13

NAME SERVICES

This chapter introduces the name service as a distinct service that is used by client processes to obtain attributes such as the addresses of resources or objects when given their names. The entities named can be of many types, and they may be managed by different services. For example, name services are often used to hold the addresses and other details of users, computers, network domains, services and remote objects. As well as name services, we describe directory services, which look up services when given some of their attributes.

Basic design issues for name services, such as the structure and management of the space of names recognized by the service and the operations that the name service supports, are outlined and illustrated in the context of the Internet Domain Name System(DNS).

We also examine how name services are implemented, covering such aspects as navigation through a collection of name servers when resolving a name, caching naming data and replicating naming data to increase performance and availability.

Two further case studies are included: the Global Name Service (GNS), and the X.500 Directory Service, including LDAP.

13.1 Introduction

In a distributed system, names are used to refer to a wide variety of resources such as computers, services, remote objects and files, as well as to users. Naming is an issue that is easily overlooked but is nonetheless fundamental in distributed system design. Names facilitate communication and resource sharing. A name is needed to request a computer system to act upon a specific resource chosen out of many; for example, a name in the form of a URL is needed to access a specific web page. Processes cannot share particular resources managed by a computer system unless they can name them consistently. Users cannot communicate with one another via a distributed system unless they can name one another, for example, with email addresses.

Names are not the only useful means of identification: descriptive attributes are another. Sometimes clients do not know the name of the particular entity that they seek, but they do have some information that describes it. Or they may require a service and know some of its characteristics but not what entity implements it.

This chapter introduces name services, which provide clients with data about named objects in distributed systems, and the related concept of directory services, which provide data about objects that satisfy a given description. We describe approaches to be taken in the design and implementation of these services, using the Domain Name Service (DNS), the Global Name Service (GNS) and X500 as case studies. We begin by examining the fundamental concepts of names and attributes.

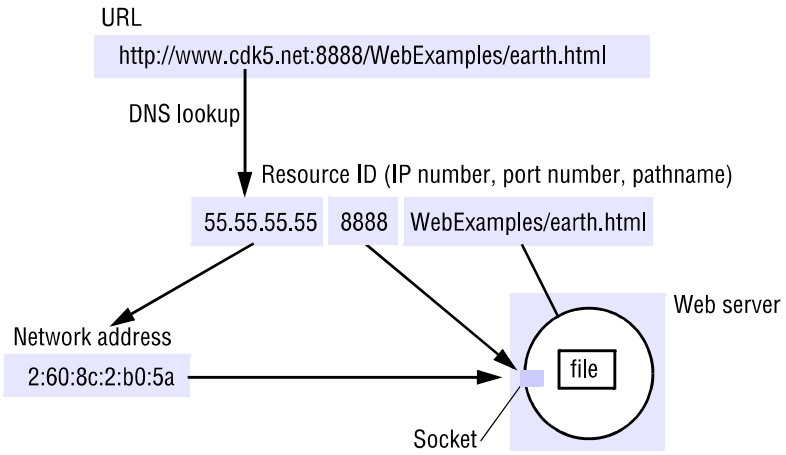
13.1.1 Names, addresses and other attributes

Any process that requires access to a specific resource must possess a name or an identifier for it. Examples of human-readable names are file names such as */etc/passwd*, URLs such as *http://www.cdk5.net/* and Internet domain names such as *www.cdk5.net.* The term *identifier* is sometimes used to refer to names that are interpreted only by programs. Remote object references and NFS file handles are examples of identifiers. Identifiers are chosen for the efficiency with which they can be looked up and stored by software.

Needham [1993] makes the distinction between a *pure* name and other names. Pure names are simply uninterpreted bit patterns. Non-pure names contain information about the object that they name; in particular, they may contain information about the location of the object. Pure names always have to be looked up before they can be of any use. At the other extreme from a pure name is an object's *address*: a value that identifies the location of the object rather than the object itself. Addresses are efficient for accessing objects, but objects can sometimes be relocated, so addresses are inadequate as a means of identification. For example, users' email addresses usually have to change when they move between organizations or Internet service providers; they are not in themselves guaranteed to refer to a specific individual over time.

