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 server. 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 server.

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:<label-1, label-2, ..., 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

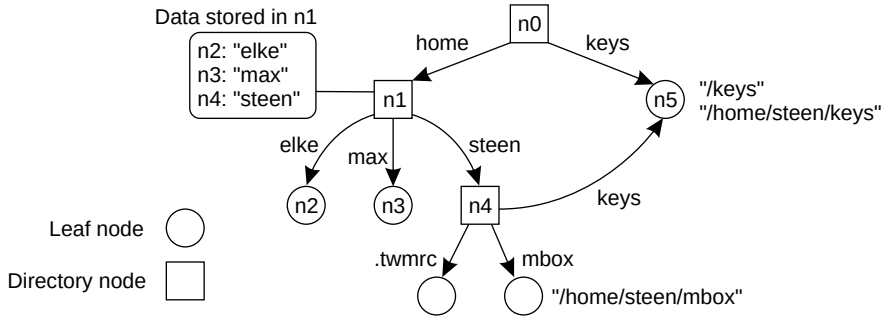


Figure 4-1. A general naming graph with a single root node.

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 name 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:<home, steen, mbox>* it is common practice to use its string representation */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 */home/steen/keys* as well as */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.

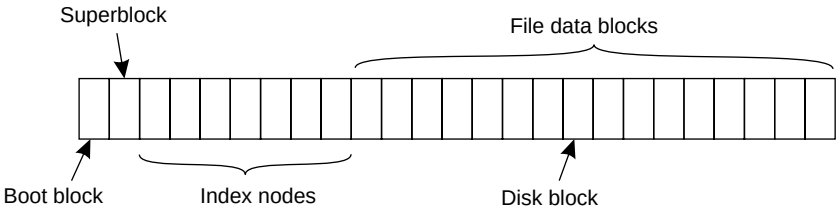


Figure 4-2. The general organization of the UNIX file system implementation on a logical disk of contiguous disk blocks.

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 *N:<label-1, label-2, ..., label-n>*. Resolution of this name starts at node *N* of the naming graph, where the name *label-1* is looked up in the directory table, and which returns the identifier of the node to which *label-1* refers. Resolution then continues at the identified node by looking up the name *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 *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 *n5* 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 *n5*.

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 *n5*.

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 **mount 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

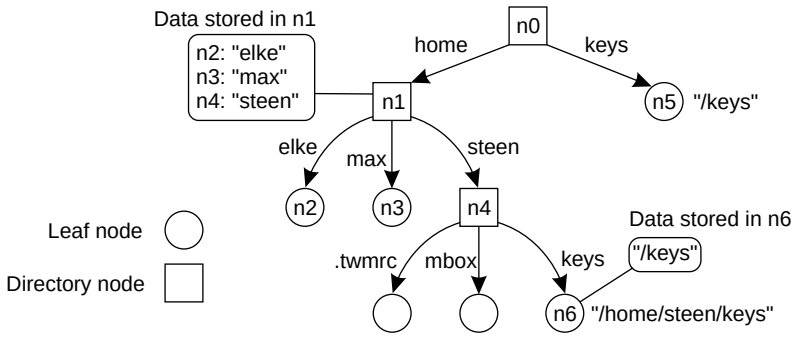


Figure 4-3. The concept of a symbolic link explained in a naming graph.

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.

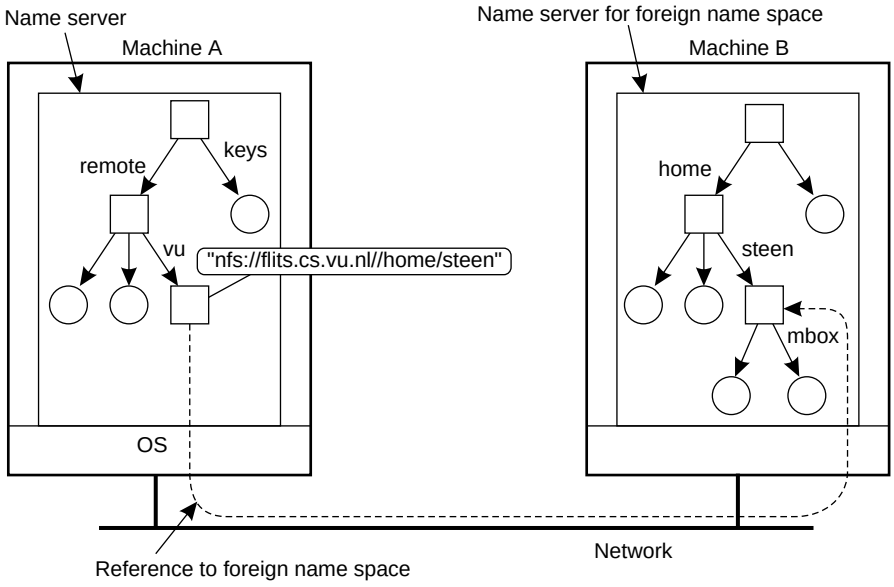


Figure 4-4. Mounting remote name spaces through a specific access protocol.

