Names play an important role in all computer systems. They are used to share resources, to uniquely identify entities, to refer to locations, and so on. An important issue with naming is that a name can be resolved to the entity it refers to. Name resolution thus allows a process to access the named entity. To resolve names, it is necessary to implement a naming system. The difference between naming in distributed systems and nondistributed systems lies in the way naming systems are implemented.

In a distributed system, the implementation of a naming system is itself often distributed across multiple machines. How this distribution is done plays a key role in the efficiency and scalability of the naming system. In this chapter, we concentrate on three different, important ways that names are used in distributed systems.

First, after discussing some general issues with respect to naming, we take a closer look at the organization and implementation of human-friendly names. Typical examples of such names include those for file systems and the World Wide Web. Building worldwide, scalable naming systems is a primary concern for these types of names.

Second, names are used to locate mobile entities. As it turns out, naming systems for human-friendly names are not particularly suited for supporting large numbers of mobile entities, which may additionally be dispersed across a large-scale network. Alternative organizations are needed, such as those being used for mobile telephony where names are location-independent identifiers.
Our third and last topic deals with the organization of names. In particular, names that are no longer referenced, and thus can no longer be located and accessed, should be automatically removed. This subject is also known as garbage collection, and has its roots in programming languages. However, with the introduction of large-scale distributed object-based systems, automatically collecting unreferenced objects is becoming increasingly important.

4.1 NAMING ENTITIES

In this section, we first concentrate on different kinds of names, and how names are organized into name spaces. We then continue with a discussion of the important issue of how to resolve a name such that the entity it refers to can be accessed. Also, we explain various options for distributing and implementing large name spaces across multiple machines. The Internet Domain Name System and OSI's X.500 will be discussed as examples of large-scale naming services.

4.1.1 Names, Identifiers, and Addresses

A name in a distributed system is a string of bits or characters that is used to refer to an entity. An entity in a distributed system can be practically anything. Typical examples include resources such as hosts, printers, disks, and files. Other well-known examples of entities that are often explicitly named are processes, users, mailboxes, newsgroups, Web pages, graphical windows, messages, network connections, and so on.

Entities can be operated on. For example, a resource such as a printer offers an interface containing operations for printing a document, requesting the status of a print job, and the like. Furthermore, an entity such as a network connection may provide operations for sending and receiving data, setting quality-of-service parameters, requesting the status, and so forth.

To operate on an entity, it is necessary to access it, for which we need an access point. An access point is yet another, but special, kind of entity in a distributed system. The name of an access point is called an address. The address of an access point of an entity is also simply called an address of that entity.

An entity can offer more than one access point. As a comparison, a telephone can be viewed as an access point of a person, whereas the telephone number corresponds to an address. Indeed, many people nowadays have several telephone numbers, each number corresponding to a point where they can be reached. In a distributed system, a typical example of an access point is a host running a specific server, with its address formed by the combination of, for example, an IP address and port number (i.e., the server's transport-level address).

An entity may change its access points in the course of time. For example, when a mobile computer moves to another location, it is often assigned a different

IP address than the one it had before. Likewise, when a person moves to another city or country, it is often necessary to change telephone numbers as well. In a similar fashion, changing jobs or Internet Service Provider, means changing your e-mail address.

An address is thus just a special kind of name: it refers to an access point of an entity. Because an access point is tightly associated with an entity, it would seem convenient to use the address of an access point as a regular name for the associated entity. Nevertheless, this is hardly ever done.

There are many benefits to treating addresses as a special type of name. For example, it is not uncommon to regularly reorganize a distributed system, so that a specific server, such as the one handling FTP requests, is now running on a different host than previously. The old machine on which the server used to be running may be reassigned to a completely different server, such as a back-up server for the local file system. In other words, an entity may easily change an access point, or an access point may be reassigned to a different entity.

If an address is used to refer to an entity, we will have an invalid reference the instant the access point changes or is reassigned to another entity. For example, imagine that an organization's FTP service would be known only by the address of the host running the FTP service. As soon as that server is moved to another host, the whole FTP service would become inaccessible until the new address is known to all its users. In this case, it would have been much better to let the FTP service be known by a separate name, independent of the address of the associated FTP service.

Likewise, if an entity offers more than one access point, it is not clear which address to use as a reference. For instance, as we discussed in Chap. 1, many organizations distribute their Web service across several servers. If we would use the addresses of those servers as a reference for the Web service, it is not obvious which address should be chosen as the best one. A much better solution would be to have a single name for the Web service, independent from the addresses of the different Web servers.

These examples illustrate that a name for an entity that is independent from its addresses, is often much easier and more flexible to use. Such a name is called location independent.

In addition to addresses, there are other types of names that deserve special treatment, such as names that are used to uniquely identify an entity. A true identifier is a name that has the following properties (Wieringa and de Jonge, 1995):

1. An identifier refers to at most one entity.
2. Each entity is referred to by at most one identifier.
3. An identifier always refers to the same entity (i.e., it is never reused).

By using identifiers, it becomes much easier to unambiguously refer to an entity. For example, assume two processes each refer to an entity by means of an
identifier. To check if the processes are referring to the same entity, it is sufficient to test if the two identifiers are equal. Such a test would not be sufficient if the two processes were using regular, nonidentifying names. For example, the name “John Smith” cannot be taken as a unique reference to just a single person.

Likewise, if an address can be reassigned to a different entity, we cannot use an address as an identifier. Consider the use of telephone numbers, which are reasonably stable in the sense that a number generally refers to the same person or organization. However, using a telephone number as an identifier will not work, as it can be reassigned in the course of time. Consequently, Bob’s new bakery may be receiving phone calls for Alice’s old hardware store for a long time. In this case, it would have been better to use a true identifier for Alice instead of her phone number.

Addresses and identifiers are two important types of names that are each used for very different purposes. In many computer systems, addresses and identifiers are represented in machine-readable form only, that is, in the form of bit strings. For example, an Ethernet address is essentially a random string of 48 bits. Likewise, memory addresses are typically represented as 32-bit or 64-bit strings.

Another important type of name is that which is tailored to be used by humans, also referred to as human-friendly names. In contrast to addresses and identifiers, a human-friendly name is generally represented as a character string. These names appear in many different forms. For example, files in UNIX systems have character-string names that can be as long as 255 characters, and which are defined entirely by the user. Similarly, DNS names are represented as relatively simple case-insensitive character strings.

**Name Spaces**

Names in a distributed system are organized into what is commonly referred to as a name space. A name space can be represented as a labeled, directed graph with two types of nodes. A leaf node represents a named entity and has the property that it has no outgoing edges. A leaf node generally stores information on the entity it is representing—for example, its address—so that a client can access it. Alternatively, it can store the state of that entity, such as in the case of file systems in which a leaf node actually contains the complete file it is representing. We return to the contents of nodes below.

In contrast to a leaf node, a directory node has a number of outgoing edges, each labeled with a name, as shown in Fig. 4-1. Each node in a naming graph is considered as yet another entity in a distributed system, and, in particular, has an associated identifier. A directory node stores a table in which an outgoing edge is represented as a pair (edge label, node identifier). Such a table is called a directory table.

The naming graph shown in Fig. 4-1 has one node, namely n0, which has only outgoing and no incoming edges. Such a node is called the root (node) of the naming graph. Although it is possible for a naming graph to have several root nodes, for simplicity, many naming systems have only one. Each path in a naming graph can be referred to by the sequence of labels corresponding to the edges in that path, such as

\[ N<\text{label-1}, \text{label-2}, ..., \text{label-n}> \]

where \( N \) refers to the first node in the path. Such a sequence is called a path name. If the first node in a path name is the root of the naming graph, it is called an absolute path name. Otherwise, it is called a relative path name.

It is important to realize that names are always organized in a name space. As a consequence, a name is always defined relative only to a directory node. In this sense, the term absolute name is somewhat misleading. Likewise, the difference between global and local names can sometimes be confusing. A global name is a name that denotes the same entity, no matter where that entity is used in a system. In other words, a global name is always interpreted with respect to the same directory node. In contrast, a local name is a name whose interpretation depends on where that name is being used. Put differently, a local name is essentially a relative name whose directory in which it is contained is (implicitly) known. We return to these issues when discussing name resolution.

This description of a naming graph comes close to what is implemented in many file systems. However, instead of writing the sequence of edge labels to represent a path name, path names in file systems are generally represented as a single string in which the labels are separated by a special separator character, such as a slash ("/"). This character is also used to indicate whether or not a path name is absolute. For example, in Fig. 4-1, instead of using the path name \( n0<\text{home, steen, mbox}> \) it is common practice to use its string representation \(/\text{home/steen/mbox}\). Note also that when there are several paths that lead to the same node, that node can be represented by different path names. For example, node n5 in Fig. 4-1, can be referred to by \(/\text{home/steen/keys}\) as well as \(/\text{keys}\). The string representation of path names can be equally well applied to naming graphs...
other than those used for only file systems. In Plan 9 (Pike et al., 1995), all resources, such as processes, hosts, I/O devices, and network interfaces, are named in the same fashion as traditional files. This approach is analogous to implementing a single naming graph for all resources in a distributed system.

There are many different ways to organize a name space. As we mentioned, most name spaces have only a single root node. In many cases, a name space is also strictly hierarchical in the sense that the naming graph is organized as a tree. This means that each node except the root has exactly one incoming edge; the root has no incoming edges. As a consequence, each node also has exactly one associated (absolute) path name.

The naming graph shown in Fig. 4-1 is an example of directed acyclic graph. In such an organization, a node can have more than one incoming edge, but the graph is not permitted to have a cycle. There are also name spaces that do not have this restriction.

To make matters more concrete, consider the way that files in a traditional UNIX file system are named. In a naming graph for UNIX, a directory node represents a file directory, whereas a leaf node represents a file. There is a single root directory, represented in the naming graph by the root node. The implementation of the naming graph is an integral part of the complete implementation of the file system. That implementation consists of a contiguous series of blocks from a logical disk, generally divided into a boot block, a superblock, a series of index nodes (called inodes), and file data blocks. See also (Crowley, 1997; Nutt, 2000; Tanenbaum and Woodhull, 1997). This organization is shown in Fig. 4-2.

![Figure 4-2. The general organization of the UNIX file system implementation on a logical disk of contiguous disk blocks.](image)

The boot block is a special block of data and instructions that are automatically loaded into main memory when the system is booted. The boot block is used to load the operating system into main memory.

The superblock contains information on the entire file system, such as its size, which blocks on disk are not yet allocated, which inodes are not yet used, and so on. Inodes are referred to by an index number, starting at number zero, which is reserved for the inode representing the root directory.

Each inode contains exact information on where the data of its associated file can be found on disk. In addition, an inode contains information on its owner, time of creation and last modification, protection, and the like. Consequently, when given the index number of an inode, it is possible to access its associated file. Each directory is implemented as a file as well. This is also the case for the root directory, which contains a mapping between file names and index numbers of inodes. It is thus seen that the index number of an inode corresponds to a node identifier in the naming graph.

### 4.1.2 Name Resolution

Name spaces offer a convenient mechanism for storing and retrieving information about entities by means of names. More generally, given a path name, it should be possible to look up any information stored in the node referred to by that name. The process of looking up a name is called name resolution.

To explain how name resolution works, consider a path name such as $\text{N:label-1, label-2, ..., label-n}$. Resolution of this name starts at node $N$ of the naming graph, where the name $\text{label-1}$ is looked up in the directory table, and which returns the identifier of the node to which $\text{label-1}$ refers. Resolution then continues at the identified node by looking up the name $\text{label-2}$ in its directory table, and so on. Assuming that the named path actually exists, resolution stops at the last node referred to by $\text{label-n}$, by returning the content of that node.

A name lookup returns the identifier of a node from where the name resolution process continues. In particular, it is necessary to access the directory table of the identified node. Consider again a naming graph for a UNIX file system. As mentioned, a node identifier is implemented as the index number of an inode. Accessing a directory table means that first the inode has to be read to find out where the actual data are stored on disk, and then subsequently to read the data blocks containing the directory table.

### Closure Mechanism

Name resolution can take place only if we know how and where to start. In our example, the starting node was given, and we assumed we had access to its directory table. Knowing how and where to start name resolution is generally referred to as a closure mechanism. Essentially, a closure mechanism deals with selecting the initial node in a name space from which name resolution is to start (Radia, 1989). What makes closure mechanisms sometimes hard to understand is that they are necessarily partly implicit and may be very different when comparing them to each other.

For example, name resolution in the naming graph for a UNIX file system makes use of the fact that the inode of the root directory is the first inode in the logical disk representing the file system. Its actual byte offset is calculated from the values in other fields of the superblock, together with hard-coded information in the operating system itself on the internal organization of the superblock.
To make this point clear, consider the string representation of a file name such as /home/steen/mbox. To resolve this name, it is necessary to already have access to the directory table of the root node of the appropriate naming graph. Being a root node, the node itself cannot have been looked up unless it is implemented as a different node in a another naming graph, say $G$. But in that case, it would have been necessary to already have access to the root node of $G$. Consequently, resolving a file name requires that some mechanism has already been implemented by which the resolution process can start.

