hafael et al. (2004). The specifications can be found in Sun Microsystems (2005a).

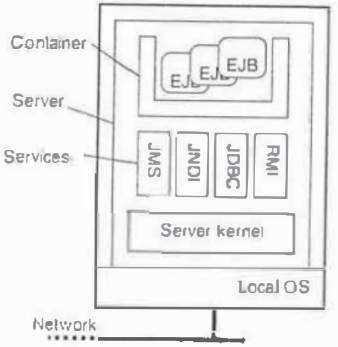

Figure 10-2. General architecture of an EJB server.

With this separation in mind, EJBs can be pictured as shown in Fig. 10-2. The important issue is that an EJB is embedded inside a container which effectively provides interfaces to underlying services that are implemented by the application server. The container can more or less automatically bind the EJB to these services, meaning that the correct references are readily available to a programmer. Typical services include those for remote method invocation (RMI), database access (JDBC), naming (JNDI), and messaging (JMS). Making use of these services is more or less automated, but does require that the programmer makes a distinction between four kinds of EJBs:

1. Stateless session beans
2. Stateful session beans
3. Entity beans
4. Message-driven beans

As its name suggests, a **stateless session bean** is a transient object that is invoked once, does its work, after which it discards any information it needed to perform the service it offered to a client. For example, a stateless session bean could be used to implement a service that lists the top-ranked books. In this case, the bean would typically consist of an SQL query that is submitted to an database. The results would be put into a special format that the client can handle, after which its work would have been completed and the listed books discarded.

In contrast, a **stateful session bean** maintains client-related state. The canonical example is a bean implementing an electronic shopping cart like those widely deployed for electronic commerce. In this case, a client would typically be able to put things in a cart, remove items, and use the cart to go to an electronic checkout. The bean, in turn, would typically access databases for getting current prices and information on number of items still in stock. However, its lifetime would still be limited, which is why it is referred to as a session bean: when the client is finished (possibly having invoked the object several times), the bean will automatically be destroyed.

An **entity bean** can be considered to be a long-lived persistent object. As such, an entity bean will generally be stored in a database, and likewise, will often also be part of distributed transactions. Typically, entity beans store information that may be needed a next time a specific client access the server. In settings for electronic commerce, an entity bean can be used to record customer information, for example, shipping address, billing address, credit card information, and so on. In these cases, when a client logs in, his associated entity bean will be restored and used for further processing.

Finally, **message-driven beans** are used to program objects that should react to incoming messages (and likewise, be able to send messages). Message-driven beans cannot be invoked directly by a client, but rather fit into a *publish-subscribe* way of communication, which we briefly discussed in Chap. 4. What it boils down to is that a message-driven bean is automatically called by the server when a specific message m is received, to which the server (or rather an application it is hosting) had previously subscribed. The bean contains application code for handling the message, after which the server simply discards it. Message-driven beans are thus seen to be stateless. We will return extensively to this type of communication in Chap. 13.

10.1.3 Example: Globe Distributed Shared Objects

Let us now take a look at a completely different type of object-based distributed system. Globe is a system in which scalability plays a central role. All aspects that deal with constructing a large-scale wide-area system that can support huge numbers of users and objects drive the design of Globe. Fundamental to this approach is the way objects are viewed. Like other object-based systems, objects in Globe are expected to encapsulate state and operations on that state.

An important difference with other object-based systems is that objects are also expected to encapsulate the implementation of policies that prescribe the distribution of an object's state across multiple machines. In other words, each object determines how its state will be distributed over its replicas. Each object also controls its own policies in other areas as well.

By and large, objects in Globe are put in charge as much as possible. For example, an object decides how, when, and where its state should be migrated. Also, an object decides if its state is to be replicated, and if so, how replication should take place. In addition, an object may also determine its security policy and implementation. Below, we describe how such encapsulation is achieved.

Object Model

Unlike most other object-based distributed systems, Globe does not adopt the remote-object model. Instead, objects in Globe can be physically distributed, meaning that the state of an object can be distributed and replicated across multiple processes. This organization is shown in Fig. 10-3, which shows an object that is distributed across four processes, each running on a different machine. Objects in Globe are referred to as **distributed shared objects**, to reflect that objects are normally shared between several processes. The object model originates from the distributed objects used in Orca as described in Bal (1989). Similar approaches have been followed for fragmented objects (Makpangou et al., 1994).

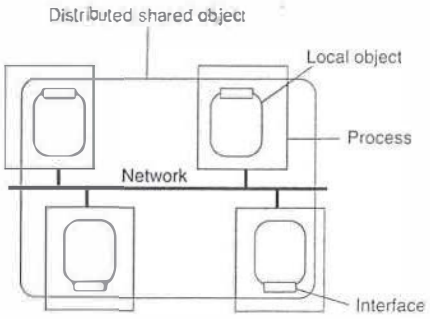

Figure 10-3. The organization of a Globe distributed shared object.

A process that is bound to a distributed shared object is offered a local implementation of the interfaces provided by that object. Such a local implementation is called a **local representative**, or simply **local object**. In principle, whether or not a local object has state is completely transparent to the bound process. All implementation details of an object are hidden behind the interfaces offered to a process. The only thing visible outside the local object are its methods.

Globe local objects come in two flavors. A primitive local object is a local object that does not contain any other local objects. In contrast, a composite local object is an object that is composed of multiple (possibly composite) local objects. Composition is used to construct a local object that is needed for implementing distributed shared objects. This local object is shown in Fig. 10-4 and consists of at least four subobjects.

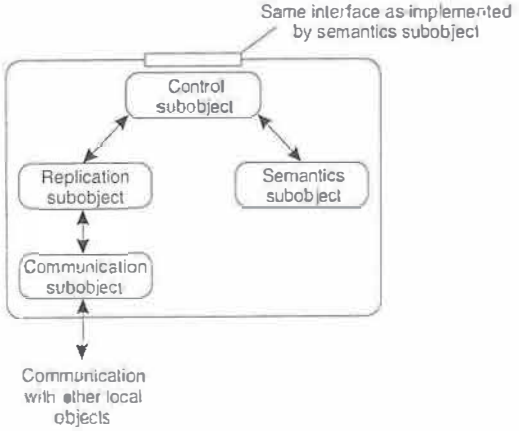

Figure 10-4. The general organization of a local object for distributed shared objects in Globe.

The **semantics subobject** implements the functionality provided by a distributed shared object. In essence, it corresponds to ordinary remote objects, similar in flavor to EJBs.

The **communication subobject** is used to provide a standard interface to the underlying network. This subobject offers a number of message-passing primitives for connection-oriented as well as connectionless communication. There are also more advanced communication subobjects available that implement multicasting interfaces. Communication subobjects can be used that implement reliable communication, while others offer only unreliable communication.

Crucial to virtually all distributed shared objects is the **replication subobject**. This subobject implements the actual distribution strategy for an object. As in the case of the communication subobject, its interface is standardized. The replication subobject is responsible for deciding exactly when a method as provided by the semantics subobject is to be carried out. For example, a replication subobject that implements active replication will ensure that all method invocations are carried out in the same order at each replica. In this case, the subobject will have to communicate with the replication subobjects in other local objects that comprise the distributed shared object.

The **control subobject** is used as an intermediate between the user-defined interfaces of the semantics subobject and the standardized interfaces of the replication subobject. In addition, it is responsible for exporting the interfaces of the semantics subobject to the process bound to the distributed shared object. All method invocations requested by that process are marshaled by the control subobject and passed to the replication subobject.

The replication subobject will eventually allow the control subobject to carry on with an invocation request and to return the results to the process. Likewise, invocation requests from remote processes are eventually passed to the control subobject as well. Such a request is then unmarshaled, after which the invocation is carried out by the control subobject, passing results back to the replication subobject.

10.2 PROCESSES

A key role in object-based distributed systems is played by object servers, that is, the server designed to host distributed objects. In the following, we first concentrate on general aspects of object servers, after which we will discuss the open-source JBoss server.

10.2.1 Object Servers

An object server is a server tailored to support distributed objects. The important difference between a general object server and other (more traditional) servers is that an object server by itself does not provide a specific service. Specific services are implemented by the objects that reside in the server. Essentially, the server provides only the means to invoke local objects, based on requests from remote clients. As a consequence, it is relatively easy to change services by simply adding and removing objects.

An object server thus acts as a place where objects live. An object consists of two parts: data representing its state and the code for executing its methods. Whether or not these parts are separated, or whether method implementations are shared by multiple objects, depends on the object server. Also, there are differences in the way an object server invokes its objects. For example, in a multithreaded server, each object may be assigned a separate thread, or a separate thread may be used for each invocation request. These and other issues are discussed next.

Alternatives for Invoking Objects

For an object to be invoked, the object server needs to know which code to execute, on which data it should operate, whether it should start a separate thread to take care of the invocation, and so on. A simple approach is to assume that all

objects look alike and that there is only one way to invoke an object. Unfortunately, such an approach is generally inflexible and often unnecessarily constrains developers of distributed objects.

A much better approach is for a server to support different policies. Consider, for example, transient objects. Recall that a transient object is an object that exists only as long as its server exists, but possibly for a shorter period of time. An in-memory, read-only copy of a file could typically be implemented as a transient object. Likewise, a calculator could also be implemented as a transient object. A reasonable policy is to create a transient object at the first invocation request and to destroy it as soon as no clients are bound to it anymore.

The advantage of this approach is that a transient object will need a server's resources only as long as the object is really needed. The drawback is that an invocation may take some time to complete, because the object needs to be created first. Therefore, an alternative policy is sometimes to create all transient objects at the time the server is initialized, at the cost of consuming resources even when no client is making use of the object.