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 */vu/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.

Merging two name spaces *NS1* and *NS2* but without having to change existing names, now proceeds as follows. When adding a new root node, that node stores a table mapping the identifier of the root node of *NS1* to the name under which that root is known in the new name space, as shown in Fig. 4-5. The same is done for the root node of *NS2*. By always starting name resolution in the root of the new name space, a name such as *n0:/home/steen* is first transformed to the name */vu/home/steen*, by looking up the node identifier *n0* in the table of the root node.

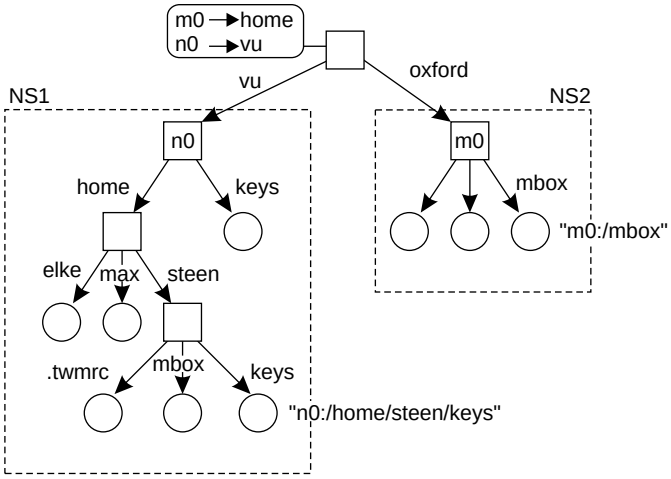


Figure 4-5. Organization of the DEC Global Name Service.

A potential problem with GNS is that the root node of the merged name space is required to maintain a mapping of identifiers of old root nodes to their new respective names. If thousands of name spaces are merged, this approach will eventually lead to performance problems.

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 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.

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 administrational 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

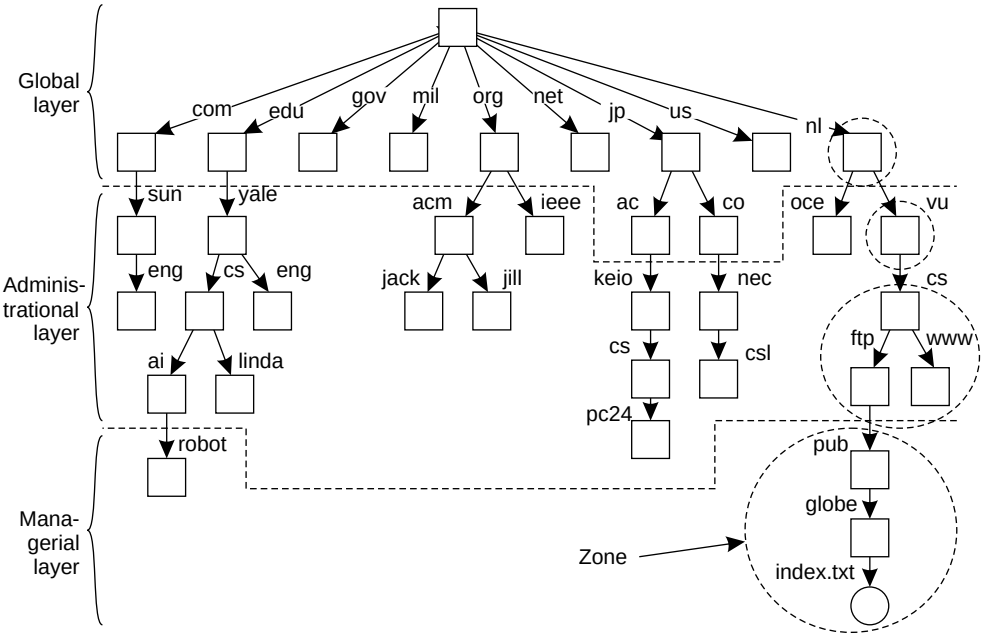


Figure 4-6. An example partitioning of the DNS name space, including Internet-accessible files, into three layers.