A completely different example is the use of the string "0031204430784." Many people will not know what to do with these numbers, unless they are told that the sequence is a telephone number. That information is enough to start the resolution process, in particular, by dialing the number. The telephone system subsequently does the rest.

As a last example, consider the use of global and local names in distributed systems. A typical example of a local name is an environment variable. For example, in UNIX systems, the variable named HOME is used to refer to the home directory of a user. Each user has its own copy of this variable, which is initialized to the global, systemwide name corresponding to the user's home directory. The closure mechanism associated with environment variables ensures that the name of the variable is properly resolved by looking it up in a user-specific table.

Linking and Mounting

Strongly related to name resolution is the use of aliases. An alias is another name for the same entity. An environment variable is an example of an alias. In terms of naming graphs, there are basically two different ways to implement an alias. The first approach is to simply allow multiple absolute paths names to refer to the same node in a naming graph. This approach is illustrated in Fig. 4-1, in which node $n^5$ can be referred to by two different path names. In UNIX terminology, both path names /keys and /home/steen/keys in Fig. 4-1 are called hard links to node $n^5$.

The second approach is to represent an entity by a leaf node, say $N$, but instead of storing the address or state of that entity, the node stores an absolute path name. When first resolving an absolute path name that leads to $N$, name resolution will return the path name stored in $N$, at which point it can continue with resolving that new path name. This principle corresponds to the use of symbolic links in UNIX file systems, and is illustrated in Fig. 4-3. In this case, the path name /home/steen/keys, which refers to a node containing the absolute path name /keys, is a symbolic link to node $n^5$.

Name resolution as described so far takes place completely within a single name space. However, name resolution can also be used to merge different name spaces in a transparent way. Let us first consider a mounted file system. In terms of our naming model, a mounted file system corresponds to letting a directory node store the identifier of a directory node from a different name space, which we refer to as a foreign name space. The directory node storing the node identifier is called a mounting point. Accordingly, the directory node in the foreign name space is called a mounting point. Normally, the mounting point is the root of a name space. During name resolution, the mounting point is looked up and resolution proceeds by accessing its directory table.

The principle of mounting can be generalized to other name spaces as well. In particular, what is needed is that a directory node that acts as a mount point stores all the necessary information for identifying and accessing the mounting point in the foreign name space. This approach has been followed in the Jade naming system (Rao and Peterson, 1993), and is actually also followed in many distributed file systems.

Consider a collection of name spaces that is distributed across different machines. In particular, each name space is implemented by a different server, each possibly running on a separate machine. Consequently, if we want to mount a foreign name space NS2 into a name space NS1, it may be necessary to communicate over a network with the server of NS2, as that server may be running on a different machine than the server for NS1. To mount a foreign name space in a distributed system requires at least the following information:

1. The name of an access protocol.
2. The name of the server.
3. The name of the mounting point in the foreign name space.

Note that each of these names needs to be resolved. The name of an access protocol needs to be resolved to the implementation of a protocol by which communication with the server of the foreign name space can take place. The name of the server needs to be resolved to an address where that server can be reached. As the last part in name resolution, the name of the mounting point needs to be resolved to a node identifier in the foreign name space.
In nondistributed systems, none of the three points may actually be needed. For example, in UNIX, there is no access protocol and no server. Also, the name of the mounting point is not necessary, as it is simply the root directory of the foreign name space.

The name of the mounting point is to be resolved by the server of the foreign name space. However, we also need name spaces and implementations for the access protocol and the server name. One possibility is to represent the three names listed above as a URL.

To make matters concrete, consider a situation in which a user with a laptop computer wants to access files that are stored on a remote file server. The client machine and the file server are both configured with Sun’s Network File System (NFS), which we will discuss in detail in Chap. 10. NFS is a distributed file system that comes with a protocol that describes precisely how a client can access a file stored on a (remote) NFS file server. In particular, to allow NFS to work across the Internet, a client can specify exactly which file it wants to access by means of an NFS URL, for example, nfs://flits.cs.vu.nl/home/steen. This URL names a file (which happens to be a directory) called /home/steen on an NFS file server flits.cs.vu.nl, which can be accessed by a client by means of the NFS protocol (Callaghan, 2000).

The name nfs is a well-known name in the sense that worldwide agreement exists on how to interpret that name. In other words, given that we are dealing with a URL, the name nfs will be resolved to an implementation of the NFS protocol. The server name is resolved to its address using the Domain Name System, which is discussed in a later section. As we said, /home/steen is resolved by the server of the foreign name space.

The organization of a file system on the client machine is partly shown in Fig. 4-4. The root directory has a number of user-defined entries, including a subdirectory called remote. This subdirectory is intended to include mount points for foreign name spaces such as the user’s home directory at the Vrije Universiteit. To this end, a directory node named /remote/vu is used to store the URL nfs://flits.cs.vu.nl/home/steen.

Now consider the name /remote/vu/mbox. This name is resolved by starting in the root directory on the client’s machine and continues until the node /remote/vu is reached. Name resolution then continues by returning the URL nfs://flits.cs.vu.nl/home/steen, in turn leading the client machine to contact the file server flits.cs.vu.nl by means of the NFS protocol, and to subsequently access directory /home/steen. Name resolution can then be continued by reading the file named mbox in that directory.

Distributed systems that allow mounting a remote file system as just described allow a client machine to, for example, execute the following commands:

```
   cd /remote/vu
   ls -l
```

which subsequently list the files in the directory /home/steen on the remote file server. The beauty of this all is that the user is spared the details of the actual access to the remote server. Ideally, only some loss in performance is noticed compared to accessing locally available files. In effect, to the client it appears that the name space rooted on the local machine, and the one rooted at /home/steen on the remote machine, form a single name space.

Mounting is one way to merge different name spaces. Another approach, which was followed in DEC’s Global Name Service (GNS), is to add a new root node and to make the existing root nodes its children (Lampson, 1986). This principle is shown in Fig. 4-5 and is explained below.

A problem with this approach is that existing names need to be changed. For instance, the absolute path name /home/steen in name space NS1 in Fig. 4-5 has now changed into a relative path name that is to be resolved starting in node n0, and corresponds to the absolute path name /n0/home/steen. To solve these problems and to allow other name spaces to be added in the future, names in GNS always (implicitly) include the identifier of the node from where resolution should normally start. So, for example, in name space NS1 in Fig. 4-5, the name /home/steen/keys is always expanded to include the node identifier n0, leading to n0/home/steen/keys. Expansion is generally hidden from users. It is assumed that a node identifier is universally unique. Consequently, even nodes from different name spaces are assumed to always have different node identifiers.
partition it into logical layers. Cheriton and Mann (1989) distinguish the following three layers.

The **global layer** is formed by highest-level nodes, that is, the root node and other directory nodes logically close to the root, that is, its children. Nodes in the global layer are often characterized by their stability, in the sense that directory tables are rarely changed. Such nodes may represent organizations, or groups of organizations, for which names are stored in the name space.

The **administrational layer** is formed by directory nodes that together are managed within a single organization. A characteristic feature of the directory nodes in the administrational layer is that they represent groups of entities that belong to the same organization or administrational unit. For example, there may be a directory node for each department in an organization, or a directory node from which all hosts can be found. Another directory node may be used as the starting point for naming all users, and so forth. The nodes in the administrational layer are relatively stable, although changes generally occur more frequently than to nodes in the global layer.

Finally, the **managerial layer** consists of nodes that may typically change regularly. For example, nodes representing hosts in the local network belong to this layer. For the same reason, the layer includes nodes representing shared files such as those for libraries or binaries. Another important class of nodes includes those that represent user-defined directories and files. In contrast to the global and administrational layer, the nodes in the managerial layer are maintained not only by system administrators, but also by individual end users of a distributed system.

To make matters more concrete, Fig. 4-6 shows an example of the partitioning of part of the DNS name space, including the names of files within an organization that can be accessed through the Internet, for example, Web pages and transferable files. The name space is divided into nonoverlapping parts, called zones in DNS (Mockapetris, 1987). A zone is a part of the name space that is implemented by a separate name server. Some of these zones are illustrated in Fig. 4-6.

If we take a look at availability and performance, name servers in each layer have to meet different requirements. High availability is especially critical for name servers in the global layer. If a name server fails, a large part of the name space will be unreachable because name resolution cannot proceed beyond the failing server.

Performance is somewhat subtle. Due to the low rate of change of nodes in the global layer, the results of lookup operations generally remain valid for a long time. Consequently, those results can be effectively cached (i.e., stored locally) by the clients. The next time the same lookup operation is performed, the results can be retrieved from the client’s cache instead of letting the name server return the results. As a result, name servers in the global layer do not have to respond quickly to a single lookup request. On the other hand, throughput may be important, especially in large-scale systems with millions of users.

---

4.1.3 The Implementation of a Name Space

A name space forms the heart of a naming service, that is, a service that allows users and processes to add, remove, and look up names. A naming service is implemented by name servers. If a distributed system is restricted to a local-area network, it is often feasible to implement a naming service by means of only a single name server. However, in large-scale distributed systems with many entities, possibly spread across a large geographical area, it is necessary to distribute the implementation of a name space over multiple name servers.

**Name Space Distribution**

Name spaces for a large-scale, possibly worldwide distributed system, are usually organized hierarchically. As before, assume such a name space has only a single root node. To effectively implement such a name space, it is convenient to
NAMING

The availability and performance requirements for name servers in the global layer can be met by replicating servers, in combination with client-side caching. As we discuss in Chap. 6, updates in this layer generally do not have to come into effect immediately, making it much easier to keep replicas consistent.

Availability for a name server in the administrative layer is primarily important for clients in the same organization as the name server. If the name server fails, many resources within the organization become unreachable because they cannot be looked up. On the other hand, it may be less important that resources in an organization are temporarily unreachable for users outside that organization.

With respect to performance, name servers in the administrative layer have similar characteristics as those in the global layer. Because changes to nodes do not occur very often, caching lookup results can be highly effective, making performance less critical. However, in contrast to the global layer, the administrative layer should take care that lookup results are returned within a few milliseconds, either directly from the server or from the client’s local cache. Likewise, updates should generally be processed quicker than those of the global layer. For example, it is unacceptable that an account for a new user takes hours to become effective.

These requirements can generally be met by using high-performance machines to run name servers. In addition, client-side caching should be applied, combined with replication for increased overall availability.

Availability requirements for name servers at the managerial level are generally less demanding. In particular, it often suffices to use a single (dedicated) machine to run name servers at the risk of temporary unavailability. However, performance is crucial. Users expect operations to take place immediately. Because updates occur regularly, client-side caching is often less effective, unless special measures are taken, which we discuss in Chap. 6.

<table>
<thead>
<tr>
<th>Item</th>
<th>Global</th>
<th>Administrative</th>
<th>Managerial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geographical scale of network</td>
<td>Worldwide</td>
<td>Organization</td>
<td>Department</td>
</tr>
<tr>
<td>Total number of nodes</td>
<td>Few</td>
<td>Many</td>
<td>Vast numbers</td>
</tr>
<tr>
<td>Responsiveness to lookups</td>
<td>Seconds</td>
<td>Milliseconds</td>
<td>Immediate</td>
</tr>
<tr>
<td>Update propagation</td>
<td>Lazy</td>
<td>Immediate</td>
<td>Immediate</td>
</tr>
<tr>
<td>Number of replicas</td>
<td>Many</td>
<td>None or few</td>
<td>None</td>
</tr>
<tr>
<td>Is client-side caching applied?</td>
<td>Yes</td>
<td>Yes</td>
<td>Sometimes</td>
</tr>
</tbody>
</table>

**Figure 4-7.** A comparison between name servers for implementing nodes from a large-scale name space partitioned into a global layer, an administrative layer, and a managerial layer.

A comparison between name servers at different layers is shown in Fig. 4-7. In distributed systems, name servers in the global and administrative layer are the most difficult to implement. Difficulties are caused by replication and caching, which are needed for availability and performance, but which also introduce consistency problems. Some of the problems are aggravated by the fact that caches and replicas are spread across a wide-area network, which introduces long communication delays thereby making synchronization even harder. Replication and caching are discussed extensively in Chap. 6.

**Implementation of Name Resolution**

The distribution of a name space across multiple name servers affects the implementation of name resolution. To explain the implementation of name resolution in large-scale name services, we assume that name servers are not replicated and that no client-side caches are used. Each client has access to a local name resolver, which is responsible for ensuring that the name resolution process is carried out. Referring to Fig. 4-6, assume the (absolute) path name

```
root::<nl, vu, cs, ftp, pub, globe, index.txt>
```

is to be resolved. Using a URL notation, this path name would correspond to `ftp://ftp.cs.vu.nl/pub/globe/index.txt`. There are now two ways to implement name resolution.