In a similar fashion, a server could follow the policy that each of its objects is placed in a memory segment of its own. In other words, objects share neither code nor data. Such a policy may be necessary when an object implementation does not separate code and data, or when objects need to be separated for security reasons. In the latter case, the server will need to provide special measures, or require support from the underlying operating system, to ensure that segment boundaries are not violated.

The alternative approach is to let objects at least share their code. For example, a database containing objects that belong to the same class can be efficiently implemented by loading the class implementation only once into the server. When a request for an object invocation comes in, the server need only fetch that object's state from the database and execute the requested method.

Likewise, there are many different policies with respect to threading. The simplest approach is to implement the server with only a single thread of control. Alternatively, the server may have several threads, one for each of its objects. Whenever an invocation request comes in for an object, the server passes the request to the thread responsible for that object. If the thread is currently busy, the request is temporarily queued.

The advantage of this approach is that objects are automatically protected against concurrent access: all invocations are serialized through the single thread associated with the object. Neat and simple. Of course, it is also possible to use a separate thread for each invocation request, requiring that objects should have already been protected against concurrent access. Independent of using a thread per object or thread per method is the choice of whether threads are created on demand or the server maintains a pool of threads. Generally there is no single best policy. Which one to use depends on whether threads are available, how much performance matters, and similar factors.

Object Adapter

Decisions on how to invoke an object are commonly referred to as **activation policies**, to emphasize that in many cases the object itself must first be brought into the server's address space (i.e., activated) before it can actually be invoked. What is needed then is a mechanism to group objects per policy. Such a mechanism is sometimes called an **object adapter**, or alternatively an **object wrapper**. An object adapter can best be thought of as software implementing a specific activation policy. The main issue, however, is that object adapters come as generic components to assist developers of distributed objects, and which need only to be configured for a specific policy.

An object adapter has one or more objects under its control. Because a server should be capable of simultaneously supporting objects that require different activation policies, several object adapters may reside in the same server at the same time. When an invocation request is delivered to the server, that request is first dispatched to the appropriate object adapter, as shown in Fig. 10-5.

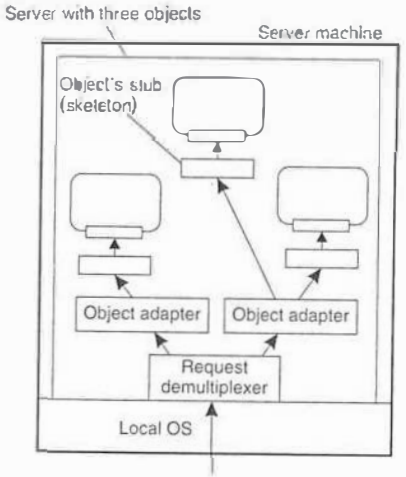

Figure 10-5. Organization of an object server supporting different activation policies.

An important observation is that object adapters are unaware of the specific interfaces of the objects they control. Otherwise, they could never be generic. The only issue that is important to an object adapter is that it can extract an object reference from an invocation request, and subsequently dispatch the request to the referenced object, but now following a specific activation policy. As is also illustrated in Fig. 10-5, rather than passing the request directly to the object, an

adapter hands an invocation request to the server-side stub of that object. The stub, also called a skeleton, is normally generated from the interface definitions of the object, unmarshals the request and invokes the appropriate method.

An object adapter can support different activation policies by simply configuring it at runtime. For example, in CORBA-compliant systems (OMG, 2004a), it is possible to specify whether an object should continue to exist after its associated adapter has stopped. Likewise, an adapter can be configured to generate object identifiers, or to let the application provide one. As a final example, an adapter can be configured to operate in single-threaded or multithreaded mode as we explained above.

As a side remark, note that although in Fig. 10-5 we have spoken about objects, we have said nothing about what these objects actually are. In particular, it should be stressed that as part of the implementation of such an object the server may (indirectly) access databases or call special library routines. The implementation details are hidden for the object adapter who communicates only with a skeleton. As such, the actual implementation may have nothing to do with what we often see with language-level (i.e., compile-time) objects. For this reason, a different terminology is generally adopted. A servant is the general term for a piece of code that forms the implementation of an object. In this light, a Java bean can be seen as nothing but just another kind of servant.

10.2.2 Example: The Ice Runtime System

Let us take a look at how distributed objects are handled in practice. We briefly consider the Ice distributed-object system, which has been partly developed in response to the intricacies of commercial object-based distributed systems (Henning, 2004). In this section, we concentrate on the core of an Ice object server and defer other parts of the system to later sections.

An object server in Ice is nothing but an ordinary process that simply starts with initializing the Ice runtime system (RTS). The basis of the runtime environment is formed by what is called a *communicator*. A communicator is a component that manages a number of basic resources, of which the most important one is formed by a pool of threads. Likewise, it will have associated dynamically allocated memory, and so on. In addition, a communicator provides the means for configuring the environment. For example, it is possible to specify maximum message lengths, maximum invocation retries, and so on.

Normally, an object server would have only a single communicator. However, when different applications need to be fully separated and protected from each other, a separate communicator (with possibly a different configuration) can be created within the same process. At the very least, such an approach would separate the different thread pools so that if one application has consumed all its threads, then this would not affect the other application.

A communicator can also be used to create an object adapter, such as shown in Fig. 10-6. We note that the code is simplified and incomplete. More examples and detailed information on Ice can be found in Henning and Spruiell (2005).

```
main(int argc, char* argv[]) {
    Ice::Communicator      ic;
    Ice::ObjectAdapter     adapter;
    Ice::Object            object;

    ic = Ice::initialize(argc, argv);
    adapter =
        ic->createObjectAdapterWithEnd Points( "MyAdapter","tcp -p 10000");
    object = new MyObject;
    adapter->add(object, objectID);
    adapter->activate();
    ic->waitForShutdown();
}
```

Figure 10-6. Example of creating an object server in Ice.

In this example, we start with creating and initializing the runtime environment. When that is done, an object adapter is created. In this case, it is instructed to listen for incoming TCP connections on port 10000. Note that the adapter is created in the context of the just created communicator. We are now in the position to create an object and to subsequently add that object to the adapter. Finally, the adapter is *activated*, meaning that, under the hood, a thread is activated that will start listening for incoming requests.

This code does not yet show much differentiation in activation policies. Policies can be changed by modifying the *properties* of an adapter. One family of properties is related to maintaining an adapter-specific set of threads that are used for handling incoming requests. For example, one can specify that there should always be only one thread, effectively serializing all accesses to objects that have been added to the adapter.

Again, note that we have not specified MyObject. Like before, this could be a simple C++ object, but also one that accesses databases and other external services that jointly implement an object. By registering MyObject with an adapter, such implemenation details are completely hidden from clients, who now believe that they are invoking a remote object.

In the example above, an object is created as part of the application, after which it is added to an adapter. Effectively, this means that an adapter may need to support many objects at the same time, leading to potential scalability problems. An alternative solution is to dynamically load objects into memory when they are needed. To do this, Ice provides support for special objects known as *locators*. A locator is called when the adapter receives an incoming request for an

object that has not been explicitly added. In that case, the request is forwarded to the locator, whose job is to further handle the request.

To make matters more concrete, suppose a locator is handed a request for an object of which the locator knows that its state is stored in a relational database system. Of course, there is no magic here: the locator has been programmed explicitly to handle such requests. In this case, the object's identifier may correspond to the key of a record in which that state is stored. The locator will then simply do a lookup on that key, fetch the state, and will then be able to further process the request.

There can be more than one locator added to an adapter. In that case, the adapter would keep track of which object identifiers would belong to the same locator. Using multiple locators allows supporting many objects by a single adapter. Of course, objects (or rather their state) would need to be loaded at runtime, but this dynamic behavior would possibly make the server itself relatively simple.

10.3 COMMUNICATION

We now draw our attention to the way communication is handled in object-based distributed systems. Not surprisingly, these systems generally offer the means for a remote client to invoke an object. This mechanism is largely based on remote procedure calls (RPCs), which we discussed extensively in Chap. 4. However, before this can happen, there are numerous issues that need to be dealt with.

10.3.1 Binding a Client to an Object

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

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

With implicit binding, the client is offered a simple mechanism that allows it to directly invoke methods using only a reference to an object. For example, C++

allows overloading the unary member selection operator ("→") permitting us to introduce object references as if they were ordinary pointers as shown in Fig. 10-7(a). With implicit binding, the client is transparently bound to the object at the moment the reference is resolved to the actual object. In contrast, with explicit binding, the client should first call a special function to bind to the object before it can actually invoke its methods. Explicit binding generally returns a pointer to a proxy that is then become locally available, as shown in Fig. 10-7(b).

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

(a)

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

(b)

Figure 10-7. (a) An example with implicit binding using only global references. (b) An example with explicit binding using global and local references.

Implementation of Object References

It is clear that an object reference must contain enough information to allow a client to bind to an object. A simple object reference would include the network address of the machine where the actual object resides, along with an end point identifying the server that manages the object, plus an indication of which object. Note that part of this information will be provided by an object adapter. However, there are a number of drawbacks to this scheme.

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

However, encoding the network address of the server's machine into an object reference is not always a good idea. The problem with this approach is that the server can never move to another machine without invalidating all the references to the objects it manages. An obvious solution is to expand the idea of a local

daemon maintaining an end point table to a location server that keeps track of the machine where an object's server is currently running. An object reference would then contain the network address of the location server, along with a systemwide identifier for the server. Note that this solution comes close to implementing flat name spaces as we discussed in Chap. 5.