We say that a name is *resolved* when it is translated into data about the named resource or object, often in order to invoke an action upon it. The association between a name and an object is called a *binding*. In general, names are bound to *attributes* of the named objects, rather than the implementation of the objects themselves. An attribute is

Figure 13.1 Composed naming domains used to access a resource from a URL



the value of a property associated with an object. A key attribute of an entity that is usually relevant in a distributed system is its address. For example:

- The DNS maps domain names to the attributes of a host computer: its IP address, the type of entry (for example, a reference to a mail server or another host) and, for example, the length of time the host's entry will remain valid.
- The X500 directory service can be used to map a person's name onto attributes including their email address and telephone number.
- The CORBA Naming Service and Trading Service were presented in Chapter 8. The Naming Service maps the name of a remote object onto its remote object reference, whereas the Trading Service maps the name of a remote object onto its remote object reference, together with an arbitrary number of attributes describing the object in terms understandable by human users.

Note that an 'address' may be considered just another name that must be looked up, or it may contain such a name. An IP address must be looked up to obtain a network address such as an Ethernet address. Similarly, web browsers and email clients make use of the DNS to interpret the domain names in URLs and email addresses. Figure 13.1 shows the domain name portion of a URL resolved first via the DNS into an IP address and then, at the final hop of Internet routing, via ARP to an Ethernet address for the web server. The last part of the URL is resolved by the file system on the web server to locate the relevant file.

Names and services • Many of the names used in a distributed system are specific to some particular service. For example, users of the social networking web site *twitter.com*, have names such as *@magmapoetry* that no other service resolves. Also, a client may use a service-specific name when requesting a service to perform an operation upon a named object or resource that it manages. For example, a file name is given to the file service when requesting that the file be deleted, and a process identifier is presented to the process management service when requesting that it be sent a signal.

These names are used only in the context of the service that manages the objects named, except when clients communicate about shared objects.

Names are also sometimes needed to refer to entities in a distributed system that are beyond the scope of any single service. The major examples of these entities are users (with proper names and email addresses), computers (with *hostnames* such as *www.cdk5.net*) and services themselves (such as *file service* or *printer service*). In object-based middleware, names refer to remote objects that provide services or applications. Note that many of these names must be readable by and meaningful to humans, since users and system administrators need to refer to the major components and configuration of distributed systems, programmers need to refer to services in programs, and users need to communicate with each other via the distributed system and discuss what services are available in different parts of it. Given the connectivity provided by the Internet, these naming requirements are potentially world-wide in scope.

Uniform Resource Identifiers • *Uniform Resource Identifiers* (URIs) [Berners-Lee *et al.* 2005] came about from the need to identify resources on the Web, and other Internet resources such as electronic mailboxes. An important goal was to identify resources in a coherent way, so that they could all be processed by common software such as browsers. URIs are 'uniform' in that their syntax incorporates that of indefinitely many individual types of resource identifiers (that is, URI *schemes*), and there are procedures for managing the global namespace of schemes. The advantage of uniformity is that it eases the process of introducing new types of identifier, as well as using existing types of identifier in new contexts, without disrupting existing usage.

For example, if someone was to invent a new type of 'widget' URI, then URIs beginning *widget*: would have to obey the global URI syntax, as well as any local rules defined for the widget identifier scheme. These URIs would identify *widget* resources in a well-defined way. But even existing software that did not access *widget* resources could still process *widget* URIs – for example, by managing directories containing them. Turning to an example of incorporating existing identifiers, that has been done for telephone numbers by prefixing them with the scheme name *tel* and standardizing their representation, as in *tel:+1-816-555-1212*. These *tel* URIs are intended for uses such as web links that cause telephone calls to be made when invoked.

Uniform Resource Locators: Some URIs contain information that can be used to locate and access a resource; others are pure resource names. The familiar term *Uniform Resource Locator* (URL) is often used for URIs that provide location information and specify the method for accessing the resource, including the 'http' URLs introduced in Section 1.6. For example, *http://www.cdk5.net/* identifies a web page at the given path ('/') on the host *www.cdk5.net*, and specifies that the HTTP protocol be used to access it. Another example is a 'mailto' URL, such as *mailto:fred@flintstone.org*, which identifies the mailbox at the given address.