In **iterative name resolution**, a name resolver hands over the complete name to the root name server. It is assumed that the address where the root server can be
contacted, is well known. The root server will resolve the path name as far as it
can, and return the result to the client. In our example, the root server can resolve
only the label \( nl \), for which it will return the address of the associated name server.

At that point, the client passes the remaining path name (i.e., \( nl < vu, cs, ftp, pub, globe, index.txt > \)) to that name server. This server can resolve only the label \( vu \), and returns the address of the associated name server, along with the remaining
path name \( vu < cs, ftp, pub, globe, index.txt > \).

The client’s name resolver will then contact this next name server, which
responds by resolving the label \( cs \), and subsequently also \( ftp \), returning the address
of the FTP server along with the path name \( ftp < pub, globe, index.txt > \). The client
then contacts the FTP server, requesting it to resolve the last part of the original
path name. The FTP server will subsequently resolve the labels \( pub, globe, \) and
\( index.txt \), and transfer the requested file (in this case using FTP). This process
of iterative name resolution is shown in Fig. 4-8. (The notation \( # < cs > \) is used to indicate the address of the server responsible for handling the node referred to by
\( < cs > \).)

![Diagram of iterative name resolution](image)

**Figure 4-8.** The principle of iterative name resolution.

In practice, the last step, namely contacting the FTP server and requesting it
to transfer the file with path name \( ftp < pub, globe, index.txt > \), is carried out
separately by the client process. In other words, the client would normally hand
only the path name \( root < nl, vu, cs, ftp > \) to the name resolver, from which it
would expect the address where it can contact the FTP server, as is also shown in
Fig. 4-8.

An alternative to iterative name resolution is to use recursion during name
resolution. Instead of returning each intermediate result back to the client’s name
resolver, with **recursive name resolution**, a name server passes the result to the
next name server it finds. So, for example, when the root name server finds the

address of the name server implementing the node named \( nl \), it requests that name
server to resolve the path name \( nl < vu, cs, ftp, pub, globe, index.txt > \). Using
recursive name resolution as well, this next server will resolve the complete path
and eventually return the file \( index.txt \) to the root server, which, in turn, will pass
that file to the client’s name resolver.

Recursive name resolution is presented in Fig. 4-9. As in iterative name resolu-
tion, the last resolution step, namely contacting the FTP server and asking it to
transfer the indicated file, is generally carried out as a separate process by the client.

![Diagram of recursive name resolution](image)

**Figure 4-9.** The principle of recursive name resolution.

The main drawback of recursive name resolution is that it puts a higher perform-
ance demand on each name server. Basically, a name server is required to
handle the complete resolution of a path name, although it may do so in coopera-
tion with other name servers. This additional burden is generally so high that
name servers in the global layer of a name space support only iterative name reso-
lution.

There are two important advantages to recursive name resolution. The first
advantage is that caching results is more effective compared to iterative name
resolution. The second advantage is that communication costs may be reduced.
To explain these advantages, assume that a client’s name resolver will accept path
names referring only to nodes in the global or administrative layer of the name
space. To resolve that part of a path name that corresponds to nodes in the
managerial layer, a client will separately contact the name server returned by its
name resolver, as we discussed above.

Recursive name resolution allows each name server to gradually learn the
address of each name server responsible for implementing lower-level nodes. As a
result, caching can be effectively used to enhance performance. For example,
when the root server is requested to resolve the path name \( root < nl, vu, cs, ftp > \), it
will eventually get the address of the name server implementing the node referred
to by that path name. To come to that point, the name server for the nl node has to look up the address of the name server for the vu node, whereas the latter has to look up the address of the name server handling the cs node.

Because changes to nodes in the global and administrative layer do not occur often, the root name server can effectively cache the returned address. Moreover, because the address is also returned, by recursion, to the name server responsible for implementing the vu node and to the one implementing the nl node, it might as well be cached at those servers as well.

Likewise, the results of intermediate name lookups can also be returned and cached. For example, the server for the nl node will have to look up the address of the vu node server. That address can be returned to the root server when the nl server returns the result of the original name lookup. A complete overview of the resolution process, and the results that can be cached by each name server, is shown in Fig. 4-10.

<table>
<thead>
<tr>
<th>Server for node</th>
<th>Should resolve</th>
<th>Looks up</th>
<th>Passes to child</th>
<th>Receives and caches</th>
<th>Returns to requester</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs</td>
<td>&lt;ftp&gt;</td>
<td>#&lt;ftp&gt;</td>
<td>---</td>
<td>#&lt;ftp&gt;</td>
<td>#&lt;ftp&gt;</td>
</tr>
<tr>
<td>vu</td>
<td>&lt;cs,ftp&gt;</td>
<td>#&lt;cs&gt;</td>
<td>#&lt;ftp&gt;</td>
<td>#&lt;cs&gt;</td>
<td>#&lt;cs&gt;, ftp&gt;</td>
</tr>
<tr>
<td>nl</td>
<td>&lt;vu,cs,ftp&gt;</td>
<td>#&lt;vu&gt;</td>
<td>#&lt;cs&gt;, ftp&gt;</td>
<td>#&lt;vu&gt;</td>
<td>#&lt;vu,cs, ftp&gt;</td>
</tr>
<tr>
<td>root</td>
<td>&lt;nl,vu,cs,ftp&gt;</td>
<td>#&lt;nl&gt;</td>
<td>#&lt;vu,cs, ftp&gt;</td>
<td>#&lt;nl&gt;</td>
<td>#&lt;nl,vu,cs, ftp&gt;</td>
</tr>
</tbody>
</table>

**Figure 4-10.** Recursive name resolution of <nl, vu, cs, ftp>. Name servers cache intermediate results for subsequent lookups.

The benefit of this approach is that, eventually, lookup operations can be handled extremely efficiently. For example, suppose that another client later requests resolution of the path name root:<nl, vu, cs, flits>. This name is passed to the root, which can immediately forward it to the name server for the cs node, and request it to resolve the remaining path name cs:<flits>.

With iterative name resolution, caching is necessarily restricted to the client's name resolver. Consequently, if a client A requests the resolution of a name, and another client B later requests that same name to be resolved, name resolution will have to pass through the same name servers as was done for client A. As a compromise, many organizations use a local, intermediate name server that is shared by all clients. This local name server handles all naming requests and caches results. Such an intermediate server is also convenient from a management point of view. For example, only that server needs to know where the root name server is located; other machines do not require this information.

The second advantage of recursive name resolution is that it is often cheaper with respect to communication. Again, consider the resolution of the path name root:<nl, vu, cs, ftp> and assume the client is located in San Francisco. Assuming that the client knows the address of the server for the nl node, with recursive name resolution, communication follows the route from the client's host in San Francisco to the nl server in The Netherlands, shown as R1 in Fig. 4-11. From there on, communication is subsequently needed between the nl server and the name server of the Vrije Universiteit on the university campus in Amsterdam, The Netherlands. This communication is shown as R2. Finally, communication is needed between the vu server and the name server in the Computer Science Department, shown as R3. The route for the reply is the same, but in the opposite direction. Clearly, communication costs are dictated by the message exchange between the client's host and the nl server.

In contrast, with iterative name resolution, the client's host has to communicate separately with the nl server, the vu server, and the cs server, of which the total costs may be roughly three times that of recursive name resolution. The arrows in Fig. 4-11 labeled I1, I2, and I3 show the communication path for iterative name resolution.

**Figure 4-11.** The comparison between recursive and iterative name resolution with respect to communication costs.

**4.1.4 Example: The Domain Name System**

One of the largest distributed naming services in use today, is the Internet Domain Name System (DNS). DNS is primarily used for looking up host addresses and mail servers. In the following pages, we concentrate on the organization of the DNS name space, and the information stored in its nodes. Also, we take a closer look at the actual implementation of DNS. More information can be found in (Mockapetris, 1987; Albitz and Liu, 1998).
The DNS Name Space

The DNS name space is hierarchically organized as a rooted tree. A label is a case-insensitive string made up of alphanumeric characters. A label has a maximum length of 63 characters; the length of a complete path name is restricted to 255 characters. The string representation of a path name consists of listing its labels, starting with the rightmost one, and separating the labels by a dot ("."). The root is represented by a dot. So, for example, the path name root: <nl, vu, cs, flits>, is represented by the string flits.cs.vu.nl, which includes the rightmost dot to indicate the root node. We generally omit this dot for readability.

Because each node in the DNS name space has exactly one incoming edge (with the exception of the root node, which has no incoming edges), the label attached to a node’s incoming edge is also used as the name for that node. A subtree is called a domain; a path name to its root node is called a domain name. Note that, just like a path name, a domain name can be either absolute or relative.

The contents of a node is formed by a collection of resource records. There are different types of resource records, of which the most important ones are shown in Fig. 4-12.

<table>
<thead>
<tr>
<th>Type of record</th>
<th>Associated entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA</td>
<td>Zone</td>
<td>Holds information on the represented zone</td>
</tr>
<tr>
<td>A</td>
<td>Host</td>
<td>Contains an IP address of the host this node represents</td>
</tr>
<tr>
<td>MX</td>
<td>Domain</td>
<td>Refers to a mail server to handle mail addressed to this node</td>
</tr>
<tr>
<td>SRV</td>
<td>Domain</td>
<td>Refers to a server handling a specific service</td>
</tr>
<tr>
<td>NS</td>
<td>Zone</td>
<td>Refers to a name server that implements the represented zone</td>
</tr>
<tr>
<td>CNAME</td>
<td>Node</td>
<td>Symbolic link with the primary name of the represented node</td>
</tr>
<tr>
<td>PTR</td>
<td>Host</td>
<td>Contains the canonical name of a host</td>
</tr>
<tr>
<td>HINFO</td>
<td>Host</td>
<td>Holds information on the host this node represents</td>
</tr>
<tr>
<td>TXT</td>
<td>Any kind</td>
<td>Contains any entity-specific information considered useful</td>
</tr>
</tbody>
</table>

Figure 4-12. The most important types of resource records forming the contents of nodes in the DNS name space.

A node in the DNS name space often will represent several entities at the same time. For example, a domain name such as vu.nl is used to represent a domain and a zone. In this case, the domain is implemented by several zones.

An SOA (start of authority) resource record contains information such as an e-mail address of the system administrator responsible for the represented zone, the name of the host where data on the zone can be fetched, and so on.

An A (address) record, represents a particular host in the Internet. The A record contains an IP address for that host to allow communication. If a host has several IP addresses, as is the case with multi-homed machines, the node will contain an A record for each address.

An important type of resource record is the MX (mail exchange) record, which is essentially a symbolic link to a node representing a mail server. For example, the node representing the domain cs.vu.nl has an MX record containing the name sep/hr.cs.vu.nl, which refers to a mail server. That server will handle all incoming mail addressed to users in the cs.vu.nl domain. There may be several MX records stored in a node.

Related to MX records are SRV records, which contain the name of a server for a specific service. SRV records are defined in (Vixie, 1996). The service itself is identified by means of a name along with the name of a protocol. For example, the Web server in the cs.vu.nl domain could be named by means of an SRV record such as http.tep.cs.vu.nl. This record would then refer to the actual name of the server (which is soling.cs.vu.nl).

Nodes that represent a zone, contain one or more NS (name server) records. Like MX records, an NS record contains the name of a name server that implements the zone represented by the node. In principle, each node in the name space can store an NS record referring to the name server that implements it. However, as we discuss below, the implementation of the DNS name space is such that only nodes representing zones need to store NS records.

DNS distinguishes aliases from what are called canonical names. Each host is assumed to have a canonical, or primary name. An alias is implemented by means of node storing a CNAME record containing the canonical name of a host. The name of the node storing such a record is thus the same as a symbolic link, as was shown in Fig. 4-3.

DNS maintains an inverse mapping of IP addresses to host names by means of PTR (pointer) records. To accommodate the lookups of host names when given only an IP address, DNS maintains a domain named in-addr.arpa, which contains nodes that represent Internet hosts and which are named by the IP address of the represented host. For example, host www.cs.vu.nl has IP address 130.37.24.11. DNS creates a node named 11.24.37.130.in-addr.arpa, which is used to store the canonical name of that host (which happens to be soling.cs.vu.nl) in a PTR record.

The last two record types are HINFO records and TXT records. An HINFO (host info) record is used to store additional information on a host such as its machine type and operating system. In a similar fashion, TXT records are used for any other kind of data that a user finds useful to store about the entity represented by the node.

DNS Implementation

The implementation of DNS is very similar to what has been described in the previous section. In essence, the DNS name space can be divided into a global layer and an administrative layer as shown in Fig. 4-6. The managerial layer,
which is generally formed by local file systems, is formally not part of DNS and is therefore also not managed by it.