What we have tacitly assumed so far is that the client and server have somehow already been configured to use the same protocol stack. Not only does this mean that they use the same transport protocol, for example, TCP; furthermore it means that they use the same protocol for marshaling and unmarshaling parameters. They must also use the same protocol for setting up an initial connection, handle errors and flow control the same way, and so on.

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

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

10.3.2 Static versus Dynamic Remote Method Invocations

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

The usual way to provide RMI support is to specify the object's interfaces in an interface definition language, similar to the approach followed with RPCs.

Alternatively, we can make use of an object-based language such as Java, that will handle stub generation automatically. This approach of using predefined interface definitions is generally referred to as **static invocation**. Static invocations require that the interfaces of an object are known when the client application is being developed. It also implies that if interfaces change, then the client application must be recompiled before it can make use of the new interfaces.

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

```
invoke(object, method, input_parameters, output_parameters);
```

where *object* identifies the distributed object, *method* is a parameter specifying exactly which method should be invoked, *input_parameters* is a data structure that holds the values of that method's input parameters, and *output_parameters* refers to a data structure where output values can be stored.

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

```
fobject.append(int)
```

whereas the dynamic invocation would look something like

```
invoke(fobject, id(append), int)
```

where the operation *id(append)* returns an identifier for the method append.

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

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

10.3.3 Parameter Passing

Because most RMI systems support systemwide object references, passing parameters in method invocations is generally less restricted than in the case of RPCs. However, there are some subtleties that can make RMIs trickier than one might initially expect, as we briefly discuss in the following pages.

Let us first consider the situation that there are only distributed objects. In other words, all objects in the system can be accessed from remote machines. In that case, we can consistently use object references as parameters in method invocations. References are passed by value, and thus copied from one machine to the other. When a process is given an object reference as the result of a method invocation, it can simply bind to the object referred to when needed later.

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

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

These two situations are illustrated in Fig. 10-8, which shows a client program running on machine *A*, and a server program on machine *C*. The client has a reference to a local object *O*1 that it uses as a parameter when calling the server program on machine *C*. In addition, it holds a reference to a remote object *O*2 residing at machine *B*, which is also used as a parameter. When calling the server, a copy of *O*1 is passed to the server on machine *C*, along with only a copy of the reference to *O*2.

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

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

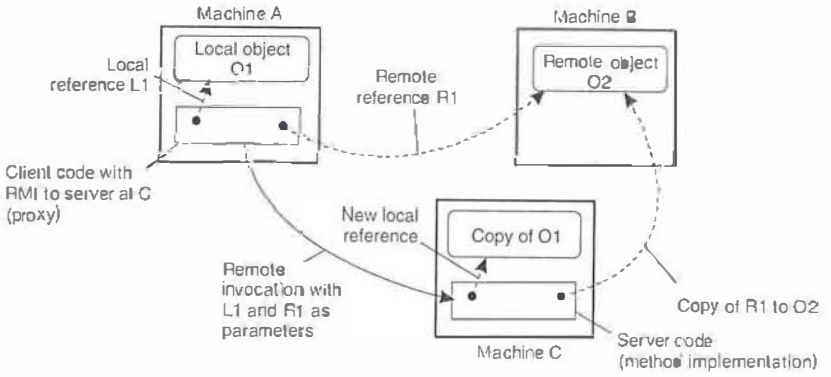

Figure 10-8. The situation when passing an object by reference or by value.

10.3.4 Example: Java RMI

In Java, distributed objects have been integrated into the language. An important goal was to keep as much of the semantics of nondistributed objects as possible. In other words, the Java language developers have aimed for a high degree of distribution transparency. However, as we shall see, Java's developers have also decided to make distribution apparent where a high degree of transparency was simply too inefficient, difficult, or impossible to realize.

The Java Distributed-Object Model

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

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

Proxies of the actual object are thus not cloned. If a client at a remote machine wants access to the cloned object at the server, it will first have to bind to that object again.

Java Remote Object Invocation

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

The only distinction made between local and remote objects during an RMI is that local objects are passed by value (including large objects such as arrays), whereas remote objects are passed by reference. In other words, a local object is first copied after which the copy is used as parameter value. For a remote object, a reference to the object is passed as parameter instead of a copy of the object, as was also shown in Fig. 10-8.

In Java RMI, a reference to a remote object is essentially implemented as we explained in Sec. 10.3.3. Such a reference consists of the network address and end point of the server, as well as a local identifier for the actual object in the server's address space. That local identifier is used only by the server. As we also explained, a reference to a remote object also needs to encode the protocol stack that is used by a client and the server to communicate. To understand how such a stack is encoded in the case of Java RMI, it is important to realize that each object in Java is an instance of a class. A class, in turn, contains an implementation of one or more interfaces.

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

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

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

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

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

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

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

10.3.5 Object-Based Messaging

Although RMI is the preferred way of handling communication in object-based distributed systems, messaging has also found its way as an important alternative. There are various object-based messaging systems available, and, as can be expected, offer very much the same functionality. In this section we will take a closer look at CORBA messaging, partly because it also provides an interesting way of combining method invocation and message-oriented communication.

CORBA is a well-known specification for distributed systems. Over the years, several implementations have come to existence, although it remains to be seen to what extent CORBA itself will ever become truly popular. However, independent of popularity, the CORBA specifications are comprehensive (which to many also means they are very complex). Recognizing the popularity of messaging systems, CORBA was quick to include a specification of a messaging service.

What makes messaging in CORBA different from other systems is its inherent object-based approach to communication. In particular, the designers of the messaging service needed to retain the model that all communication takes place by invoking an object. In the case of messaging, this design constraint resulted in two forms of asynchronous method invocations (in addition to other forms that were provided by CORBA as well).

An **asynchronous method invocation** is analogous to an asynchronous RPC: the caller continues after initiating the invocation without waiting for a result. In CORBA's **callback model**, a client provides an object that implements an interface containing callback methods. These methods can be called by the underlying communication system to pass the result of an asynchronous invocation. An important design issue is that asynchronous method invocations do not affect the original implementation of an object. In other words, it is the client's responsibility to transform the original synchronous invocation into an asynchronous one: the server is presented with a normal (synchronous) invocation request.

Constructing an asynchronous invocation is done in two steps. First, the original interface as implemented by the object is replaced by two new interfaces that are to be implemented by client-side software only. One interface contains the specification of methods that the client can call. None of these methods returns a value or has any output parameter. The second interface is the callback interface. For each operation in the original interface, it contains a method that will be called by the client's runtime system to pass the results of the associated method as called by the client.

As an example, consider an object implementing a simple interface with just one method:

```
int add(in int i, in int j, out int k);
```

Assume that this method takes two nonnegative integers i and j and returns $i + j$ as output parameter k. The operation is assumed to return -1 if the operation did not complete successfully. Transforming the original (synchronous) method invocation into an asynchronous one with callbacks is achieved by first generating the following pair of method specifications (for our purposes, we choose convenient names instead of following the strict rules as specified in OMG (2004a)):

```
void sendcb_add(in int i, in int j);          // Downcall by the client
void replycb_add(in int ret_val, in int k);   // Upcall to the client
```

In effect, all output parameters from the original method specification are removed from the method that is to be called by the client, and returned as input parameters of the callback operations. Likewise, if the original method specified a return value, that value is passed as an input parameter to the callback operation.

The second step consists of compiling the generated interfaces. As a result, the client is offered a stub that allows it to asynchronously invoke sendcb_add. However, the client will need to provide an implementation for the callback interface, in our example containing the method replycb_add. This last method is called by the client's local runtime system (RTS), resulting in an upcall to the client application. Note that these changes do not affect the server-side implementation of the object. Using this example, the callback model is summarized in Fig. 10-9.

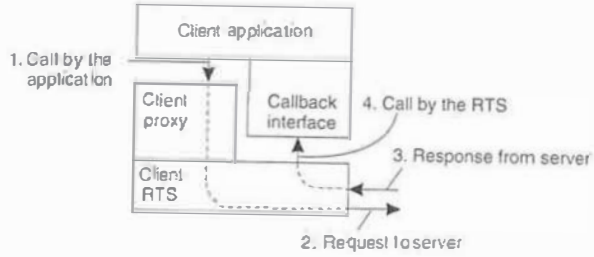

Figure 10-9. CORBA's callback model for asynchronous method invocation.

As an alternative to callbacks, CORBA provides a **polling model**. In this model, the client is offered a collection of operations to poll its local RTS for incoming results. As in the callback model, the client is responsible for transforming the original synchronous method invocations into asynchronous ones. Again, most of the work can be done by automatically deriving the appropriate method specifications from the original interface as implemented by the object.

Returning to our example, the method add will lead to the following two generated method specifications (again, we conveniently adopt our own naming conventions):

```
void sendpoll_add(in int i, in int j);          // Called by the client
void replypoll_add(out int ret_val, out int k); // Also called by the client
```

The most important difference between the polling and callback models is that the method replypoll_add will have to be implemented by the client's RTS. This implementation can be automatically generated from interface specifications, just as the client-side stub is automatically generated as we explained for RPCs. The polling model is summarized in Fig. 10-10. Again, notice that the implementation of the object as it appears at the server's side does not have to be changed.

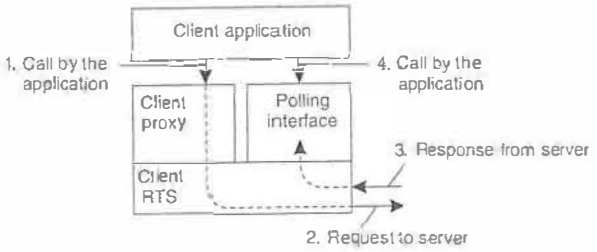

Figure 10-10. CORBA's polling model for asynchronous method invocation.