URLs are efficient identifiers for accessing resources. But they suffer from the disadvantage that if a resource is deleted or if it moves, say from one web site to another, there may be dangling links to the resource containing the old URL. If a user clicks on a dangling link to a web resource, then the web server will either respond that the resource is not found or – worse, perhaps – supply a different resource that now occupies the same location.

Uniform Resource Names: *Uniform Resource Names* (URNs) are URIs that are used as pure resource names rather than locators. For example, the URI:

mid:0E4FC272-5C02-11D9-B115-000A95B55BC8@hpl.hp.com

is a URN that identifies the email message containing it in its 'Message-Id' field. The URI distinguishes that message from any other email message. But it does not provide the message's address in any store, so a lookup operation is needed to find it.

A special subtree of URIs beginning with *urn:* has been reserved for URNs – although, as the *mid:* example shows, not all URNs are *urn*: URIs. The latter *urn*-prefixed URIs are all of the form *urn:nameSpace:nameSpace-specificName*. For example, *urn:ISBN:0-201-62433-8* identifies books that bear the name 0-201-62433-8 in the standard ISBN naming scheme. For another example, the (invented) name *urn:doi:10.555/music-pop-1234* refers to the publication called *music-pop-1234* in the naming scheme of the publisher known as *10.555* in the Digital Object Identifier (DOI) scheme [www.doi.org].

There are *resolution services* (name services, in the terminology of this chapter) such as the Handle System [www.handle.net] for resolving URNs such as DOIs to resource attributes, but none is in widespread use. Indeed, there continues to be debate in the Web and Internet research communities about the extent to which a separate category of URNs is needed. One school of thought is that 'cool URLs do not change' – in other words, that everyone should assign URLs to resources with guarantees about their continuity of reference. Against that point of view is the observation that not everyone is in a position to make such guarantees, which require the wherewithal to maintain control of a domain name and administer resources carefully.

13.2 Name services and the Domain Name System

A *name service* stores information about a collection of textual names, in the form of bindings between the names and the attributes of the entities they denote, such as users, computers, services and objects. The collection is often subdivided into one or more naming *contexts*: individual subsets of the bindings that are managed as a unit. The major operation that a name service supports is to resolve a name – that is, to look up attributes from a given name. We describe the implementation of name resolution in Section 13.2.2. Operations are also required for creating new bindings, deleting bindings and listing bound names, and adding and deleting contexts.

Name management is separated from other services largely because of the openness of distributed systems, which brings the following motivations:

Unification: It is often convenient for resources managed by different services to use the same naming scheme. URIs are a good example of this.

Integration: It is not always possible to predict the scope of sharing in a distributed system. It may become necessary to share and therefore name resources that were created in different administrative domains. Without a common name service, the administrative domains may use entirely different naming conventions.

General name service requirements • Name services were originally quite simple, since they were designed only to meet the need to bind names to addresses in a single management domain, corresponding to a single LAN or WAN. The interconnection of networks and the increased scale of distributed systems have produced a much larger name-mapping problem.

Grapevine [Birrell *et al.* 1982] was one of the earliest extensible, multi-domain name services. It was designed to be scalable in the number of names and the load of requests that it could handle.

The Global Name Service, developed at the Digital Equipment Corporation Systems Research Center [Lampson 1986], is a descendant of Grapevine with ambitious goals, including:

To handle an essentially arbitrary number of names and to serve an arbitrary number of administrative organizations: For example, the system should be capable of handling the names of all the documents in the world.

A long lifetime: Many changes will occur in the organization of the set of names and in the components that implement the service during its lifetime.

High availability: Most other systems depend upon the name service; they can't work when it is broken.

Fault isolation: Local failures should not cause the entire service to fail.

Tolerance of mistrust: A large open system cannot have any component that is trusted by all of the clients in the system.

Two examples of name services that have concentrated on the goal of scalability to large numbers of objects such as documents are the Globe name service [van Steen *et al.* 1998] and the Handle System [www.handle.net]. Far more familiar is the Internet Domain Name System (DNS), introduced in Chapter 3, which names computers (and other entities) across the Internet.