Each zone is implemented by a name server, which is virtually always replicated for availability. Updates for a zone are normally handled by the primary name server. Updates take place by modifying the DNS database local to the primary server. Secondary name servers do not access the database directly, but, instead, request the primary server to transfer its content. The latter is called a zone transfer in DNS terminology.

A DNS database is implemented as a (small) collection of files, of which the most important one contains all the resource records for all the nodes in a particular zone. This approach allows nodes to be simply identified by means of their domain name, by which the notion of a node identifier reduces to an (implicit) index into a file.

To better understand these implementation issues, Fig. 4.13 shows part of the file that contains most of the information for the cs.vu.nl domain. (It should be noted that the file has been edited for presentational purposes.) The file shows the contents of eight different nodes that are part of the cs.vu.nl domain, where each node is identified by means of its domain name.

The node cs.vu.nl represents the domain as well as the zone. Its SOA resource record contains specific information on the validity of this file, which will not concern us further. There are three name servers for this zone, referred to by their canonical host names in the NS records. The TXT record is used to give some additional information on this zone, but cannot be automatically processed by any name server. Furthermore, there are three mail servers that can handle incoming mail addressed to users in this domain. The number preceding the name of a mail server specifies a selection priority. A sending mail server should always first attempt to contact the mail server with the lowest number, in this example, zephyr.cs.vu.nl.

The host star.cs.vu.nl operates as a name server for this zone. Name servers are critical to any naming service. What can be seen about this name server, is that additional robustness has been created by giving two separate network interfaces, each represented by a separate A resource record. In this way, the effects of a broken network link can be somewhat circumvented.

The next four lines give the necessary information about the mail server. Note that this mail server is also backed up by another mail server, whose path is tornado.cs.vu.nl.

The next six lines show a typical configuration in which the department’s Web server, as well as the FTP server are implemented by a single machine, called soling.cs.vu.nl. By executing both servers on the same machine (and essentially using that machine only for Internet services), system management becomes easier. For example, both servers will have the same view of the file system, and for efficiency, part of the file system may be implemented on soling.cs.vu.nl. This approach is often applied in the case of WWW and FTP services.

<table>
<thead>
<tr>
<th>Name</th>
<th>Record type</th>
<th>Record value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs.vu.nl</td>
<td>SOA</td>
<td>star (1999121502,7200,3600,2419200,86400)</td>
</tr>
<tr>
<td>cs.vu.nl</td>
<td>NS</td>
<td>star.cs.vu.nl</td>
</tr>
<tr>
<td>cs.vu.nl</td>
<td>NS</td>
<td>top.cs.vu.nl</td>
</tr>
<tr>
<td>cs.vu.nl</td>
<td>NS</td>
<td>solo.cs.vu.nl</td>
</tr>
<tr>
<td>cs.vu.nl</td>
<td>TXT</td>
<td>&quot;Vrije Universiteit - Math. &amp; Comp. Sc.&quot;</td>
</tr>
<tr>
<td>cs.vu.nl</td>
<td>MX</td>
<td>1 zephyr.cs.vu.nl</td>
</tr>
<tr>
<td>cs.vu.nl</td>
<td>MX</td>
<td>2 tornado.cs.vu.nl</td>
</tr>
<tr>
<td>cs.vu.nl</td>
<td>MX</td>
<td>3 star.cs.vu.nl</td>
</tr>
<tr>
<td>star.cs.vu.nl</td>
<td>HINFO</td>
<td>Sun Unix</td>
</tr>
<tr>
<td>star.cs.vu.nl</td>
<td>MX</td>
<td>1 star.cs.vu.nl</td>
</tr>
<tr>
<td>star.cs.vu.nl</td>
<td>MX</td>
<td>10 zephyr.cs.vu.nl</td>
</tr>
<tr>
<td>star.cs.vu.nl</td>
<td>A</td>
<td>192.31.24.6</td>
</tr>
<tr>
<td>star.cs.vu.nl</td>
<td>A</td>
<td>192.31.231.42</td>
</tr>
<tr>
<td>zephyr.cs.vu.nl</td>
<td>HINFO</td>
<td>Sun Unix</td>
</tr>
<tr>
<td>zephyr.cs.vu.nl</td>
<td>MX</td>
<td>1 zephyr.cs.vu.nl</td>
</tr>
<tr>
<td>zephyr.cs.vu.nl</td>
<td>MX</td>
<td>2 tornado.cs.vu.nl</td>
</tr>
<tr>
<td>zephyr.cs.vu.nl</td>
<td>A</td>
<td>192.31.231.66</td>
</tr>
<tr>
<td><a href="http://www.cs.vu.nl">www.cs.vu.nl</a></td>
<td>CNAME</td>
<td>soling.cs.vu.nl</td>
</tr>
<tr>
<td>ftp.cs.vu.nl</td>
<td>CNAME</td>
<td>soling.cs.vu.nl</td>
</tr>
<tr>
<td>soling.cs.vu.nl</td>
<td>HINFO</td>
<td>Sun Unix</td>
</tr>
<tr>
<td>soling.cs.vu.nl</td>
<td>MX</td>
<td>1 soling.cs.vu.nl</td>
</tr>
<tr>
<td>soling.cs.vu.nl</td>
<td>MX</td>
<td>10 zephyr.cs.vu.nl</td>
</tr>
<tr>
<td>soling.cs.vu.nl</td>
<td>A</td>
<td>192.31.24.11</td>
</tr>
<tr>
<td>laser.cs.vu.nl</td>
<td>HINFO</td>
<td>PC MS-DOS</td>
</tr>
<tr>
<td>laser.cs.vu.nl</td>
<td>A</td>
<td>192.30.30.32</td>
</tr>
<tr>
<td>vuc_ING.cs.vu.nl</td>
<td>PTR</td>
<td>0.26.37.130.in-addr.arpa</td>
</tr>
<tr>
<td>vuc_ING.cs.vu.nl</td>
<td>A</td>
<td>192.37.26.0</td>
</tr>
</tbody>
</table>

Figure 4.13. An excerpt from the DNS database for the zone cs.vu.nl.

The following two lines show information on one of the laser printers connected to the local network. The last two lines illustrate the inverse mapping from addresses to canonical names. In this case, the name of the department’s supercomputer can be looked up by its address in the in-addr.arpa domain.

Because the cs.vu.nl domain is implemented as a single zone, Fig. 4.13 does not include references to other zones. For example, when referring to nodes in a subdomain that are implemented in a different zone is shown in Fig. 4.14. What needs to be done is to specify a name server for the subdomain, by simply giving its domain name and IP address. When resolving a name for a node that lies in the cs.vu.nl domain, name resolution will continue at a certain point by reading the DNS database stored by the name server for the cs.vu.nl domain.
4.1.5 Example: X.500

DNS is an example of a traditional naming service: when given a (possibly hierarchically structured) name, DNS resolves the name to a node in the naming graph and returns the content of that node in the form of a resource record. In this sense, DNS is comparable to a telephone book for looking up phone numbers.

A different approach is taken by what are called directory services. A directory service is a special kind of naming service in which a client can look for an entity based on a description of properties instead of a full name. This approach is very similar to the way people use the yellow pages when they need, for example, a person to repair a broken window. In that case, a user may look under the heading ‘Window repair’ to obtain a list of (names of) firms that replace windows.

In this section, we take a brief look at the OSI X.500 directory service. Although this directory service has been available for over a decade, it is only recently gaining more popularity as lightweight versions are being implemented as Internet services. Detailed information on X.500 can be found in (Chadwick, 1994; Radicati, 1994). Practical information on various directory services, including X.500, can be found in (Sheresh and Sheresh, 2000).

The X.500 Name Space

Conceptually, an X.500 directory service consists of a number of records, usually referred to as directory entries. A directory entry in X.500 is comparable to a resource record in DNS. Each record is made up of a collection of (attribute, value) pairs, where each attribute has an associated type. A distinction is made between single-valued attributes and multiple-valued attributes. The latter typically represent arrays and lists. As an example, a simple directory entry identifying the network addresses of some general servers from Fig. 4-13 is shown in Fig. 4-15.

In our example, we have used a naming convention described in the X.500 standards, which applies to the first five attributes. The attributes Organization and OrganizationalUnit describe, respectively, the organization and the department associated with the data that are stored in the record. Likewise, the attributes Locality and Country provide additional information on where the entry is stored. The CommonName attribute is often used as an (ambiguous) name to identify an entry within a limited part of the directory. For example, the name “Main servers” may be enough to find our example entry given the specific values for the other four attributes Country, Locality, Organization, and OrganizationalUnit.

In our example, only attribute Mail_Servers has multiple values associated with it. All other attributes have only a single value.

The collection of all directory entries in an X.500 directory service is called a Directory Information Base (DIB). An important aspect of a DIB is that each record is uniquely named so that it can be looked up. Such a globally unique name appears as a sequence of naming attributes in each record. Each naming attribute is called a Relative Distinguished Name, or RDN for short. In our example in Fig. 4-15, the first five attributes are all naming attributes. Using the conventional abbreviations for representing naming attributes in X.500 as shown in Fig. 4-15, the attributes Country, Organization, and OrganizationalUnit could be used to form the globally unique name

```
```

analogous to the DNS name 

As in DNS, the use of globally unique names by listing RDNs in sequence, leads to a hierarchy of the collection of directory entries, which is referred to as a Directory Information Tree (DIT). A DIT essentially forms the naming graph of an X.500 directory service in which each node represents a directory entry. In addition, a node may also act as a directory in the traditional sense, in that there may be several children for which the node acts as parent. To explain, consider the naming graph as partly shown in Fig. 4-16(a).

Node N corresponds to the directory entry shown in Fig. 4-15. At the same time, this node acts as a parent to a number of other directory entries that have an additional naming attribute Host_Name that is used as an RDN. For example, such entries may be used to represent hosts as shown in Fig. 4-16(b).
SEC. 4.1 NAMING ENTITIES  209

X.500 Implementation

Implementing an X.500 directory service proceeds in much the same way as implementing a naming service such as DNS, except that X.500 supports more lookup operations as we will discuss shortly. When dealing with a large-scale directory, the DIT is usually partitioned and distributed across several servers, known as Directory Service Agents (DSA) in X.500 terminology. Each part of a partitioned DIT thus corresponds to a zone in DNS. Likewise, each DSA behaves very much the same as a normal name server, except that it implements a number of typical directory services, such as advanced search operations.

Clients are represented by what are called Directory User Agents, or simply DUAAs. A DUA is similar to a name resolver in traditional naming services. A DUA exchanges information with a DSA according to a standardized access protocol.

What makes an X.500 implementation different from a DNS implementation are the facilities for searching through a DIT. In particular, facilities are provided to search for a directory entry given a set of criteria that attributes of the searched entries should meet. For example, suppose that we want a list of all main servers at the Vrije Universiteit. Using the notation defined in (Howes, 1997), such a list can be returned using a search operation such as

answer = search("(&C=NL)(O=Vrije Universiteit)(OU=*)(CN=Main server*)")

In this example, we have specified that the place to look for main servers is the organization named Vrije Universiteit in country NL, but that we are not interested in a particular organizational unit. However, each returned result should have the CN attribute equal to Main server.

An important observation is that searching in a directory service is generally an expensive operation. For example, to find all main servers at the Vrije Universiteit requires searching all entries at each department and combining the results in a single answer. In other words, we will generally need to access several leaf nodes of a DIT in order to get an answer. In practice, this also means that several DSAs need to be accessed. In contrast, naming services can often be implemented in such a way that a lookup operation requires accessing only a single leaf node.

Staying in line with many other OSI protocols, accessing an X.500 directory service in the Internet, a simplified protocol has been devised, known as the Lightweight Directory Access Protocol (LDAP).

LDAP is an application-level protocol that is implemented directly on top of TCP (Yeong et al., 1995; Wahl et al., 1997), which alone contributes to its simplicity compared to the official OSI access protocol. In addition, parameters of lookup and update operations can simply be passed as strings, instead of using a separate encoding as required by OSI's protocol. LDAP is gradually becoming a de facto standard for Internet-based directory services. It is being integrated into

---

**Figure 4-16.** (a) Part of a directory information tree. (b) Two directory entries having Host_Name as RDN.

A node in an X.500 naming graph can thus simultaneously represent a directory in the traditional sense as we discussed previously, as well as an X.500 record. This distinction is supported by two different lookup operations. The read operation is used to read a single record given its path name in the DIT. In contrast, the list operation is used to list the names of all outgoing edges of a given node in the DIT. Each name corresponds to a child node of the given node. Note that the list operation does not return any records; it merely returns names. In other words, calling read with as input the name

```
/C=NL/O=Vrije Universiteit/OU=Math. & Comp. Sc./CN=Main server
```

will return the record shown in Fig. 4-15, whereas calling list will return the names star and zephyr from the entries shown in Fig. 4-16(b) as well as the names of other hosts that have been registered in a similar way.
4.2 LOCATING MOBILE ENTITIES

The naming services discussed so far, are primarily used for naming entities that have a fixed location. By their nature, traditional naming systems are not well suited for supporting name-to-address mappings that change regularly, as is the case with mobile entities. These issues are discussed in this section, along with solutions to locating mobile entities.