What is missing from the models described so far is that the messages sent between a client and a server, including the response to an asynchronous invocation, are stored by the underlying system in case the client or server is not yet running. Fortunately, most of the issues concerning such persistent communication do not affect the asynchronous invocation model discussed so far. What is needed is to set up a collection of message servers that will allow messages (be they invocation requests or responses), to be temporarily stored until their delivery can take place.

To this end, the CORBA specifications also include interface definitions for what are called **routers**, which are analogous to the message routers we discussed in Chap. 4, and which can be implemented, for example, using IBM's WebSphere queue managers.

Likewise, Java has its own **Java Messaging Service** (JMS) which is again very similar to what we have discussed before [see Sun Microsystems (2004a)]. We will return to messaging more extensively in Chap. 13 when we discuss the publish/subscribe paradigm.

10.4 NAMING

The interesting aspect of naming in object-based distributed systems evolves around the way that object references are supported. We already described these object references in the case of Java, where they effectively correspond to portable proxy implementations. However, this a language-dependent way of being able to refer to remote objects. Again taking CORBA as an example, let us see how basic naming can also be provided in a language and platform-independent way. We also discuss a completely different scheme, which is used in the Globe distributed system.

10.4.1 CORBA Object References

Fundamental to CORBA is the way its objects are referenced. When a client holds an object reference, it can invoke the methods implemented by the referenced object. It is important to distinguish the object reference that a client process uses to invoke a method, and the one implemented by the underlying RTS.

A process (be it client or server) can use only a language-specific implementation of an object reference. In most cases, this takes the form of a pointer to a local representation of the object. That reference cannot be passed from process *A* to process *B*, as it has meaning only within the address space of process *A*. Instead, process *A* will first have to marshal the pointer into a process-independent representation. The operation to do so is provided by its RTS. Once marshaled, the reference can be passed to process *B*, which can unmarshal it again. Note that processes *A* and *B* may be executing programs written in different languages.

In contrast, the underlying RTS will have its own language-independent representation of an object reference. This representation may even differ from the marshaled version it hands over to processes that want to exchange a reference. The important thing is that when a process refers to an object, its underlying RTS is implicitly passed enough information to know which object is actually being referenced. Such information is normally passed by the client and server-side stubs that are generated from the interface specifications of an object.

One of the problems that early versions of CORBA had was that each implementation could decide on how it represented an object reference. Consequently, if process *A* wanted to pass a reference to process *B* as described above, this would generally succeed only if both processes were using the same CORBA implementation. Otherwise, the marshaled version of the reference held by process *A* would be meaningless to the RTS used by process *B*.

Current CORBA systems all support the same language-independent representation of an object reference, which is called an **Interoperable Object Reference** or **IOR**. Whether or not a CORBA implementation uses IORs internally is not all that important. However, when passing an object reference between two different CORBA systems, it is passed as an IOR. An IOR contains all the information needed to identify an object. The general layout of an IOR is shown in Fig. 10-11, along with specific information for the communication protocol used in CORBA.

Each IOR starts with a repository identifier. This identifier is assigned to an interface so that it can be stored and looked up in an interface repository. It is used to retrieve information on an interface at runtime, and can assist in, for example,

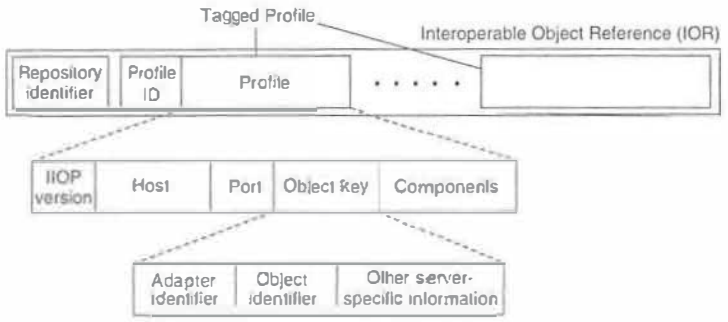

Figure 10-11. The organization of an IOR with specific information for IIOP.

type checking or dynamically constructing an invocation. Note that if this identifier is to be useful, both the client and server must have access to the same interface repository, or at least use the same identifier to identify interfaces.

The most important part of each IOR is formed by what are called **tagged profiles**. Each such profile contains the complete information to invoke an object. If the object server supports several protocols, information on each protocol can be included in a separate tagged profile. CORBA used the **Internet Inter-ORB Protocol (IIOP)** for communication between nodes. (An ORB or **Object Request Broker** is the name used by CORBA for their object-based runtime system.) IIOP is essentially a dedicated protocol for supported remote method invocations. Details on the profile used for IIOP are also shown in Fig. 10-11.

The IIOP profile is identified by a *ProfileID* field in the tagged profile. Its body consists of five fields. The *IIOP version* field identifies the version of IIOP that is used in this profile.

The *Host* field is a string identifying exactly on which host the object is located. The host can be specified either by means of a complete DNS domain name (such as *soling.cs.vu.nl*), or by using the string representation of that host's IP address, such as *130.37.24.11*.

The *Port* field contains the port number to which the object's server is listening for incoming requests.

The *Object key* field contains server-specific information for demultiplexing incoming requests to the appropriate object. For example, an object identifier generated by a CORBA object adapter will generally be part of such an object key. Also, this key will identify the specific adapter.

Finally, there is a *Components* field that optionally contains more information needed for properly invoking the referenced object. For example, this field may contain security information indicating how the reference should be handled, or what to do in the case the referenced server is (temporarily) unavailable.

10.4.2 Globe Object References

Let us now take a look at a different way of referencing objects. In Globe, each distributed shared object is assigned a globally unique object identifier (OID), which is a 256-bit string. A Globe OID is a true identifier as defined in Chap. 5. In other words, a Globe OID refers to at most one distributed shared object; it is never reused for another object; and each object has at most one OID.

Globe OIDs can be used only for comparing object references. For example, suppose processes *A* and *B* are each bound to a distributed shared object. Each process can request the OID of the object they are bound to. If and only if the two OIDs are the same, then *A* and *B* are considered to be bound to the same object.

Unlike CORBA references, Globe OIDs cannot be used to directly contact an object. Instead, to locate an object, it is necessary to look up a contact address for that object in a location service. This service returns a **contact address**, which is comparable to the location-dependent object references as used in CORBA and other distributed systems. Although Globe uses its own specific location service, in principle any of the location services discussed in Chap. 5 would do.

Ignoring some minor details, a contact address has two parts. The first one is an **address identifier** by which the location service can identify the proper leaf node to which insert or delete operations for the associated contact address are to forwarded. Recall that because contact addresses are location dependent, it is important to insert and delete them starting at an appropriate leaf node.

The second part consists of actual address information, but this information is completely opaque to the location service. To the location service, an address is just an array of bytes that can equally stand for an actual network address, a marshaled interface pointer, or even a complete marshaled proxy.

Two kinds of addresses are currently supported in Globe. A **stacked address** represents a layered protocol suite, where each layer is represented by the three-field record shown in Fig. 10-12.

Field	Description
Protocol identifier	A constant representing a (known) protocol
Protocol address	A protocol-specific address
Implementation handle	Reference to a file in a class repository

Figure 10-12. The representation of a protocol layer in a stacked contact address.

The *Protocol identifier* is a constant representing a known protocol. Typical protocol identifiers include *TCP*, *UDP*, and *IP*. The *Protocol address* field contains a protocol-specific address, such as TCP port number, or an IPv4 network address. Finally, an *implementation handle* can be optionally provided to indicate

where a default implementation for the protocol can be found. Typically, an implementation handle is represented as a URL.

The second type of contact address is an instance address, which consists of the two fields shown in Fig. 10-13. Again, the address contains an *implementation handle*, which is nothing but a reference to a file in a class repository where an implementation of a local object can be found. That local object should be loaded by the process that is currently binding to the object.

Field	Description
Implementation handle	Reference to a file in a class repository
Initialization string	String that is used to initialize an implementation

Figure 10-13. The representation of an instance contact address.

Loading follows a standard protocol, similar to class loading in Java. After the implementation has been loaded and the local object created, initialization takes place by passing the *initialization string* to the object. At that point, the object identifier has been completed resolved.

Note the difference in object referencing between CORBA and Globe, a difference which occurs frequently in distributed object-based systems. Where CORBA references contain exact information where to contact an object, Globe references require an additional lookup step to retrieve that information. This distinction also appears in systems such as Ice, where the CORBA equivalent is referred to as a *direct* reference, and the Globe equivalent as an *indirect* reference (Henning and Spruiell, 2005).

10.5 SYNCHRONIZATION

There are only a few issues regarding synchronization in distributed systems that are specific to dealing with distributed objects. In particular, the fact that implementation details are hidden behind interfaces may cause problems: when a process invokes a (remote) object, it has no knowledge whether that invocation will lead to invoking other objects. As a consequence, if an object is protected against concurrent accesses, we may have a cascading set of locks that the invoking process is unaware of, as sketched in Fig. 10-14(a).

In contrast, when dealing with data resources such as files or database tables that are protected by locks, the pattern for the control flow is actually visible to the process using those resources, as shown in Fig. 10-14(b). As a consequence, the process can also exert more control at runtime when things go wrong, such as giving up locks when it believes a deadlock has occurred. Note that transaction processing systems generally follow the pattern shown in Fig. 10-14(b).