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 administrational 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 administrational 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.

A comparison between name servers at different layers is shown in Fig. 4-7.

Item	Global	Administrational	Managerial
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side caching applied?	Yes	Yes	Sometimes

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

In distributed systems, name servers in the global and administrational 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>*.)

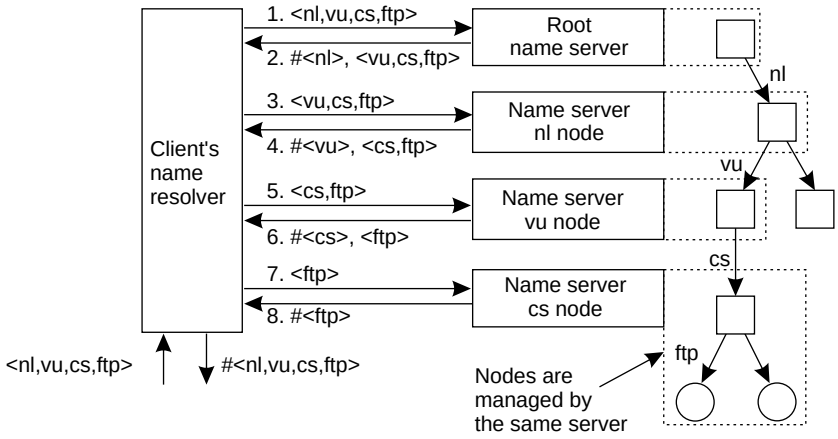


Figure 4-8. The principle of iterative name resolution.

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 resolution, 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.

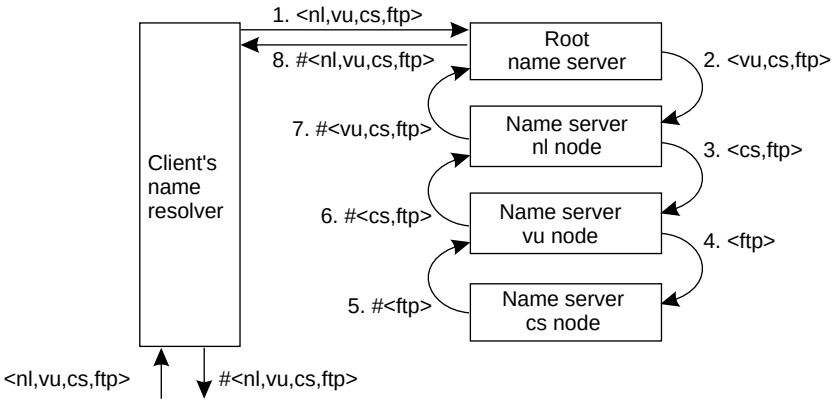


Figure 4-9. The principle of recursive name resolution.

The main drawback of recursive name resolution is that it puts a higher performance 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 cooperation 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 resolution.

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 administrational 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 administrational 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.

Server for node	Should resolve	Looks up	Passes to child	Receives and caches	Returns to requester
cs	<ftp>	#<ftp>	—	—	#<ftp>
vu	<cs,ftp>	#<cs>	<ftp>	#<ftp>	#<cs> #<cs, ftp>
nl	<vu,cs,ftp>	#<vu>	<cs,ftp>	#<cs> #<cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>
root	<nl,vu,cs,ftp>	#<nl>	<vu,cs,ftp>	#<vu> #<vu,cs> #<vu,cs,ftp>	#<nl> #<nl,vu> #<nl,vu,cs> #<nl,vu,cs,ftp>

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 *R*1 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 *R*2. Finally, communication is needed between the *vu* server and the name server in the Computer Science Department, shown as *R*3. 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.

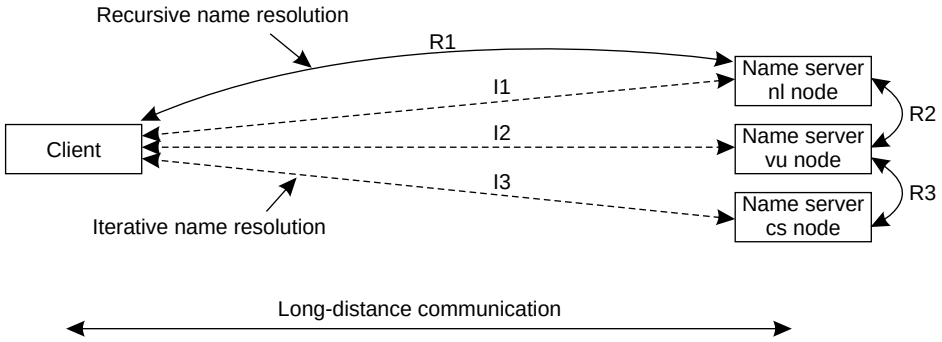


Figure 4-11. The comparison between recursive and iterative name resolution with respect to communication costs.