In this section, we discuss the main design issues for name services, giving examples from the DNS. We follow this with a more detailed case study of the DNS.

13.2.1 Name spaces

A *name space* is the collection of all valid names recognized by a particular service. The service will attempt to look up a valid name, even though that name may prove not to correspond to any object – i.e., to be *unbound*. Name spaces require a syntactic definition to separate valid names from invalid names. For example, '...' is not acceptable as the DNS name of a computer, whereas *www.cdk99.net* is valid (even though it is unbound).

Names may have an internal structure that represents their position in a hierarchic name space such as pathnames in a file system, or in an organizational hierarchy such as Internet domain names; or they may be chosen from a flat set of numeric or symbolic identifiers. One important advantage of a hierarchy is that it makes large name spaces more manageable. Each part of a hierarchic name is resolved relative to a separate context of relatively small size, and the same name may be used with different meanings in different contexts, to suit different situations of use. In the case of file systems, each

directory represents a context. Thus */etc/passwd* is a hierarchic name with two components. The first, 'etc', is resolved relative to the context '/', or root, and the second part, 'passwd', is relative to the context '/*etc*'. The name */oldetc/passwd* can have a different meaning because its second component is resolved in a different context. Similarly, the same name */etc/passwd* may resolve to different files in the contexts of two different computers.

Hierarchic name spaces are potentially infinite, so they enable a system to grow indefinitely. By contrast, flat name spaces are usually finite; their size is determined by fixing a maximum permissible length for names. Another potential advantage of a hierarchic name space is that different contexts can be managed by different people or organizations.

The structure of 'http' URLs was introduced in Chapter 1. The URL name space also includes *relative names* such as *../images/figure1.jpg*. When a browser or other web client encounters such a relative name, it uses the resource in which the relative name is embedded to determine the server host name and the directory to which this pathname refers.

DNS names are strings called *domain names*. Some examples are *www.cdk5.net* (a computer), *net*, *com* and *ac.uk* (the latter three are domains).

The DNS name space has a hierarchic structure: a domain name consists of one or more strings called *name components* or *labels*, separated by the delimiter '.'. There is no delimiter at the beginning or end of a domain name, although the root of the DNS name space is sometimes referred to as '.' for administrative purposes. The name components are non-null printable strings that do not contain '.'. In general, a *prefix* of a name is an initial section of the name that contains only zero or more entire components. For example, in DNS *www* and *www.cdk5* are both prefixes of *www.cdk5.net*. DNS names are not case-sensitive, so *www.cdk5.net* and *WWW.CDK5.NET* have the same meaning.

DNS servers do not recognize relative names: all names are referred to the global root. However, in practical implementations, client software keeps a list of domain names that are appended automatically to any single-component name before resolution. For example, the name *www* presented in the domain *cdk5.net* probably refers to *www.cdk5.net*; client software will append the default domain *cdk5.net* and attempt to resolve this name. If this fails, then further default domain names may be appended; finally, the (absolute) name *www* will be presented to the root for resolution (an operation that will of course fail in this case). Names with more than one component, however, are normally presented intact to the DNS, as absolute names.

Aliases • An *alias* is a name defined to denote the same information as another name, similar to a symbolic link between file path names. Aliases allow more convenient names to be substituted for relatively complicated ones, and allow alternative names to be used by different people for the same entity. An example is the common use of URL shorteners, often used in Twitter posts and other situations where space is at a premium. For example, using web redirection, *http://bit.ly/ctqjvH* refers to *http://cdk5.net/additional/rmi/programCode/ShapeListClient.java*. As another example, the DNS allows aliases in which one domain name is defined to stand for another. Aliases are often used to specify the names of machines that run a web server or an FTP server. For example, the name *www.cdk5.net* is an alias for *cdk5.net*. This has

the advantage that clients can use either name for the web server, and if the web server is moved to another computer, only the entry for *cdk5.net* needs to be updated in the DNS database.

Naming domains • A *naming domain* is a name space for which there exists a single overall administrative authority responsible for assigning names within it. This authority is in overall control of which names may be bound within the domain, but it is free to delegate this task.