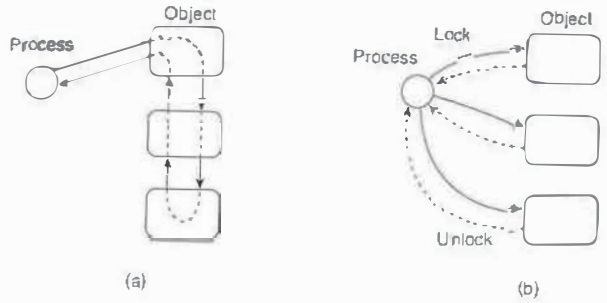

Figure 10-14. Differences in control flow for locking objects

In object-based distributed systems it is therefore important to know where and when synchronization takes place. An obvious location for synchronization is at the object server. If multiple invocation requests for the same object arrive, the server can decide to serialize those requests (and possibly keep a lock on an object when it needs to do a remote invocation itself).

However, letting the object server maintain locks complicates matters in the case that invoking clients crash. For this reason, locking can also be done at the client side, an approach that has been adopted in Java. Unfortunately, this scheme has its own drawbacks.

As we mentioned before, the difference between local and remote objects in Java is often difficult to make. Matters become more complicated when objects are protected by declaring its methods to be synchronized. If two processes simultaneously call a synchronized method, only one of the processes will proceed while the other will be blocked. In this way, we can ensure that access to an object's internal data is completely serialized. A process can also be blocked inside an object, waiting for some condition to become true.

Logically, blocking in a remote object is simple. Suppose that client A calls a synchronized method of a remote object. To make access to remote objects look always *exactly* the same as to local objects, it would be necessary to block A in the client-side stub that implements the object's interface and to which A has direct access. Likewise, another client on a different machine would need to be blocked locally as well before its request can be sent to the server. The consequence is that we need to synchronize different clients at different machines. As we discussed in Chap. 6, distributed synchronization can be fairly complex.

An alternative approach would be to allow blocking only at the server. In principle, this works fine, but problems arise when a client crashes while its invocation is being handled by the server. As we discussed in Chap. 8, we may require relatively sophisticated protocols to handle this situation, and which that may significantly affect the overall performance of remote method invocations.

Therefore, the designers of Java RMI have chosen to restrict blocking on remote objects only to the proxies (Wollrath et al., 1996). This means that threads in the same process will be prevented from concurrently accessing the same remote object, but threads in different processes will not. Obviously, these synchronization semantics are tricky: at the syntactic level (i.e., when reading source code) we may see a nice, clean design. Only when the distributed application is actually executed, unanticipated behavior may be observed that should have been dealt with at design time. Here we see a clear example where striving for distribution transparency is *not* the way to go.

10.6 CONSISTENCY AND REPLICATION

Many object-based distributed systems follow a traditional approach toward replicated objects, effectively treating them as containers of data with their own special operations. As a result, when we consider how replication is handled in systems supporting Java beans, or CORBA-compliant distributed systems, there is not really that much new to report other than what we have discussed in Chap. 7.

For this reason, we focus on a few particular topics regarding consistency and replication that are more profound in object-based distributed systems than others. We will first consider consistency and move to replicated invocations.

10.6.1 Entry Consistency

As we mentioned in Chap. 7, data-centric consistency for distributed objects comes naturally in the form of entry consistency. Recall that in this case, the goal is to group operations on shared data using synchronization variables (e.g., in the form of locks). As objects naturally combine data and the operations on that data, locking objects during an invocation serializes access and keeps them consistent.

Although conceptually associating a lock with an object is simple, it does not necessarily provide a proper solution when an object is replicated. There are two issues that need to be solved for implementing entry consistency. The first one is that we need a means to prevent concurrent execution of multiple invocations on the same object. In other words, when any method of an object is being executed, no other methods may be executed. This requirement ensures that access to the internal data of an object is indeed serialized. Simply using local locking mechanisms will ensure this serialization.

The second issue is that in the case of a replicated object, we need to ensure that all changes to the replicated state of the object are the same. In other words, we need to make sure that no two independent method invocations take place on different replicas at the same time. This requirement implies that we need to order invocations such that each replica sees all invocations in the same order. This

requirement can generally be met in one of two ways: (1) using a primary-based approach or (2) using totally-ordered multicast to the replicas.

In many cases, designing replicated objects is done by first designing a single object, possibly protecting it against concurrent access through local locking, and subsequently replicating it. If we were to use a primary-based scheme, then additional effort from the application developer is needed to serialize object invocations. Therefore, it is often convenient to assume that the underlying middleware supports totally-ordered multicasting, as this would not require any changes at the clients, nor would it require additional programming effort from application developers. Of course, how the totally ordered multicasting is realized by the middleware should be transparent. For all the application may know, its implementation may use a primary-based scheme, but it could equally well be based on Lamport clocks.

However, even if the underlying middleware provides totally-ordered multicasting, more may be needed to guarantee orderly object invocation. The problem is one of granularity: although all replicas of an object server may receive invocation requests in the same order, we need to ensure that all threads in those servers process those requests in the correct order as well. The problem is sketched in Fig. 10-15.

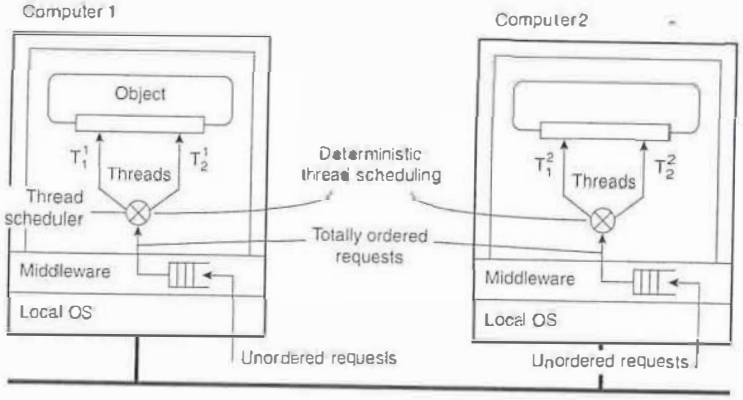

Figure 10-15. Deterministic thread scheduling for replicated object servers.

Multithreaded (object) servers simply pick up an incoming request, pass it on to an available thread, and wait for the next request to come in. The server's thread scheduler subsequently allocates the CPU to runnable threads. Of course, if the middleware has done its best to provide a total ordering for request delivery, the thread schedulers should operate in a deterministic fashion in order not to mix the ordering of method invocations on the same object. In other words, if threads

T_1^1 and T_1^2 from Fig. 10-15 handle the same incoming (replicated) invocation request, they should both be scheduled before T_2^1 and T_2^2, respectively.

Of course, simply scheduling *all* threads deterministically is not necessary. In principle, if we already have totally-ordered request delivery, we need only to ensure that all requests for the same replicated object are handled in the order they were delivered. Such an approach would allow invocations for different objects to be processed concurrently, and without further restrictions from the thread scheduler. Unfortunately, only few systems exist that support such concurrency.

One approach, described in Basile et al. (2002), ensures that threads sharing the same (local) lock are scheduled in the same order on every replica. At the basics lies a primary-based scheme in which one of the replica servers takes the lead in determining, for a specific lock, which thread goes first. An improvement that avoids frequent communication between servers is described in Basile et al. (2003). Note that threads that do not share a lock can thus operate concurrently on each server.

One drawback of this scheme is that it operates at the level of the underlying operating system, meaning that every lock needs to be managed. By providing application-level information, a huge improvement in performance can be made by identifying only those locks that are needed for serializing access to replicated objects (Taiani et al., 2005). We return to these issues when we discuss fault tolerance for Java.

Replication Frameworks

An interesting aspect of most distributed object-based systems is that by nature of the object technology it is often possible to make a clean separation between devising functionality and handling extra-functional issues such as replication. As we explained in Chap. 2, a powerful mechanism to accomplish this separation is formed by interceptors.

Babaoglu et al. (2004) describe a framework in which they use interceptors to replicate Java beans for J2EE servers. The idea is relatively simple: invocations to objects are intercepted at three different points, as also shown in Fig. 10-16:

1. At the client side just before the invocation is passed to the stub.
2. Inside the client's stub, where the interception forms part of the replication algorithm.
3. At the server side, just before the object is about to be invoked.

The first interception is needed when it turns out that the caller is replicated. In that case, synchronization with the other callers may be needed as we may be dealing with a replicated invocation as discussed before.

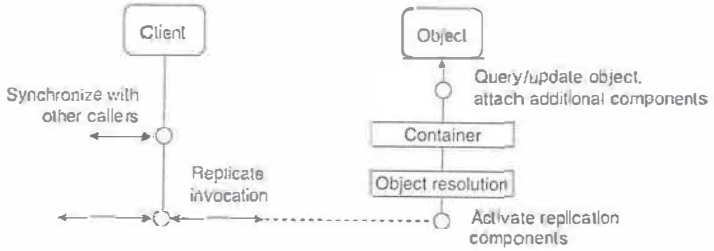

Figure 10-16. A general framework for separating replication algorithms from objects in an EJB environment.

Once it has been decided that the invocation can be carried out, the interceptor in the client-side stub can take decisions on where to be forward the request to, or possibly implement a fail-over mechanism when a replica cannot be reached.

Finally, the server-side interceptor handles the invocation. In fact, this interceptor is split into two. At the first point, just after the request has come in and before it is handed over to an adapter, the replication algorithm gets control. It can then analyze for whom the request is intended allowing it to activate, if necessary, any replication objects that it needs to carry out the replication. The second point is just before the invocation, allowing the replication algorithm to, for example, get and set attribute values of the replicated object.