4.2.1 Naming versus Locating Entities

As we explained in the previous section, entities are named so that they can be looked up and subsequently accessed. Three types of names were distinguished: human-friendly names, identifiers, and addresses. Because distributed systems are built to be used by humans and because it is necessary to have an entity’s address to access it, virtually all naming systems maintain a mapping of human-friendly names to addresses.

As we also explained, to effectively implement a large-scale name space such as in DNS, it is useful to partition the name space into three layers. The global layer and the administrative layer are characterized by the fact that names do not change often. More precisely, the content of nodes in those parts of the name space is relatively stable. As a consequence, an efficient implementation can be achieved through replication and caching.

The contents of nodes in the managerial layer change often. Therefore, performance of updates and lookups becomes important. In practice, performance demands can be met by implementing nodes on a local, high-performance name server.

Let us take a closer look at which assumptions are actually made, and why this approach toward implementing large-scale naming systems works. First again consider looking up the address of the (remote) host ftp.cs.vu.nl. By assuming that the content of nodes in the global and administrative layer are stable, it is probable that the client can find the address of the name server for the cs.vu.nl domain in its local cache. Consequently, only a single request needs to be sent to that name server to find the address of ftp.cs.vu.nl.

Next, consider updating the address of ftp.cs.vu.nl, for instance, because the FTP server is to be moved to a different machine. As long as the server is moved to a machine within the cs.vu.nl domain, the update can be done efficiently. In that case, only the DNS database of the name server for cs.vu.nl will have to be changed. Lookups will be as efficient as they were before.

Consequently, by assuming that nodes in the global and administrative layer do not change often, and also by assuming that updates are generally restricted to a single name server, naming systems such as DNS can be made highly efficient.

Now consider what happens if ftp.cs.vu.nl were to move to a machine named ftp.cs.unisa.edu.au, which lies in a completely different domain. The first observation to make, is that the name ftp.cs.vu.nl should preferably not change, as it can be expected that many applications and users will have symbolic links to it. In other words, the name is presumably used as an identifier. Changing it will cause all links to it to become invalid.

There are now basically two solutions. One solution is to record the address of the new machine in the DNS database for cs.vu.nl. An alternative solution is to keep the name of the new machine, instead of its address, effectively turning ftp.cs.vu.nl into a symbolic link. Both solutions have serious drawbacks.

Let us first consider recording the address of the new machine. Clearly, lookup operations are not affected by this approach. However, whenever ftp.cs.vu.nl moves once again to another machine, its entry in the DNS database in cs.vu.nl will have to be updated as well. It is important to note that this update is no longer a local operation but may actually take hundreds of milliseconds to complete. In other words, this approach violates the assumption that operations on nodes in the managerial layer are efficient.

The main drawback of using a symbolic link is that lookup operations become less efficient. In effect, each lookup is split into two steps:

1. Find the name of the new machine.
2. Look up the address associated with that name.

However, if ftp.cs.vu.nl is to move again, say to ftp.cs.berkeley.edu, we can perform a local update operation by turning the name ftp.cs.unisa.edu.au into a symbolic link to ftp.cs.berkeley.edu, and leave the entry in the DNS database for cs.vu.nl as it was. The drawback is that we have added another step to the lookup operations.

For highly mobile entities, matters become only worse. Each time an entity moves, either a nonlocal update operation needs to be performed or another step is added to lookup operations.

Another serious problem with the approaches mentioned so far, is that the name ftp.cs.vu.nl is not allowed to change. Consequently, it becomes extremely important to choose names that can be expected not to change for the lifetime of the entity they represent. Moreover, that name cannot be used for any other entity. In practice, choosing such names, especially for very long-lived entities, is difficult, as is demonstrated by naming in the World Wide Web. In particular, many entities are known under different names, and all these names should remain valid, that is, always refer to the same entity, even in the face of mobility.
For these reasons, traditional naming services such as DNS cannot cope well with mobile entities, and different solutions are needed. In essence, problems arise because traditional naming services maintain a direct mapping between human-friendly names and the addresses of entities. Each time a name or an address changes, the mapping needs to change as well, as shown in Fig. 4-17(a).

![Diagram showing naming and location services](image)

**Figure 4-17.** (a) Direct, single-level mapping between names and addresses. (b) Two-level mapping using identifiers.

A better solution is to separate naming from locating entities by introducing identifiers, as shown in Fig. 4-17(b). Recall that an identifier never changes, that each entity has exactly one identifier, and that an identifier is never assigned to a different entity (Wieringa and de Jonge, 1995). In general, an identifier is not intended to have a human-friendly representation. In other words, it is optimized for machine processing only.

When looking up an entity by means of a naming service, that service returns an identifier. The identifier can be stored locally for as long as needed because it is known never to refer to a different entity, nor will it ever change. Under which name it is stored locally, is not important. Consequently, when that identifier is needed the next time, it can simply be retrieved locally without having to look it up by means of the naming service.

Locating an entity is handled by means of a separate location service. A location service essentially accepts an identifier as input, and returns the current address of the identified entity. If multiple copies exist, then multiple addresses may be returned. In this section, we concentrate solely on the problem of implementing efficient location services.

### 4.2.2 Simple Solutions

We first consider two simple solutions for locating an entity. Both solutions are applicable only to local-area networks. Nevertheless, in that environment, they often do the job well, making their simplicity particularly attractive.
Forwarding Pointers

Another popular approach to locating mobile entities is to make use of forwarding pointers (Fowler, 1985). The principle is simple: when an entity moves from A to B, it leaves behind a reference to its new location at B. The main advantage of this approach is its simplicity: as soon as an entity has been located, for example by using a traditional naming service, a client can look up the current address by following the chain of forwarding pointers.

There are also a number of important drawbacks. First, if no special measures are taken, a chain can become so long that locating an entity is prohibitively expensive. Second, all intermediate locations in a chain will have to maintain their part of the chain of forwarding pointers as long as needed. A third, and related drawback, is the vulnerability to broken links. As soon as a forwarding pointer is lost for whatever reason, the entity can no longer be reached. An important issue is, therefore, to keep chains relatively short, and to ensure that forwarding pointers are robust.

To better understand how forwarding pointers work, consider their use with respect to distributed objects. Following the approach in SSP chains (Shapiro et al., 1992), each forwarding pointer is implemented as a (proxy, skeleton) pair as shown in Fig. 4-18 [in SSP, a proxy is called a stub and a skeleton a scion, leading to (stub, scion) pairs, which explains the abbreviation SSP]. A skeleton (i.e., the server-side stub) contains either a local reference to the actual object or a local reference to a proxy (i.e., the client-side stub) for that object. To emphasize that skeletons act as entry items for remote references, and proxies as exit items, we use the notation as shown in Fig. 4-18.

![Figure 4-18. The principle of forwarding pointers using (proxy, skeleton) pairs.](image)

Whenever an object moves from address space A to B, it leaves behind a proxy in its place in A and installs a skeleton that refers to it in B. An interesting aspect of this approach is that migration is completely transparent to a client. The only thing the client sees of an object, is a proxy. How, and to which location that proxy forwards its invocation is hidden from the client. Also note that this use of forwarding pointers is not the same as looking up an address. Instead, a client’s request is forwarded along the chain to the actual object.

To short-cut a chain of (proxy, skeleton) pairs, an invocation carries the identification of the proxy from where that invocation was initiated. A proxy identification consists of the client’s transport-level address, combined with a locally generated number to identify the proxy. When the invocation reaches the object at its current location, a response is sent back to the proxy where the invocation was initiated. The current location is piggybacked with this response, and the proxy adjusts its companion skeleton to the one in the object’s current location. This principle is shown in Fig. 4-19.

![Figure 4-19. Redirecting a forwarding pointer by storing a shortcut in a proxy.](image)

There is a trade-off between sending the response directly to the initiating proxy, or along the reverse path of forwarding pointers. In the former case, communication is faster because fewer processes may need to be passed. On the other hand, only the initiating proxy can be adjusted, whereas sending the response along the reverse path allows adjustment of all intermediate proxies.

When a skeleton is no longer referred to by any proxy, it can be removed. How this can be done automatically, is discussed in Sec. 4.3.

As we explained in Chap. 2, object references in distributed-object systems can be implemented as proxies that are passed as parameters in method invocations. This scheme still works with forwarding pointers. Suppose that process $P_1$ in Fig. 4-18 passes its reference to object $O$ to process $P_2$. Reference passing is done by installing a copy $p'$ of proxy $p$ in the address space of process $P_2$. Proxy $p'$ refers to the same skeleton as $p$, so that the forwarding invocation mechanism works the same as before.

Problems arise when a process in a chain of (proxy, skeleton) pairs crashes or becomes otherwise unreachable. Several solutions are possible. One possibility, as
followed in Emerald (Jul et al., 1988) and in the LII system (Black and Artsy, 1990), is to let the machine where an object was created (called the object’s home location), always keep a reference to its current location. That reference is stored and maintained in a fault-tolerant way. When a chain is broken, the object’s home location is asked where the object is now. To allow an object’s home location to change, a traditional naming service can be used to record the current home location. Such home-based approaches are discussed next.

### 4.2.3 Home-Based Approaches

The use of broadcasting and forwarding pointers imposes scalability problems. Broadcasting or multicasting is difficult to implement efficiently in large-scale networks whereas long chains of forwarding pointers introduce performance problems and are susceptible to broken links.

A popular approach to supporting mobile entities in large-scale networks, is to introduce a home location, which keeps track of the current location of an entity. Special techniques may be applied to safeguard against network or process failures. In practice, the home location is often chosen to be the place where an entity was created.

The home-based approach is used as a fall-back mechanism for location services based on forwarding pointers, as discussed above. Another example where the home-based approach is followed, is in Mobile IP (Perkins, 1997). Each mobile host uses a fixed IP address. All communication to that IP address is initially directed to the mobile host’s home agent. This home agent is located on the local-area network corresponding to the network address contained in the mobile host’s IP address. Whenever the mobile host moves to another network, it requests a temporary address that it can use for communication. This care-of address is registered at the home agent.

When the home agent receives a packet for the mobile host, it looks up the host’s current location. If the host is on the current local network, the packet is simply forwarded. Otherwise, it is tunneled to the host’s current location, that is, wrapped as data in an IP packet and sent to the care-of address. At the same time, the sender of the packet is informed of the host’s current location. This principle is shown in Fig. 4-20. Note that the IP address is effectively used as an identifier for the mobile host.

Fig. 4-20 also illustrates another drawback of home-based approaches in large-scale networks. To communicate with a mobile entity, a client first has to contact the home, which may be at a completely different location than the entity itself. The result is an increase in communication latency.

A solution applied to mobile telephony is to use a two-tiered scheme (Mohan and Jain, 1994). When setting up a connection to a mobile entity, a client first checks a local registry to see whether the mobile entity is available locally. If not, the entity’s home location is contacted to find the current location. Below, we discuss an extension to this scheme that expands across multiple hierarchical layers.

Another important drawback of the home-based approach is the use of a fixed home location. For one thing, it must be ensured that the home location always exists. Otherwise, contacting the entity will become impossible. Problems are aggravated when a long-lived entity decides to move permanently to a completely different part of the network than where its home is located. In that case, it would have been better to let the home move along with the host.

A solution to this problem is to register the home at a traditional naming service and to let a client first look up the location of the home. Because the home location can be assumed to be relatively stable, that location can be effectively cached after it has been looked up.

### 4.2.4 Hierarchical Approaches

The two-tiered home-based approach to locating entities can be generalized to multiple layers. In this section, we first discuss a general approach to a hierarchical location scheme, after which a number of optimizations are presented. The approach we present is based on the Globe location service, described in (van Steen et al., 1998b). This is a general-purpose location service that is representative of many hierarchical location services proposed for what are called Personal Communication Systems (Pitoura and Samaras, 2001; Wang, 1993).
General Mechanism

In a hierarchical scheme, a network is divided into a collection of domains, very similar to the hierarchical organization of DNS. There is a single top-level domain that spans the entire network. Each domain can be subdivided into multiple, smaller subdomains. A lowest-level domain, called a leaf domain, typically corresponds to a local-area network in a computer network or a cell in a mobile telephone network.

Also analogous to DNS and other hierarchical naming systems, each domain has an associated directory node dir(D) that keeps track of the entities in that domain. This leads to a tree of directory nodes. The directory node of the top-level domain, called the root (directory) node, knows about all entities. This general organization of a network into domains and directory nodes is illustrated in Fig. 4-21.