Domains in DNS are collections of domain names; syntactically, a domain's name is the common suffix of the domain names within it, but otherwise it cannot be distinguished from, for example, a computer name. For example, *net* is a domain that contains *cdk5.net*. Note that the term 'domain name' is potentially confusing, since only some domain names identify domains (others identify computers).

The administration of domains may be devolved to subdomains. The domain *dcs.qmul.ac.uk* – the Department of Computer Science at Queen Mary, University of London in the UK – can contain any name the department wishes. But the domain name *dcs.qmul.ac.uk* itself had to be agreed with the college authorities, who manage the domain *qmul.ac.uk*. Similarly, *qmul.ac.uk* had to be agreed with the registered authority for *ac.uk*, and so on.

Responsibility for a naming domain normally goes hand in hand with responsibility for managing and keeping up-to-date the corresponding part of the database stored in an authoritative name server and used by the name service. Naming data belonging to different naming domains are in general stored by distinct name servers managed by the corresponding authorities.

Combining and customizing name spaces • The DNS provides a global and homogeneous name space in which a given name refers to the same entity, no matter which process on which computer looks up the name. By contrast, some name services allow distinct name spaces – sometimes heterogeneous name spaces – to be embedded into them; and some name services allow the name space to be customized to suit the needs of individual groups, users or even processes.

Merging: The practice of mounting file systems in UNIX and NFS (see Section 12.3) provides an example in which a part of one name space is conveniently embedded in another. But consider how to merge the *entire* UNIX file systems of two (or more) computers called *red* and *blue*. Each computer has its own root, with overlapping file names. For example, */etc/passwd* refers to one file on *red* and a different file on *blue*. The obvious way to merge the file systems is to replace each computer's root with a 'super root' and mount each computer's file system in this super root, say as */red* and */blue*. Users and programs can then refer to */red/etc/passwd* and */blue/etc/passwd*. But the new naming convention by itself would cause programs on the two computers that still use the old name */etc/passwd* to malfunction. A solution is to leave the old root contents on each computer and embed the mounted file systems */red* and */blue* of both computers (assuming that this does not produce name clashes with the old root contents).

The moral is that we can always merge name spaces by creating a higher-level root context, but this may raise a problem of backward-compatibility. Fixing the compatibility problem, in turn, leaves us with hybrid name spaces and the inconvenience of having to translate old names between the users of the two computers.

Heterogeneity: The Distributed Computing Environment (DCE) name space [OSF 1997] allows heterogeneous name spaces to be embedded within it. DCE names may contain *junctions*, which are similar to mount points in NFS and UNIX (see Section 12.3), except that they allow heterogeneous name spaces to be mounted. For example, consider the full DCE name */.../dcs.qmul.ac.uk/principals/Jean.Dollimore*. The first part of this name, */.../dcs.qmul.ac.uk,* denotes a context called a *cell*. The next component is a junction. For example, the junction *principals* is a context containing security principals in which the final component, *Jean.Dollimore*, may be looked up, and in which these principal names have their own syntax. Similarly, in */.../dcs.qmul.ac.uk/files/pub/reports/TR2000-99*, the junction *files* is a context corresponding to a file system directory, in which the final component *pub/reports/TR2000-99* is looked up and in which the file name space has a distinct syntax. The two junctions *principals* and *files* are the roots of heterogeneous name spaces, implemented by heterogeneous name services.

Customization: We saw in the example of embedding NFS-mounted file systems above that sometimes users prefer to construct their name spaces independently rather than sharing a single name space. File system mounting enables users to import files that are stored on servers and shared, while the other names continue to refer to local, unshared files and can be administered autonomously. But the same files accessed from two different computers may be mounted at different points and thus have different names. Not sharing the entire name space means users must translate names between computers.

The Spring naming service [Radia *et al.* 1993] provides the ability to construct name spaces dynamically and to share individual naming contexts selectively. Even two different processes on the same computer can have different naming contexts. Spring naming contexts are first-class objects that can be shared around a distributed system. For example, suppose a user on computer *red* wishes to run a program on *blue* that issues file pathnames such as */etc/passwd*, but these names are to resolve to the files on *red*'s file system, not *blue*'s. This can be achieved in Spring by passing a reference to *red*'s local naming context to *blue* and using it as the program's naming context. Plan 9 [Pike *et al.* 1993] also allows processes to have their own file system name space. A novel feature of Plan 9 (which can also be implemented in Spring) is that physical directories can be ordered and merged into a single logical directory. The effect is that a name looked up in the single logical directory is looked up in the succession of physical directories until there is a match, when the attributes are returned. This eliminates the need to supply lists of paths when looking for program or library files.