The interesting aspect is that the framework can be set up independent of any replication algorithm, thus leading to a complete separation of object functionality and replication of objects.

10.6.2 Replicated Invocations

Another problem that needs to be solved is that of replicated invocations. Consider an object A calling another object B as shown in Fig. 10-17. Object B is assumed to call yet another object C. If B is replicated, each replica of B will, in principle, call C independently. The problem is that C is now called multiple times instead of only once. If the called method on C results in the transfer of $100,000, then clearly, someone is going to complain sooner or later.

There are not many general-purpose solutions to solve the problem of replicated invocations. One solution is to simply forbid it (Maassen et al., 2001), which makes sense when performance is at stake. However, when replicating for fault tolerance, the following solution proposed by Mazouni et al. (1995) may be deployed. Their solution is independent of the replication policy, that is, the exact details of how replicas are kept consistent. The essence is to provide a replication-aware communication layer on top of which (replicated) objects execute. When a replicated object B invokes another replicated object C, the invocation

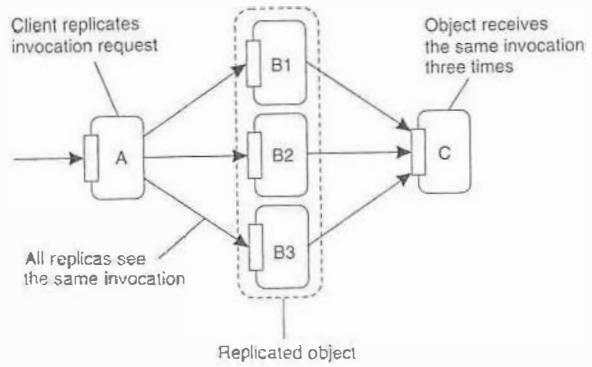

Figure 10-17. The problem of replicated method invocations.

request is first assigned the same, unique identifier by each replica of B. At that point, a coordinator of the replicas of B forwards its request to all the replicas of object C, while the other replicas of B hold back their copy of the invocation request, as shown in Fig. 10-18(a). The result is that only a single request is forwarded to each replica of C.

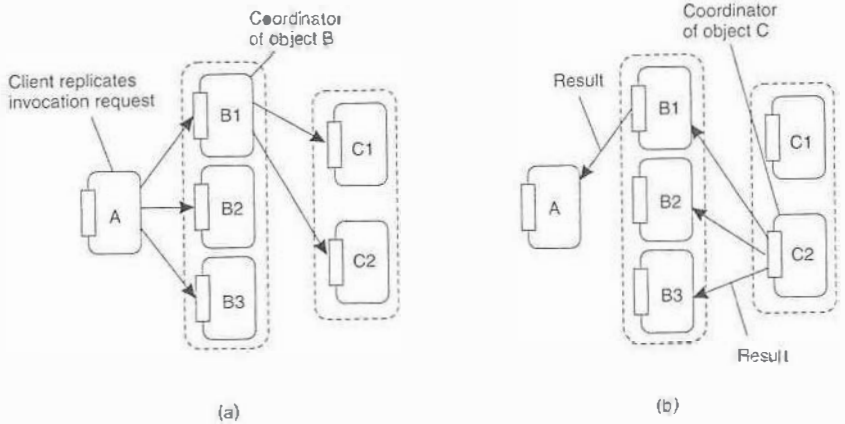

Figure 10-18. (a) Forwarding an invocation request from a replicated object to another replicated object. (b) Returning a reply from one replicated object to another.

The same mechanism is used to ensure that only a single reply message is returned to the replicas of B. This situation is shown in Fig. 10-18(b). A coordinator

of the replicas of C notices it is dealing with a replicated reply message that has been generated by each replica of C. However, only the coordinator forwards that reply to the replicas of object B, while the other replicas of C hold back their copy of the reply message.

When a replica of B receives a reply message for an invocation request it had either forwarded to C or held back because it was not the coordinator, the reply is then handed to the actual object.

In essence, the scheme just described is based on using multicast communication, but in preventing that the same message is multicast by different replicas. As such, it is essentially a sender-based scheme. An alternative solution is to let a receiving replica detect multiple copies of incoming messages belonging to the same invocation, and to pass only one copy to its associated object. Details of this scheme are left as an exercise.

10.7 FAULT TOLERANCE

Like replication, fault tolerance in most distributed object-based systems use the same mechanisms as in other distributed systems, following the principles we discussed in Chap. 8. However, when it comes to standardization, CORBA arguably provides the most comprehensive specification.

10.7.1 Example: Fault-Tolerant CORBA

The basic approach for dealing with failures in CORBA is to replicate objects into **object groups**. Such a group consists of one or more identical copies of the same object. However, an object group can be referenced as if it were a single object. A group offers the same interface as the replicas it contains. In other words, replication is transparent to clients. Different replication strategies are supported, including primary-backup replication, active replication, and quorum-based replication. These strategies have all been discussed in Chap. 7. There are various other properties associated with object groups, the details of which can be found in OMG (2004a).

To provide replication and failure transparency as much as possible, object groups should not be distinguishable from normal CORBA objects, unless an application prefers otherwise. An important issue, in this respect, is how object groups are referenced. The approach followed is to use a special kind of IOR, called an **Interoperable Object Group Reference (IOGR)**. The key difference with a normal IOR is that an IOGR contains multiple references to *different* objects, notably replicas in the same object group. In contrast, an IOR may also contain multiple references, but all of them will refer to the *same* object, although possibly using a different access protocol.

Whenever a client passes an IOGR to its runtime system (RTS), that RTS attempts to bind to one of the referenced replicas. In the case of IIOP, the RTS may possibly use additional information it finds in one of the IIOP profiles of the IOGR. Such information can be stored in the *Components* field we discussed previously. For example, a specific IIOP profile may refer to the primary or a backup of an object group, as shown in Fig. 10-19, by means of the separate tags *TAG_PRIMARY* and *TAG_BACKUP*, respectively.

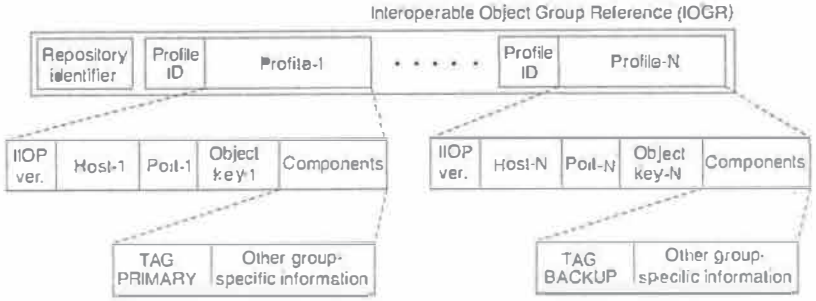

Figure 10-19. A possible organization of an IOGR for an object group having a primary and backups.

If binding to one of the replicas fails, the client RTS may continue by attempting to bind to another replica, thereby following any policy for next selecting a replica that it suits to best. To the client, the binding procedure is completely transparent: it appears as if the client is binding to a regular CORBA object.

An Example Architecture

To support object groups and to handle additional failure management, it is necessary to add components to CORBA. One possible architecture of a fault-tolerant version of CORBA is shown in Fig. 10-20. This architecture is derived from the Eternal system (Moser et al., 1998; and Narasimhan et al., 2000), which provides a fault tolerance infrastructure constructed on top of the Totem reliable group communication system (Moser et al., 1996).

There are several components that play an important role in this architecture. By far the most important one is the replication manager, which is responsible for creating and managing a group of replicated objects. In principle, there is only one replication manager, although it may be replicated for fault tolerance.

As we have stated, to a client there is no fundamental difference between an object group and any other type of CORBA object. To create an object group, a client simply invokes the normal create_object operation as offered, in this case,

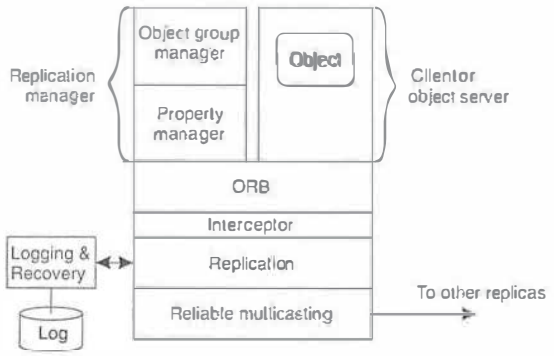

Figure 10-20. An example architecture of a fault-tolerant CORBA system.

by the replication manager, specifying the type of object to create. The client remains unaware of the fact that it is implicitly creating an object group. The number of replicas that are created when starting a new object group is normally determined by the system-dependent default value. The replica manager is also responsible for replacing a replica in the case of a failure, thereby ensuring that the number of replicas does not drop below a specified minimum.

The architecture also shows the use of message-level interceptors. In the case of the Eternal system, each invocation is intercepted and passed to a separate replication component that maintains the required consistency for an object group and which ensures that messages are logged to enable recovery.

Invocations are subsequently sent to the other group members using reliable, totally-ordered multicasting. In the case of active replication, an invocation request is passed to each replica object by handing it to that object's underlying runtime system. However, in the case of passive replication, an invocation request is passed only to the RTS of the primary, whereas the other servers only log the invocation request for recovery purposes. When the primary has completed the invocation, its state is then multicast to the backups.