To keep track of the whereabouts of an entity, each entity currently located in a domain D, is represented by a location record in the directory node dir(D). A location record for entity E in the directory node N for a leaf domain D contains the entity's current address in that domain. In contrast, the directory node N for the next higher-level domain D' that contains D, will have a location record for E containing only a pointer to N. Likewise, the parent node of N will store a location record for E containing only a pointer to N. Consequently, the root node will have a location record for each entity, where each location record stores a pointer to the directory node of the next lower-level subdomain where that record’s associated entity is currently located.

An entity may have multiple addresses, for example if it is replicated. If an entity has an address in leaf domain D1 and D2 respectively, then the directory node of the smallest domain containing both D1 and D2, will have two pointers, one for each subdomain containing an address. This leads to the general organization of the tree as shown in Fig. 4-22.

![diagram](image_url)

**Figure 4-22.** An example of storing information of an entity having two addresses in different leaf domains.

Let us now consider how a lookup operation proceeds in such a hierarchical location service. As is shown in Fig. 4-23, a client wishing to locate an entity E, issues a lookup request to the directory node of the leaf domain D in which the client resides. If the directory node does not store a location record for the entity, then the entity is currently not located in D. Consequently, the node forwards the request to its parent. Note that the parent node represents a larger domain than its child. If the parent also has no location record for E, the lookup request is forwarded to a next level higher, and so on.

![diagram](image_url)

**Figure 4-23.** Looking up a location in a hierarchically organized location service.

As soon as the request reaches a directory node M that stores a location record for entity E, we know that E is somewhere in the domain dom(M) represented by
node $M$. In Fig. 4-23, $M$ is shown to store a location record containing a pointer to one of its subdomains. The lookup request is then forwarded to the directory node of that subdomain, which in turn forwards it further down the tree, until the request finally reaches a leaf node. The location record stored in the leaf node will contain the address of $E$ in that leaf domain. This address can then be returned to the client that initially requested the lookup to take place.

An important observation with respect to hierarchical location services, is that the lookup operation exploits locality. In principle, the entity is searched in a gradually increasing ring centered around the requesting client. The search area is expanded each time the lookup request is forwarded to a next higher-level directory node. In the worst case, the search continues until the request reaches the root node. Because the root node has a location record for each entity, the request can then simply be forwarded along a downward path of pointers to one of the leaf nodes.

Update operations exploit locality in a similar fashion, as shown in Fig. 4-24. Consider an entity $E$ that has created a replica in leaf domain $D$ for which it needs to insert its address. The insertion is initiated at the leaf node $\text{dir}(D)$ of $D$, which immediately forwards the insert request to its parent. The parent will forward the insert request as well, until it reaches a directory node $M$ that already stores a location record for $E$.

Node $M$ will then store a pointer in the location record for $E$, referring to the child node from where the insert request was forwarded. At that point, the child node creates a location record for $E$, containing a pointer to the next lower-level node from where the request came. This process continues until we reach the leaf node from which the insert was initiated. The leaf node, finally, creates a record with the entity’s address in the associated leaf domain.

![Diagram](image)

Figure 4-24. (a) An insert request is forwarded to the first node that knows about entity $E$. (b) A chain of forwarding pointers to the leaf node is created.

SEC. 4.2 LOCATING MOBILE ENTITIES

Inserting an address as just described, leads to installing the chain of pointers in a top-down fashion starting at the lowest-level directory node that has a location record for entity $E$. An alternative is to create a location record before passing the insert request to the parent node. In other words, the chain of pointers is constructed from the bottom up. The advantage of the latter is that an address becomes available for lookups as soon as possible. Consequently, if a parent node is temporarily unreachable, the address can still be looked up within the domain represented by the current node.

A delete operation is analogous to an insert operation. When an address for entity $E$ in leaf domain $D$ needs to be removed, directory node $\text{dir}(D)$ is requested to remove that address from its location record for $E$. If that location record becomes empty, that is, it contains no other addresses for $E$ in $D$, the record can be removed. In that case, the parent node of $\text{dir}(D)$ wants to remove its pointer to $\text{dir}(D)$. If the location record for $E$ at the parent node now also becomes empty, that record should be removed as well and the next higher-level directory node should be informed. Again, this process continues until a pointer is removed from a location record that remains nonempty afterward or until the root is reached.

**Pointer Caches**

A hierarchical location service is aimed at supporting mobile entities, that is, entities of which the current location changes regularly. In traditional naming services, the mapping between a name and an address is assumed to be stable, at least for nodes in the global and administration layer. Consequently, storing lookup results from those nodes in local caches can be highly effective.

Caching addresses locally in the case of a location service will generally not be very effective. The next time the address of a mobile entity is looked up, it may very well have moved on to another location. Consequently, we are forced to follow the entire lookup operation as described above. This approach makes a hierarchical location service unavoidably more expensive than most naming services.

Caching is effective only if the cached data rarely change. Consider a mobile entity $E$ that moves regularly within a domain $D$. Moving within that domain means that $E$ will regularly change its current address. However, the path of pointers for entity $E$ from the root node to $\text{dir}(D)$ does not have to change. In other words, the location where information concerning the most recent whereabouts of $E$ is stored, remains the same, in this case the directory node $\text{dir}(D)$. Therefore, it is effective to cache a reference to the directory node.

In general, if $D$ is the smallest domain in which a mobile entity moves regularly, then it makes sense to start a lookup operation for the current location of $E$ at $\text{dir}(D)$, instead of any other node. This approach is essentially followed in the location service described in (Jain, 1996) and the Globe location service (van Steen et al., 1998b; Baggio et al., 2000), and is referred to as **pointer caching**.
reference to \( \text{dir}(E) \), can, in principle, be cached at every node along the path from the leaf node where the lookup was initiated as shown in Fig. 4.25.

![Figure 4.25](image)

**Figure 4.25.** Caching a reference to a directory node of the lowest-level domain in which an entity will reside most of the time.

Further improvements can be made by not letting \( \text{dir}(E) \) store a pointer to the subdomain where \( E \) currently resides, but instead letting it directly store the actual address of \( E \). Combined with pointer caching, a lookup operation can possibly be realized in only two steps. The first step requires inspecting the local pointer cache, leading directly to the appropriate directory node. The second step involves requesting that node to return the current address of \( E \).

Although the principle of pointer caching in hierarchical location services works, there are a number of open questions that need further attention. One question is how to find the best directory node to store the current address of a mobile entity. Imagine a user with a mobile computer regularly moving within, and between, two different cities, say San Francisco and Los Angeles.

When the user is San Francisco, it can be expected that he will change location regularly within that city. Consequently, it would make sense to store its current location in the directory node representing the San Francisco domain. A similar behavior pattern occurs when our busy beaver is in Los Angeles.

However, what also happens is that the user flies between San Francisco and Los Angeles all the time. Given that fact, it may be more effective to store its current location in a higher-level directory node such as the one for the state of California, irrespective of whether that location is in San Francisco or Los Angeles.

Another open question is when to invalidate a cache entry. Suppose, in our example, that the user gets so many requests from New York that he decides it makes sense to open an office in Manhattan and let one of his friends handle all incoming requests from the New York district. In terms of the location service, what happens is that a permanent address is available in the leaf node for the Manhattan domain. Any lookup request from New York should return that new address, and not follow a cached pointer to the California directory node, as shown in Fig. 4.26.

![Figure 4.26](image)

**Figure 4.26.** A cache entry that needs to be invalidated because it returns a nonlocal address, while such an address is available.

### Scalability Issues

One of the main problems with hierarchical location services is that the root node is required to store a location record for each entity and to process requests for each entity. Storage, itself, is not a major problem. Each location record can be relatively small, as it consists only of an identifier for an entity, with one or more pointers to lower-level directory nodes. If the size of each location record is approximately 1 KB, the required storage capacity for, say, a billion entities, is only one terabyte. That capacity can be provided by ten 100 GB disks.

The real problem is that without any special measures, the root may be required to handle so many lookup and update requests that it will become a bottleneck. The solution to this problem is to partition the root node and other high-level directory nodes into subnodes. Each subnode is responsible for handling the requests related to a specific subset of all the entities that are to be supported by the location service.

Unfortunately, simply partitioning the high-level nodes is not enough. To understand the scalability problem at hand, consider the partitioning of only the root node into, say, 100 subnodes. The question is where to physically place each subnode in the network that is covered by the location service.

One possibility for placing the subnodes is to follow a centralized approach by which the subnodes are kept close to each other, for example, in a cluster. Effectively, the root node is then implemented by means of a parallel computer, such as a COW or MPP (which we briefly discussed in Chap. 1). However, although the
processing capacity may now be enough, the network connections to and from the root node may not have enough capacity to handle all requests.

A much better alternative is therefore to spread the subnodes uniformly across the network. However, if not done properly, this approach may also introduce scalability problems. Consider again the mobile user moving primarily between San Francisco and Los Angeles. The root node has been partitioned into subnodes, and the question that needs to be addressed is which subnode should be made responsible for this user.

Figure 4.27. The scalability issues related to uniformly placing subnodes of a partitioned root node across the network covered by a location service.

Assume a subnode has been placed in Finland, and is chosen to always store a location record for this user. Ignoring pointer caches, this means that lookup requests from, for example, Brazil, will pass the root node in Finland before being forwarded to the California directory node. However, as shown in Fig. 4.27, it would have been more efficient if such a request had been passed through a subnode located in, for example, California. Deciding which subnodes should handle which entities in very large-scale location services is still an open question.

A possible solution is to take the location where an entity E is created into account. In particular, the subnode of the root that is closest to the location where E is created becomes responsible for handling root-level requests for E. This solution works for entities that tend to stay close to where they were created. However, if an entity moves to a faraway location, the problem remains. Details concerning this approach can be found in (Baltitjinit et al., 1999).

4.3 REMOVING UNREFERENCED ENTITIES

Naming and location services provide a global referencing service for entities. As long as an entity is referred to by such a service, it can be accessed and used. As soon as an entity can no longer be accessed, it should be removed.

In many systems, removing an entity is done explicitly. For example, if a process P knows it is the last process left to use a file, and that no other process will ever want to use that file in the future, P may just as well remove the file when it is finished. Unfortunately, managing the removal of entities in a distributed system is often difficult. In particular, it is often unknown whether a reference to an entity is stored somewhere in the system, with the intention of accessing the entity through that reference later on. If that is indeed the case, removing the entity will lead to an error when subsequent access is attempted.

On the other hand, it is also unacceptable to never remove an entity just because it is not known for certain whether a reference to that entity exists. If no reference exists, we have the situation that there is an entity which most likely consumes resources but which is never to be used again in the future. Clearly, such entities are garbage and should be removed.

To alleviate the problems related to removing unreferenced entities, distributed systems may offer facilities to automatically remove an entity when it is no longer needed. These facilities are also collectively known as distributed garbage collectors. In this section, we take a closer look at the relation between naming and referencing entities, and automatically collecting those entities that are no longer referenced.

4.3.1 The Problem of Unreferenced Objects

To explain how garbage collection works, we concentrate on garbage collecting distributed objects, in particular, remote objects. Recall that a remote object is implemented by having its entire state located at an object server, whereas clients have only a proxy. As we explained in Sec. 2.3, a reference to a remote object is generally implemented as what we can now refer to as a (proxy, skeleton) pair. The client-side proxy contains all the information to contact the object by means of its associated skeleton as implemented by the server. In our examples, the skeleton will take part in doing the administration necessary for garbage collection, together with the proxies. In other words, all that is needed to do garbage collection is hidden from the clients and the actual objects. Note that an object itself can hold a remote reference to another object, for example, by means of a local pointer to the proxy in that remote reference. Likewise, a remote reference can be passed to another process by copying its associated proxy to that other process.

In what follows, we assume that an object can be accessed only if there is a remote reference to it. An object for which no remote reference exists should be
removed from the system. On the other hand, having a remote reference to an object does not mean that the object will ever be accessed. For various reasons, it is possible that there are two objects, each storing a reference to the other, but are otherwise not referenced at all. This situation is easily generalized to a cycle of objects referring only to each other. Such objects should also be detected and removed.

In general, this model can be represented by a graph in which each node represents an object. An edge from node $M$ to node $N$ represents the fact that object $M$ contains a reference to object $N$. There is a distinguished subset of objects called the root set, which need not be referenced themselves. An object in the root set typically represents a systemwide service, a user, and so on.

Fig. 4-28 shows an example of such a reference graph. All the white nodes represent objects that are not directly or indirectly referenced by objects in the root set. Such objects should be removed.

Figure 4-28. An example of a graph representing objects containing references to each other.