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.

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

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 *zephyr.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.tcp.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 administrational 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.

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. The way to refer to nodes in a subdomain

Name	Record type	Record value
cs.vu.nl	SOA	star (1999121502,7200,3600,2419200,86400)
cs.vu.nl	NS	star.cs.vu.nl
cs.vu.nl	NS	top.cs.vu.nl
cs.vu.nl	NS	solo.cs.vu.nl
cs.vu.nl	TXT	"Vrije Universiteit - Math. & Comp. Sc."
cs.vu.nl	MX	1 zephyr.cs.vu.nl
cs.vu.nl	MX	2 tornado.cs.vu.nl
cs.vu.nl	MX	3 star.cs.vu.nl
star.cs.vu.nl	HINFO	Sun Unix
star.cs.vu.nl	MX	1 star.cs.vu.nl
star.cs.vu.nl	MX	10 zephyr.cs.vu.nl
star.cs.vu.nl	A	130.37.24.6
star.cs.vu.nl	A	192.31.231.42
zephyr.cs.vu.nl	HINFO	Sun Unix
zephyr.cs.vu.nl	MX	1 zephyr.cs.vu.nl
zephyr.cs.vu.nl	MX	2 tornado.cs.vu.nl
zephyr.cs.vu.nl	A	192.31.231.66
www.cs.vu.nl	CNAME	soling.cs.vu.nl
ftp.cs.vu.nl	CNAME	soling.cs.vu.nl
soling.cs.vu.nl	HINFO	Sun Unix
soling.cs.vu.nl	MX	1 soling.cs.vu.nl
soling.cs.vu.nl	MX	10 zephyr.cs.vu.nl
soling.cs.vu.nl	A	130.37.24.11
laser.cs.vu.nl	HINFO	PC MS-DOS
laser.cs.vu.nl	A	130.37.30.32
vucs-das.cs.vu.nl	PTR	0.26.37.130.in-addr.arpa
vucs-das.cs.vu.nl	A	130.37.26.0

Figure 4-13. An excerpt from the DNS database for the zone *cs.vu.nl*.

Name	Record type	Record value
cs.vu.nl	NS	solo.cs.vu.nl
solo.cs.vu.nl	A	130.37.24.1

Figure 4-14. Part of the description for the *vu.nl* domain which contains the *cs.vu.nl* domain.

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 *OrganizationUnit* 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.

Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	Vrije Universiteit
OrganizationalUnit	OU	Math. & Comp. Sc.
CommonName	CN	Main server
Mail_Servers	—	130.37.24.6, 192.31.231.42, 192.31.231.66
FTP_Server	—	130.37.24.11
WWW_Server	—	130.37.24.11

Figure 4-15. A simple example of a X.500 directory entry using X.500 naming conventions.

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

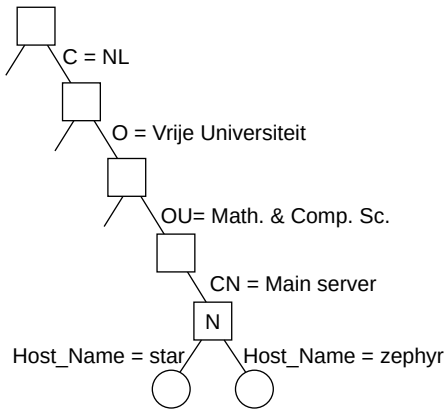
/C=NL/O=Vrije Universiteit/OU=Math. & Comp. Sc.

analogous to the DNS name *nl.vu.cs*.

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).

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



(a)

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Math. & Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Math. & Comp. Sc.
CommonName	Main server
Host_Name	zephyr
Host_Address	192.31.231.66

(b)

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

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.

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 **DUA**s. 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 DIB. 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 according to the official rules is not trivial. To accommodate X.500 directory services 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

many distributed systems, including, for example, Windows 2000, which we discuss in Chap. 9. Practical information on LDAP can be found in (Johner et al., 1998).