This architecture is based on using interceptors. Alternative solutions exist as well, including those in which fault tolerance has been incorporated in the runtime system (potentially affecting interoperability), or in which special services are used on top of the RTS to provide fault tolerance. Besides these differences, practice shows that there are other problems not (yet) covered by the CORBA standard. As an example of one problem that occurs in practice, if replicas are created on different implementations, there is no guarantee that this approach will actually work. A review of the different approaches and an assessment of fault tolerance in CORBA is discussed in Felber and Narasimhan (2004).

10.7.2 Example: Fault-Tolerant Java

Considering the popularity of Java as a language and platform for developing distributed applications, some effort has also been into adding fault tolerance to the Java runtime system. An interesting approach is to ensure that the Java virtual machine can be used for active replication.

Active replication essentially dictates that the replica servers execute as deterministic finite-state machines (Schneider, 1990). An excellent candidate in Java to fulfill this role is the Java Virtual Machine (JVM). Unfortunately, the JVM is not deterministic at all. There are various causes for nondeterministic behavior, identified independently by Napper et al. (2003) and Friedman and Kama (2003):

1. JVM can execute native code, that is, code that is external to the JVM and provided to the latter through an interface. The JVM treats native code like a black box: it sees only the interface, but has no clue about the (potentially nondeterministic) behavior that a call causes. Therefore, in order to use the JVM for active replication, it is necessary to make sure that native code behaves in a deterministic way.

2. Input data may be subject to nondeterminism. For example, a shared variable that can be manipulated by multiple threads may change for different instances of the JVM as long as threads are allowed to operate concurrently. To control this behavior, shared data should at the very least be protected through locks. As it turned out, the Java runtime environment did not always adhere to this rule, despite its support for multithreading.

3. In the presence of failures, different JVMs will produce different output revealing that the machines have been replicated. This difference may cause problems when the JVMs need to brought back into the same state. Matters are simplified if one can assume that all output is idempotent (i.e., can simply be replayed), or is testable so that one can check whether output was produced before a crash or not. Note that this assumption is necessary in order to allow a replica server to decide whether or not it should re-execute an operation.

Practice shows that turning the JVM into a deterministic finite-state machine is by no means trivial. One problem that needs to be solved is the fact that replica servers may crash. One possible organization is to let the servers run according to a primary-backup scheme. In such a scheme, one server coordinates all actions that need to be performed, and from time to time instructs the backup to do the same. Careful coordination between primary and backup is required, of course,

Note that despite the fact that replica servers are organized in a primary-backup setting, we are still dealing with active replication: the replicas are kept up to date by letting each of them execute the same operations in the same order. However, to ensure the same nondeterministic behavior by all of the servers, the behavior of one server is taken as the one to follow.

In this setting, the approach followed by Friedman and Kama (2003) is to let the primary first execute the instructions of what is called a frame. A frame consists of the execution of several context switches and ends either because all threads are blocking for I/O to complete, or after a predefined number of context switches has taken place. Whenever a thread issues an I/O operation, the thread is blocked by the JVM put on hold. When a frame starts, the primary lets all I/O requests proceed, one after the other, and the results are sent to the other replicas. In this way, at least deterministic behavior with respect to I/O operations is enforced.

The problem with this scheme is easily seen: the primary is always ahead of the other replicas. There are two situations we need to consider. First, if a replica server other than the primary crashes, no real harm is done except that the degree of fault tolerance drops. On the other hand, when the primary crashes, we may find ourselves in a situation that data (or rather, operations) are lost.

To minimize the damage, the primary works on a per-frame basis. That is, it sends update information to the other replicas only after completion of its current frame. The effect of this approach is that when the primary is working on the k-th frame, that the other replica servers have all the information needed to process the frame preceding the k-th one. The damage can be limited by making frames small, at the price of more communication between the primary and the backups.

10.8 SECURITY

Obviously, security plays an important role in any distributed system and object-based ones are no exception. When considering most object-based distributed systems, the fact that distributed objects are remote objects immediately leads to a situation in which security architectures for distributed systems are very similar. In essence, each object is protected through standard authentication and authorization mechanisms, like the ones we discussed in Chap. 9.

To make clear how security can fit in specifically in an object-based distributed system, we shall discuss the security architecture for the Globe system. As we mentioned before, Globe supports truly distributed objects in which the state of a single object can be spread and replicated across multiple machines. Remote objects are just a special case of Globe objects. Therefore, by considering the Globe security architecture, we can also see how its approach can be equally applied to more traditional object-based distributed systems. After discussing Globe, we briefly take a look at security in traditional object-based systems.

10.8.1 Example: Globe

As we said, Globe is one of the few distributed object-based systems in which an object's state can be physically distributed and replicated across multiple machines. This approach also introduces specific security problems, which have led to an architecture as described in Popescu et al. (2002).

Overview

When we consider the general case of invoking a method on a remote object, there are at least two issues that are important from a security perspective: (1) is the caller invoking the correct object and (2) is the caller allowed to invoke that method. We refer to these two issues as **secure object binding** and **secure method invocation**, respectively. The former has everything to do with authentication, whereas the latter involves authorization. For Globe and other systems that support either replication or moving objects around, we have an additional problem, namely that of **platform security**. This kind of security comprises two issues. First, how can the platform to which a (local) object is copied be protected against any malicious code contained in the object, and secondly, how can the object be protected against a malicious replica server.

Being able to copy objects to other hosts also brings up another problem. Because the object server that is hosting a copy of an object need not always be fully trusted, there must be a mechanism that prevents that every replica server hosting an object from being allowed to also execute any of an object's methods. For example, an object's owner may want to restrict the execution of update methods to a small group of replica servers, whereas methods that only read the state of an object may be executed by any authenticated server. Enforcing such policies can be done through reverse access control, which we discuss in more detail below.

There are several mechanisms deployed in Globe to establish security. First, every Globe object has an associated public/private key pair, referred to as the object key. The basic idea is that anyone who has knowledge about an object's private key can set the access policies for users and servers. In addition, every replica has an associated replica key, which is also constructed as a public/private key pair. This key pair is generated by the object server currently hosting the specific replica. As we will see, the replica key is used to make sure that a specific replica is part of a given distributed shared object. Finally, each user is also assumed to have a unique public/private key pair, known as the user key.

These keys are used to set the various access rights in the form of certificates. Certificates are handed out per object. There are three types, as shown in Fig. 10-21. A user certificate is associated with a specific user and specifies exactly which methods that user is allowed to invoke. To this end, the certificate contains

a bit string U with the same length as the number of methods available for the object. $U[i] = 1$ if and only if the user is allowed to invoke method M_i. Likewise, there is also a **replica certificate** that specifies, for a given replica server, which methods it is allowed to execute. It also has an associated bit string R, where $R[i] = 1$ if and only if the server is allowed to execute method M_i.

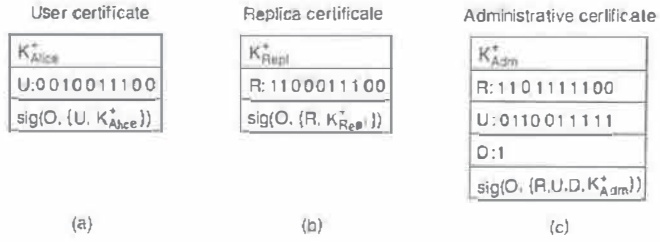

Figure 10-21. Certificates in Globe: (a) a user certificate, (b) a replica certificate, (c) an administrative certificate.

For example, the user certificate in Fig. 10-21(a) tells that Alice (who can be identified through her public key K^{+}_{Alice}), has the right to invoke methods M_2, M_5, M_6, and M_7 (note that we start indexing U at 0). Likewise, the replica certificate states that the server owning K^{+}_{Repl} is allowed to execute methods M_0, M_1, M_5, M_6, and M_7.

An administrative certificate can be used by any authorized entity to issue user and replica certificates. In the case, the R and U bit strings specify for which methods and which entities a certificate can be created. Moreover, there is bit indicating whether an administrative entity can delegate (part of) its rights to someone else. Note that when Bob in his role as administrator creates a user certificate for Alice, he will sign that certificate with his own signature, not that of the object. As a consequence, Alice's certificate will need to be traced back to Bob's administrative certificate, and eventually to an administrative certificate signed with the object's private key.

Administrative certificates come in handy when considering that some Globe objects may be massively replicated. For example, an object's owner may want to manage only a relatively small set of permanent replicas, but delegate the creation of server-initiated replicas to the servers hosting those permanent replicas. In that case, the owner may decide to allow a permanent replica to install other replicas for read-only access by all users. Whenever Alice wants to invoke a read-only method, she will succeed (provided she is authorized). However, when wanting to invoke an update method, she will have to contact one of the permanent replicas, as none of the other replica servers is allowed to execute such methods.

As we explained, the binding process in Globe requires that an object identifier (OID) is resolved to a contact address. In principle, any system that supports

flat names can be used for this purpose. To securely associate an object's public key to its OID, we simply compute the OID as a 160-bit secure hash of the public key. In this way, anyone can verify whether a given public key belongs to a given OID. These identifier are also known as **self-certifying names**, a concept pioneered in the Secure File System (Mazieres et al., 1999), which we will discuss in Chap. 11.

We can also check whether a replica R belongs to an object O. In that case, we merely need to inspect the replica certificate for R, and check who issued it. The signer may be an entity with administrative rights, in which case we need to inspect its administrative certificate. The bottom line is that we can construct a chain of certificates of which the last one is signed using the object's private key. In that case, we know that R is part of O.

To mutually protect objects and hosts against each other, techniques for mobile code, as described in Chap. 9 are deployed. Detecting that objects have been tampered with can be done with special auditing techniques which we will describe in Chap. 12.