In a single-processor system, detecting and removing unreferenced objects is relatively simple compared to the situation in a distributed system (for an overview of garbage collection in uniprocessor systems, see Wilson, 1994). Because the objects are distributed across multiple machines, distributed garbage collection requires network communication. As it turns out, this communication significantly determines the efficiency and scalability of solutions. In addition, communication as well as machines and processes are subject to failures, which makes problems even worse.

In this section, we consider a number of well-known solutions to distributed garbage collection. In most cases, these solutions only partly solve the problem. Our approach follows the one taken in (Plainsosse and Shapiro, 1995), which provides a further overview of distributed garbage collection. More information can also be found in (Abdullahi and Ringwood, 1998).

4.3.2 Reference Counting

A method that is popular in uniprocessor systems to check whether an object can be deleted is to simply count the references to that object. Each time a reference to an object is created, a reference counter for that object is incremented. Likewise, when a reference is removed, the reference counter is decremented. As soon as the counter reaches zero, the object can be removed.

Simple Reference Counting

Simple reference counting in distributed systems leads to a number of problems, which are partly caused by the fact that communication is not reliable. Without loss of generality, we can assume that an object stores its own reference counter in its associated skeleton as maintained by the object server that is responsible for the object. This situation is shown in Fig. 4-29.

When a process $P$ creates a reference to a remote object $O$, it installs a proxy $p$ for $O$ in its address space, as also shown in Fig. 4-29. To increment the reference counter, the proxy sends a message $m$ to the object's skeleton, and expects it to return an acknowledgement. However, if the acknowledgement is lost, the proxy will retransmit $m$. If no special measures are taken to detect duplicate messages, the skeleton may falsely increment its reference counter again. In practice, detecting duplicate messages is relatively easy.

Likewise, problems may also be caused when a remote reference is to be removed. In that case, a proxy will send a message to decrement the reference counter. If the acknowledgement is lost again, then a retransmission of that message may lead to another, incorrect decrement of the counter. Consequently, in distributed reference counting, it is essential to detect duplicate messages and to subsequently discard them as they come in.
Another problem that needs to be resolved occurs when copying a remote reference to another process. If process $P_1$ passes a reference to process $P_2$, the object, or more precisely, its skeleton, will be unaware that a new reference has been created. Consequently, if process $P_1$ decides to remove its own reference, the reference counter may drop to zero and $O$ may be deleted before $P_2$ ever contacts it. This problem is illustrated in Fig. 4-30(a).

![Figure 4-30](image)

Figure 4-30. (a) Copying a reference to another process and incrementing the counter too late. (b) A solution.

A solution is to let $P_1$ inform the object’s skeleton that it is going to pass a reference to process $P_2$. In addition, a process is never allowed to remove a reference before the skeleton has acknowledged that it knows about the existence of that reference. This solution is shown in Fig. 4-30(b). The acknowledgement sent by $O$ to $P_2$ confirming to $P_2$ that $O$ has registered the reference, will later permit $P_2$ to delete its reference to $O$. As long as $P_2$ is not sure that $O$ knows about its reference, $P_2$ is not allowed to request $O$ to decrement the reference counter.

Note that, in addition to reliable communication, passing a reference now requires three messages. Obviously, this can easily lead to performance problems in large-scale distributed systems.

**Advanced Reference Counting**

Simple distributed reference counting imposes a race condition between incrementing and decrementing the reference counter as just explained. Such race conditions can be avoided if only decrement operations can take place. This solution is adopted in **weighted reference counting**, in which each object has a fixed, total weight. When the object is created, the total weight is stored in its associated skeleton (which we refer to as $s$), along with a partial weight, which is initialized to the total weight, as shown in Fig. 4-31(a).

![Figure 4-31](image)

Figure 4-31. (a) The initial assignment of weights in weighted reference counting. (b) Weight assignment when creating a new reference. (c) Weight assignment when copying a reference.

When a new remote reference $(p,s)$ is created, half of the partial weight stored in the object’s skeleton is assigned to the new proxy $p$, as shown in Fig. 4-31(b). The remaining half is kept at skeleton $s$. When a remote reference is duplicated, for example, when passing it from process $P_1$ to $P_2$, half of the partial weight of the proxy in $P_1$ is assigned to the copied proxy for $P_2$, while the other half remains in the proxy at $P_1$, as shown in Fig. 4-31(c).

When a reference is destroyed, a decrement message is sent to the object’s skeleton, which subsequently subtracts the partial weight of the removed reference from the total weight. As soon as the total weight reaches zero, no more remote references exist, so that the object can be safely removed. Note that, also in this case, messages are assumed not to be lost nor delivered more than once.

The main problem with weighted reference counting is that only a limited number of references can be created. As soon as the partial weight of the object’s skeleton, as well as those of remote references, drops to zero, no more references can be created or duplicated. The solution to this problem is to make use of indirection. Assume process $P_1$ wants to pass a reference to $P_2$, but the partial weight of its own proxy has reached 1, as shown in Fig. 4-32. In that case, $P_1$ creates a skeleton $s'$ in its address space with an appropriate total weight, and a partial weight set equal to the total weight. This is completely analogous to the skeleton $s$ created in the address space where the object resides. A proxy is then sent to $P_2$, with half of the partial weight of skeleton $s'$ assigned to it. The other half of the weight is kept at $s'$ for distribution to other proxies.
to the skeleton containing the proxy’s generation number, say, \( k \), and the number of copies that have been made from \( p \), say \( n \). The skeleton adjusts \( G \) by first decrementing \( G[k] \) by one, indicating that a reference belonging to the \( k \)-th generation has been removed. Second, it increments \( G[k+1] \) by \( n \), to register that the removed reference had created \( n \) siblings, or, in other words, that it had been copied to \( n \) next-generation references. (Note that the skeleton may first need to create entry \( G[k+1] \) as generation \( k+1 \) was as yet unknown to it.) As soon as each entry \( G[j] \) is zero, it is known that the object is no longer referenced so that it can be removed.

Generation reference counting still requires reliable communication, but can handle the duplication of references without the need to contact the skeleton when making a copy.

### 4.3.3 Reference Listing

A different approach to managing references, is to let a skeleton keep track of the proxies that have a reference to it. In other words, instead of counting references, a skeleton maintains an explicit list of all proxies that point to it. Such a reference list has the following important property. Adding a proxy to a reference list has no effect when that proxy was already listed. Likewise, removing a proxy from a list in which it did not occur, also has no effect. Adding or removing proxies are thus idempotent operations.

Idempotent operations are characterized by the fact that they can be repeated without affecting the end result. In particular, when creating a new reference to an object, the creating process can repeatedly send a message to the object’s skeleton, requesting it to add its proxy to the reference list. It stops sending such a message as soon as delivery has been acknowledged. Similarly, removing a reference can be reported by (possibly repeatedly) sending a message asking the skeleton to remove the proxy from its list. Increment and decrement operations are clearly not idempotent.

Consequently, reference listing does not require communication to be reliable, nor is it necessary that duplicate messages can be detected and discarded. (However, it is necessary that the insertion or deletion of a reference is acknowledged.) This is an important advantage over reference counting.

Reference listing is used in Java RMI, based on a method described in (Birrell et al., 1993). In this method, when a process \( P \) creates a remote reference to an object, it sends its identification to the object’s skeleton, which adds \( P \) to the reference list. When an acknowledgement is returned, the process creates a proxy for the object in its own address space.

Passing a reference to another process, that is, sending a copy of a proxy, is handled similarly. Whenever process \( P_1 \) sends a copy of its proxy for object \( O \) to process \( P_2 \), \( P_2 \) first requests the object’s skeleton to add \( P_2 \) to its reference list. When that has been done, process \( P_2 \) installs the proxy in its address space.
Problems may occur when process $P_1$ removes its own proxy before $P_2$ has requested to be inserted in the object’s reference list. In that case, if the remove request that $P_1$ sends to the object’s skeleton is handled before the insert request from $P_2$, the reference list may become empty so that the skeleton falsely concludes it can let the object be removed. This race condition is completely analogous to the one with reference counting, as shown in Fig. 4-30(a), and can be solved in a similar way.

Another important advantage of reference listing, is that it is easier to keep the reference list consistent in the face of process failures. The object’s skeleton regularly checks whether each listed process is still up and running by sending it a ping message, asking it whether it is still alive and holding a reference to the object. The process is expected to promptly respond to this message. If no response is received, possibly even after several attempts have been made, the skeleton removes the process from its list.

The main drawback of reference listing, is that it may scale badly if the skeleton needs to keep track of many references. One solution for keeping the list down, is to let the skeleton promise it will register a reference for only a limited time. If a proxy has not renewed its registration at the skeleton before that time expires, the reference is simply dropped from the list. This approach is also referred to as handing out a lease. We return to leases in Chap. 6.

### 4.3.4 Identifying Unreachable Entities

As was also shown in Fig. 4-28, the collection of entities in a distributed system may consist of entities that store references to each other, but none of these entities can be reached from an entry in the root set, and as such, they should also be removed. Unfortunately, the garbage collection techniques described above fail to locate these entities.

What is needed is a method by which all entities in a distributed system are traced. In general, this is done by checking which entities can be reached from the root set and subsequently removing all others. Such methods are generally called tracing-based garbage collection. In contrast to distributed referencing discussed so far, tracing-based garbage collection has inherent scalability problems, as it needs to trace all entities in a distributed system.

### Naive Tracing in Distributed Systems

To understand distributed tracing-based garbage collection, it is helpful to consider how tracing in uniprocessor system works. The most simple approach to uniprocessor tracing, is followed by mark-and-sweep collectors. Such collectors distinguish two phases.

During the mark phase, entities are traced by following chains of references originating from entities in the root set. Each entity that can be reached in this way is marked, for example by recording the entry in a separate table. The sweep phase consists of exhaustively examining memory to locate entities that have not been marked. Such entities are considered garbage that is to be removed.

Another way to look at mark-and-sweep collectors is to apply a three-color marking to entities. Initially, each entity that needs to be inspected is colored white. By the end of the mark phase, all entities that have been colored black are reachable from root set, while those that are unreachable are still white. The color gray is used to keep track of the progress that is being made in the mark phase. An entity is marked gray when it is found to be reachable but the references stored by that entity need yet to be inspected. When all its references have been marked gray, the entity is colored black.

A distributed version of mark-and-sweep was implemented in the Emerald system, described in (Jul et al., 1998). In Emerald, a local garbage collector is started at each process, with all the collectors running in parallel. Collectors color proxies, skeletons, and the actual objects. Initially, all of them are marked white. When an object residing in the address space of process $P$ is reachable from a root that is also in $P$, the object is marked gray. When marking an object gray, all proxies contained in that object are marked gray as well. Marking a proxy gray means that the local garbage collector records that the referenced remote object still needs to be checked by its associated local garbage collector.

When a proxy is marked gray, a message is sent to the proxy’s associated skeleton to mark itself gray as well. The object associated with a skeleton is marked gray as soon as the skeleton becomes gray. By recursion, this means that all proxies in that object are marked gray as well. At that point, the skeleton and its associated object is marked black, and a message is sent back to all its associated proxies. Note that although in this approach, a skeleton knows which proxies are connected to it, this does not imply that a proxy is considered reachable from that skeleton. Logically, a (proxy, skeleton) pair is a strict unidirectional reference from the proxy to the skeleton.

When a proxy receives a message that its associated skeleton is now black, the proxy is marked black as well. In other words, the local garbage collector now knows that the remote object referenced by means of the proxy has been recorded as being reachable.

When all local collectors have finished their mark phase, they can each separately collect all white objects as garbage. A mark phase ends when all objects, skeletons, and proxies have been marked either white or black. Removing a white object also means removing its associated skeleton as well as all proxies contained in that object.

The main drawback of the mark-and-sweep algorithm is that it requires the reachability graph to remain the same during both phases. In other words, the execution of the program for which the process was originally created needs to be temporarily stopped and execution is switched to collecting garbage. In distributed mark-and-sweep, this means that all processes in the system first need to be
synchronized, then each of them switches to collecting garbage, after which they can all continue with their original work.

This scenario, also called "stop-the-world" synchronization, is often not acceptable for distributed garbage collectors. Improvements can be made with incremental garbage collectors, which allow program execution to be interleaved with garbage collection. Unfortunately, such collectors do not scale well in distributed systems. Because they run concurrently with programs that modify the reachability graph, objects are often necessarily marked gray, leading to the propagation of gray marks to remote processes. The result is high message traffic, possibly degrading overall system performance.

**Tracing in Groups**

To account for the inherent scalability problems of many tracing-based garbage collectors, Lang et al. devised a method by which the processes (which contain the objects) in a large distributed system are hierarchically organized into groups (Lang et al., 1992). Garbage collection takes place within groups through a combination of mark-and-sweep and reference counting. Let us concentrate on the basic algorithm for collecting garbage in a group of processes.

A group is simply a collection of processes. The only reason why groups are used is for scalability. The basic idea is first to collect all garbage within a group, including any cycles of references that lie entirely inside a group. A next step is to consider a larger group that encompasses a number of subgroups but which have each just been cleaned up by the garbage collector.