Secure Method Invocation

Let us now look into the details of securely invoking a method of a Globe object. The complete path from requesting an invocation to actually executing the operation at a replica is sketched in Fig. 10-22. A total of 13 steps need to be executed in sequence, as shown in the figure and described in the following text.

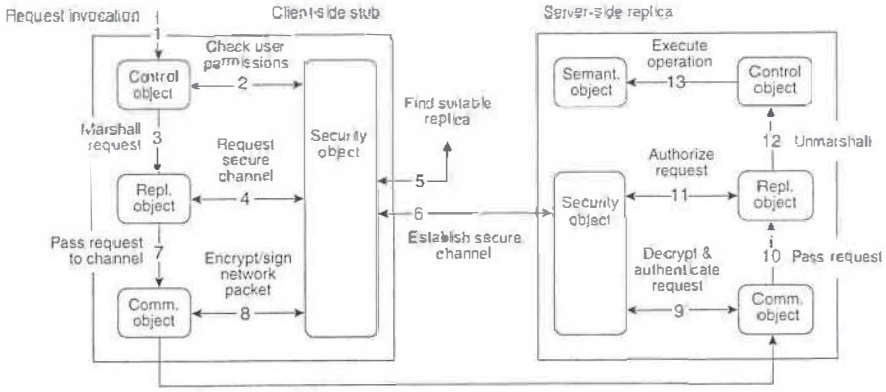

Figure 10-22. Secure method invocation in Globe.

1. First, an application issues a invocation request by locally calling the associated method, just like calling a procedure in an RPC.
2. The control subobject checks the user permissions with the information stored in the local security object. In this case, the security object should have a valid user certificate.
3. The request is marshaled and passed on.
4. The replication subobject requests the middleware to set up a secure channel to a suitable replica.
5. The security object first initiates a replica lookup. To achieve this goal, it could use any naming service that can look up replicas that have been specified to be able to execute certain methods. The Globe location service has been modified to handle such lookups (Ballintijn, 2003).
6. Once a suitable replica has been found, the security subobject can set up a secure channel with its peer, after which control is returned to the replication subobject. Note that part of this establishment requires that the replica proves it is allowed to carry out the requested invocation.
7. The request is now passed on to the communication subobject.
8. The subobject encrypts and signs the request so that it can pass through the channel.
9. After its receipt, the request is decrypted and authenticated.
10. The request is then simply passed on to the server-side replication subobject.
11. Authorization takes place: in this case the user certificate from the client-side stub has been passed to the replica so that we can verify that the request can indeed be carried out.
12. The request is then unmarshaled.
13. Finally, the operation can be executed.

Although this may seem to be a relatively large number of steps, the example shows how a secure method invocation can be broken down into small units, each unit being necessary to ensure that an authenticated client can carry out an authorized invocation at an authenticated replica. Virtually all object-based distributed systems follow these steps. The difference with Globe is that a suitable replica needs to be located, and that this replica needs to prove it may execute the method call. We leave such a proof as an exercise to the reader.

10.8.2 Security for Remote Objects

When using remote objects we often see that the object reference itself is implemented as a complete client-side stub, containing all the information that is needed to access the remote object. In its simplest form, the reference contains the exact contact address for the object and uses a standard marshaling and communication protocol to ship an invocation to the remote object.

However, in systems such as Java, the client-side stub (called a **proxy**) can be virtually anything. The basic idea is that the developer of a remote object also develops the proxy and subsequently registers the proxy with a directory service. When a client is looking for the object, it will eventually contact the directory service, retrieve the proxy, and install it. There are obviously some serious problems with this approach.

First, if the directory service is hijacked, then an attacker may be able to return a bogus proxy to the client. In effect, such a proxy may be able to compromise all communication between the client and the server hosting the remote object, damaging both of them.

Second, the client has no way to authenticate the server: it only has the proxy and all communication with the server necessarily goes through that proxy. This may be an undesirable situation, especially because the client now simply needs to trust the proxy that it will do its work correctly.

Likewise, it may be more difficult for the server to authenticate the client. Authentication may be necessary when sensitive information is sent to the client. Also, because client authentication is now tied to the proxy, we may also have the situation that an attacker is spoofing a client causing damage to the remote object.

Li et al. (2004b) describe a general security architecture that can be used to make remote object invocations safer. In their model, they assume that proxies are indeed provided by the developer of a remote object and registered with a directory service. This approach is followed in Java RMI, but also Jini (Sun Microsystems, 2005).

The first problem to solve is to authenticate a remote object. In their solution, Li and Mitchell propose a two-step approach. First, the proxy which is downloaded from a directory service is signed by the remote object allowing the client to verify its origin. The proxy, in turn, will authenticate the object using TLS with server authentication, as we discussed in Chap. 9. Note that it is the object developer's task to make sure that the proxy indeed properly authenticates the object. The client will have to rely on this behavior, but because it is capable of authenticating the proxy, relying on object authentication is at the same level as trusting the remote object to behave decently.

To authenticate the client, a separate authenticator is used. When a client is looking up the remote object, it will be directed to this authenticator from which it downloads an **authentication proxy**. This is a special proxy that offers an interface by which the client can have itself authenticated by the remote object. If this authentication succeeds, then the remote object (or actually, its object server) will pass on the actual proxy to the client. Note that this approach allows for authentication independent of the protocol used by the actual proxy, which is considered an important advantage.

Another important advantage of separating client authentication is that it is now possible to pass dedicated proxies to clients. For example, certain clients may be allowed to request only execution of read-only methods. In such a case, after authentication has taken place, the client will be handed a proxy that offers only such methods, and no other. More refined access control can easily be envisaged.

10.9 SUMMARY

Most object-based distributed systems use a remote-object model in which an object is hosted by server that allows remote clients to do method invocations. In many cases, these objects will be constructed at runtime, effectively meaning that their state, and possibly also code is loaded into an object server when a client does a remote invocation. Globe is a system in which truly distributed shared objects are supported. In this case, an object's state may be physically distributed and replicated across multiple machines.

To support distributed objects, it is important to separate functionality from extra-functional properties such as fault tolerance or scalability. To this end, advanced object servers have been developed for hosting objects. An object server provides many services to basic objects, including facilities for storing objects, or to ensure serialization of incoming requests. Another important role is providing the illusion to the outside world that a collection of data and procedures operating on that data correspond to the concept of an object. This role is implemented by means of object adapters.

When it comes to communication, the prevalent way to invoke an object is by means of a remote method invocation (RMI), which is very similar to an RPC. An important difference is that distributed objects generally provide a systemwide object reference, allowing a process to access an object from any machine. Global object reference solve many of the parameter-passing problems that hinder access transparency of RPCs.

There are many different ways in which these object references can be implemented, ranging from simple passive data structures describing precisely where a remote object can be contacted, to portable code that need simply be invoked by a client. The latter approach is now commonly adopted for Java RMI.

There are no special measures in most systems to handle object synchronization. An important exception is the way that synchronized Java methods are treated: the synchronization takes place only between clients running on the same machine. Clients running on different machines need to take special synchronization measures. These measures are not part of the Java language.

Entry consistency is an obvious consistency model for distributed objects and is (often implicitly) supported in many systems. It is obvious as we can naturally associate a separate lock for each object. One of the problems resulting from replicating objects are replicated invocations. This problem is more evident because objects tend to be treated as black boxes.

Fault tolerance in distributed object-based systems very much follows the approaches used for other distributed systems. One exception is formed by trying to make the Java virtual machine fault tolerant by letting it operate as a deterministic finite state machine. Then, by replicating a number of these machines, we obtain a natural way for providing fault tolerance.

Security for distributed objects evolves around the idea of supporting secure method invocation. A comprehensive example that generalizes these invocations to replicated objects is Globe. As it turns out, it is possible to cleanly separate policies from mechanisms. This is true for authentication as well as authorization. Special attention needs to be paid to systems in which the client is required to download a proxy from a directory service, as is commonly the case for Java.

PROBLEMS

1. We made a distinction between remote objects and distributed objects. What is the difference?
2. Why is it useful to define the interfaces of an object in an Interface Definition Language?
3. Some implementations of distributed-object middleware systems are entirely based on dynamic method invocations. Even static invocations are compiled to dynamic ones. What is the benefit of this approach?
4. Outline a simple protocol that implements at-most-once semantics for an object invocation.
5. Should the client and server-side objects for asynchronous method invocation be persistent?
6. In the text, we mentioned that an implementation of CORBA's asynchronous method invocation do not affect the server-side implementation of an object. Explain why this is the case.
7. Give an example in which the (inadvertent) use of callback mechanisms can easily lead to an unwanted situation.
8. Is it possible for an object to have more than one servant?
9. Is it possible to have system-specific implementations of CORBA object references while still being able to exchange references with other CORBA-based systems?

10. How can we authenticate the contact addresses returned by a lookup service for secure Globe objects?
11. What is the key difference between object references in CORBA and those in Globe?
12. Consider Globe. Outline a simple protocol by which a secure channel is set up between a user proxy (which has access to the Alice's private key) and a replica that we know for certain can execute a given method.
13. Give an example implementation of an object reference that allows a client to bind to a transient remote object.
14. Java and other languages support exceptions, which are raised when an error occurs. How would you implement exceptions in RPCs and RMIs?
15. How would you incorporate persistent asynchronous communication into a model of communication based on RMIs to remote objects?
16. Consider a distributed object-based system that supports object replication, in which *all* method invocations are totally ordered. Also, assume that an object invocation is atomic (e.g., because every object is automatically locked when invoked). Does such a system provide entry consistency? What about sequential consistency?
17. Describe a receiver-based scheme for dealing with replicated invocations, as mentioned in the text.