To accommodate tracing in groups, it is assumed that remote references are again implemented as (proxy, skeleton) pairs. For each object, there is only one skeleton in the address space where the object resides, but multiple proxies for that object can communicate with that skeleton. The skeleton maintains a reference counter as described in Sec. 4.3.2, which counts the number of associated proxies. A process can have at most one proxy for each distributed object.

Once a group of processes has been formed, the basic algorithm to collect garbage within a group consists of the following five steps, which are discussed in detail below:

1. Initial marking, in which only skeletons are marked.
2. Intraprocess propagation of marks from skeletons to proxies.
3. Interprocess propagation of marks from proxies to skeletons.
4. Stabilization by repetition of the previous two steps.
5. Garbage reclamation.

Before we start explaining each of these steps in some detail, it is important to understand what marking an entity stands for. The algorithm essentially deals with marking only proxies and skeletons. It is important to note that neither a proxy or skeleton can ever belong to the root set.

A skeleton can be marked either soft or hard whereas a proxy can be marked none, soft, or hard. When a skeleton is marked hard, this means that it is either reachable from a proxy in a process outside the group, or reachable from a root object inside the group, that is, an object contained in the root set of a process belonging to the group. A skeleton that is marked soft, is considered to be reachable only from proxies inside the group. The marking of a skeleton is allowed to change only from soft to hard.

A proxy that is marked hard, is reachable from an object in the root set. When marked soft, the proxy is reachable from a skeleton that has been marked soft as well. Such proxies potentially lie on a cycle that is not reachable from an object in the root set. A proxy that is marked none is neither reachable from a skeleton, nor an object in the root set. Only proxies that have been marked none can be changed to hard. As will be seen, once a proxy has been marked soft, it stays that way.

The first step consists of marking only the skeletons. A skeleton is marked either soft or hard, depending on whether it can be reached from a proxy outside the group. This reachability can be easily checked by inspecting the skeleton's reference counter. The value of this counter indicates how many proxies in other processes refer to it. Some of these processes are inside the group, while others are outside the group. If there is a proxy in a process outside the group, the skeleton should be marked hard. By simply counting how many proxies associated with the skeleton lie inside the group and subtracting that number from the reference counter, it can be decided if there are also associated proxies outside the group. This leads to the following algorithm:

1. For each proxy inside the group, decrement the reference counter of the associated skeleton, only if that skeleton is also inside the group.
2. A skeleton inside the group whose reference counter has now dropped to zero is marked soft. Otherwise, it is reachable from a proxy outside the group, and is marked hard.

This first step is illustrated in Fig. 4-34(a), in which all but one skeleton have been marked soft. The only skeleton that is marked hard, is seen to be reachable from a proxy outside the group.

The second step consists of letting each process run its own local garbage collector. How that collector works is independent of the global garbage collection. The only requirement is that a local garbage collector propagates marks from skeletons to proxies within the process it is running. More specifically, suppose that within a single process a proxy is reachable from a skeleton. (Note that the proxy and the skeleton belong to different objects.) The result of local propagation of marks is that the proxy will be marked at least as hard as the skeleton. Moreover, if a proxy is reachable from an object in the root set, it will be marked hard.
Local propagation within process $P$ can be done as follows. Initially, all proxies in $P$ are marked none. The local collector starts tracing from the set consisting of the skeletons that have previously been marked hard, as well as from the objects in the root set. Hard marks are propagated to all objects (that is, local objects and proxies) that are reachable from this set. A second trace is done from skeletons that had been marked soft. If a proxy is now reached that is marked soft, its mark is changed to hard. If the proxy was marked hard, it remains marked as such. Consequently, after local propagation, each proxy in a process will be marked either none, soft, or hard. Fig. 4-34(b) shows the marking after locally propagating the marks from Fig. 4-34(a).

The third step consists of propagating marks from proxies to their associated skeletons. In other words, marks are propagated between different processes. In particular, if a proxy has been marked hard, a message should be sent to its associated skeleton to mark it hard as well, if it had not already been marked as such. A message is sent only if the skeleton lies inside the group. Soft marks do not have to be propagated: the initial marking phase already established that each skeleton inside the group is marked either soft or hard.

The fourth step results from the global propagation of hard marks in the previous step. A skeleton in, say, process $P$ may now have had its mark changed from soft to hard. This change comes from the fact that the skeleton turned out to be reachable from an object contained in the root set of a remote process. Consequently, this hard mark first needs to be locally propagated to proxies in $P$, and subsequently, globally propagated to neighboring processes. In other words, steps 2 and 3 are to be repeated as long as marks can be either locally or globally propagated. As soon as stabilization has been reached, that is, no more changes with respect to marking happen to processes inside the group, the algorithm proceeds with the next step. In our example, the effect of repeating step 2 and 3 leads to the final marking as shown in Fig. 4-34(c).

The fifth and last step consists of removing unreachable objects, including unreachable proxies, as well as those proxies and skeletons that have been marked soft. It is important to note that the latter are not reachable from outside the group, nor are they reachable from objects in a root set. In other words, the soft-marked proxies and skeletons refer only to each other, and should thus be removed.

Garbage reclamation can actually be done as a side effect of local propagation. Instead of explicitly removing entities in the last step, a skeleton marked soft is changed to refer to nil. Consequently, it can be reclaimed later when the local garbage collection is run again. In addition, if the object associated with that skeleton now becomes unreachable, it will be reclaimed as well. If the proxies locally referred to by that object are also no longer reachable, they will be marked none and remain marked that way. It is therefore safe to let the local garbage collector reclaim a none-marked proxy, after sending a decrement message to the proxy's associated skeleton in one of remote processes.
By hierarchically organizing processes into groups, a more scalable solution to distributed garbage collection can be achieved. The basic idea is to let low-level groups collect garbage, and leave the analysis of intergroup references to the next higher-level group. By letting lower-level groups reduce the number of objects that need to be traced, a higher-level group essentially operates on a similar number of objects as each of its subgroups, but which are spread across a larger network. We omit the details, which can be found in (Lang et al., 1992).

4.4 SUMMARY

Names are used to refer to entities. Essentially, there are three types of names. An address is the name of an access point associated with an entity, also simply called the address of an entity. An identifier is another type of name. It has three properties: each entity is referred to by exactly one identifier, an identifier refers to only one entity, and is never assigned to another entity. Finally, human-friendly names are targets to be used by humans and as such are represented as character strings.

Names are organized in a name space. A name space can be represented by a naming graph in which a node represents a named entity and the label on an edge represents the name under which that entity is known. A node having multiple outgoing edges represents a collection of entities and is also known as a context node or directory. Large-scale naming graphs are often organized as rooted acyclic directed graphs.

Naming graphs are convenient to organize human-friendly names in a structured way. An entity can be referred to by a path name. Name resolution is the process of traversing the naming graph by looking up the components of a path name, one at a time. A large-scale naming graph is implemented by distributing its nodes across multiple name servers. When resolving a name path by traversing the naming graph, name resolution continues at a next name server as soon as a node is reached implemented by that server.

Naming systems for human-friendly names are not suited for highly mobile entities. Locating mobile entities can be done more efficiently using location-independent identifiers. There are basically four approaches to locating a mobile entity. The first approach is to use broadcasting or multicasting. The identifier of the entity is broadcast to every process in the distributed system. The process offering an access point for the entity responds by providing an address for that access point. Obviously, this approach has limited scalability.

The second approach is to use forwarding pointers. Each time an entity moves to a next location, it leaves behind a pointer telling where it will be next. Locating the entity requires traversing the path of forwarding pointers. To avoid large chains of pointers, it is important to reduce chains after a while.

The third approach is to allocate a home to an entity. Each time an entity moves to another location, it informs its home where it is. Locating an entity proceeds by first asking its home for the current location.

The fourth approach is to build a hierarchical search tree. The network is divided into nonoverlapping domains. Domains can be grouped into higher-level (nonoverlapping) domains, and so on. There is a single top-level domain that covers the entire network. Each domain at every level has an associated directory node. If an entity is located in a domain D, the directory node of the next higher-level domain will have a pointer to D. A lowest-level directory node stores the address of the entity. The top-level directory node knows about all entities.

Entities that can no longer be located should be removed. An important use of names in distributed systems is to organize references to entities in such a way that unreferenced entities are automatically removed. This garbage collection requires the support of reference counting or tracing.

With reference counting, an entity simply counts the number of outstanding references to it. When the counter drops to zero, the entity can be removed. Instead of counting references, it is also possible to maintain a list of processes referring to an entity. Reference listing is more robust than reference counting, but has scalability problems.

With tracing-based methods, all entities that are directly or indirectly referenced from a given set of root entities, are marked as reachable. Entities that are unreachable are to be removed. Distributed tracing is difficult as it requires that all entities in a system are to be inspected. Solutions vary, but are generally based on traditional garbage collectors used in uniprocessor systems.

PROBLEMS

1. Give an example of where an address of an entity E needs to be further resolved into another address to actually access E.

2. Would you consider a URL such as http://www.acme.org/index.html to be location independent? What about http://www.acme.nl/index.html?

3. Give some examples of true identifiers.

4. How is a mounting point looked up in most UNIX systems?

5. Jade is a distributed file system that uses per-user name spaces (Rao and Peterson, 1993). In other words, each user has his own, private name space. Can names from such name spaces be used to share resources between two different users?

6. Consider DNS. To refer to a node N in a subdomain implemented as a different zone than the current domain, a name server for that zone needs to be specified. Is it always necessary to include a resource record for that server’s address, or is it sometimes sufficient to provide only its domain name?
7. Is an identifier allowed to contain information on the entity it refers to?
8. Outline an efficient implementation of globally unique identifiers.
9. Give an example of how the closure mechanism for a URL could work.
10. Explain the difference between a hard link and a soft link in UNIX systems.
11. High-level name servers in DNS, that is, name servers implementing nodes in the DNS name space that are close to the root, generally do not support recursive name resolution. Can we expect much performance improvement if they did?
12. Explain how DNS can be used to implement a home-based approach to locating mobile hosts.
13. A special form of locating an entity is called anycasting, by which a service is identified by means of an IP address (see, for example, Partridge et al., 1993). Sending a request to anycast address returns a response from a server implementing the service identified by that anycast address. Outline the implementation of an anycast service based on the hierarchical location service described in Sec. 4.2.4.
14. Considering that a two-tiered home-based approach is a specialization of a hierarchical location service, where is the root?
15. Suppose that it is known that a specific mobile entity will almost never move outside domain D, and if it does, it can be expected to return soon. How can this information be used to speed up the lookup operation in a hierarchical location service?
16. In a hierarchical location service with a depth of k, how many location records need to be updated at most when a mobile entity changes its location?
17. Consider an entity moving from location A to B, while passing several intermediate locations where it will reside for only a relatively short time. When arriving at B, it settles down for a while. Changing an address in a hierarchical location service may still take a relatively long time to complete, and should therefore be avoided when visiting an intermediate location. How can the entity be located at an intermediate location?
18. When passing a remote reference from process P1 to P2 in distributed reference counting, would it help to let P1 increment the counter, instead of P2?
19. Make clear that weighted reference counting is more efficient than simple reference counting. Assume communication is reliable.
20. Is it possible in generation reference counting that an object is collected as garbage while there are still references, but which belong to a generation the object does not know of?
21. Is it possible in generation reference counting that an entry G[i] becomes less than 0?
22. In reference listing, if no response is received after sending a ping message to process P, the process is removed from the object’s reference list. Is it always correct to remove the process?
23. Describe a very simple way to decide that the stabilization step in the tracing-based garbage collector of Lang et al. has been reached.

5

SYNCHRONIZATION

In the previous chapters, we have looked at processes and communication between processes. While communication is important, it is not the entire story. Closely related is how processes cooperate and synchronize with one another. Cooperation is partly supported by means of naming, which allows processes to at least share resources, or entities in general.

In this chapter, we mainly concentrate on how processes can synchronize. For example, it is important that multiple processes do not simultaneously access a shared resource, such as a printer, but instead cooperate in granting each other temporary exclusive access. Another example is that multiple processes may sometimes need to agree on the ordering of events, such as whether message m1 from process P was sent before or after message m2 from process Q.

As it turns out, synchronization in distributed systems is often much more difficult compared to synchronization in uniprocessor or multiprocessor systems. The problems and solutions that are discussed in this chapter are, by their nature, rather general, and occur in many different situations in distributed systems.

We start with a discussion of the issue of synchronization based on actual time, followed by synchronization in which only relative ordering matters rather than ordering in absolute time. We also discuss the notion of a distributed global state, and how, by synchronizing processes, such a state can be recorded.

In many cases, it is important that a group of processes can appoint one process as a coordinator, which can be done by means of election algorithms. We discuss various election algorithms in a separate section.