13.2.2 Name resolution

For the common case of hierarchic name spaces, name resolution is an iterative or recursive process whereby a name is repeatedly presented to naming contexts in order to look up the attributes to which it refers. A naming context either maps a given name onto a set of primitive attributes (such as those of a user) directly, or maps it onto a further naming context and a derived name to be presented to that context. To resolve a name, it is first presented to some initial naming context; resolution iterates as long as further contexts and derived names are output. We illustrated this at the start of Section

13.2.1 with the example of */etc/passwd*, in which 'etc' is presented to the context '/', and then 'passwd' is presented to the context '*/etc*'.

Another example of the iterative nature of resolution is the use of aliases. For example, whenever a DNS server is asked to resolve an alias such as *www.dcs.qmul.ac.uk*, the server first resolves the alias to another domain name (in this case *traffic.dcs.qmul.ac.uk*), which must be further resolved to produce an IP address.

In general, the use of aliases makes it possible for cycles to be present in the name space, in which case resolution may never terminate. Two possible solutions are, to abandon a resolution process if it passes a threshold number of resolutions, or to leave administrators to veto any aliases that would introduce cycles.

Name servers and navigation • Any name service, such as DNS, that stores a very large database and is used by a large population will not store all of its naming information on a single server computer. Such a server would be a bottleneck and a critical point of failure. Any heavily used name services should use replication to achieve high availability. We shall see that DNS specifies that each subset of its database is replicated in at least two failure-independent servers.

We mentioned above that the data belonging to a naming domain is usually stored by a local name server managed by the authority responsible for that domain. Although, in some cases, a name server may store data for more than one domain, it is generally true to say that data is partitioned into servers according to its domain. We shall see that in DNS, most of the entries are for local computers. But there are also name servers for the higher domains, such as *yahoo.com* and *ac.uk*, and for the root.

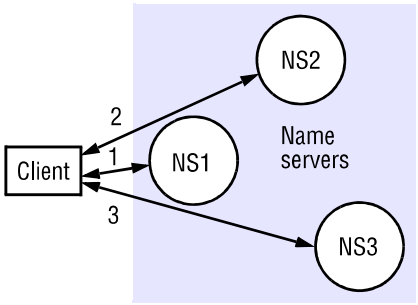
The partitioning of data implies that the local name server cannot answer all enquiries without the help of other name servers. For example, a name server in the *dcs.qmul.ac.uk* domain would not be able to supply the IP address of a computer in the domain *cs.purdue.edu* unless it was cached – certainly not the first time it is asked.

The process of locating naming data from more than one name server in order to resolve a name is called *navigation*. The client name resolution software carries out navigation on behalf of the client. It communicates with name servers as necessary to resolve a name. It may be provided as library code and linked into clients, as for example in the BIND implementation for DNS (see Section 13.2.3) or in Grapevine [Birrell *et al.* 1982]. The alternative, used with X500, is to provide name resolution in a separate process that is shared by all of the client processes on that computer.

One navigation model that DNS supports is known as *iterative navigation* (see Figure 13.2). To resolve a name, a client presents the name to the local name server, which attempts to resolve it. If the local name server has the name, it returns the result immediately. If it does not, it will suggest another server that will be able to help. Resolution proceeds at the new server, with further navigation as necessary until the name is located or is discovered to be unbound.

As DNS is designed to hold entries for millions of domains and is accessed by vast numbers of clients, it would not be feasible to have all queries starting at a root server, even if it were replicated heavily. The DNS database is partitioned between servers in such a way as to allow many queries to be satisfied locally and others to be satisfied without needing to resolve each part of the name separately. The scheme for resolving names in DNS is described in more detail in Section 13.2.3.

Figure 13.2 Iterative navigation



A client iteratively contacts name servers NS1–NS3 in order to resolve a name

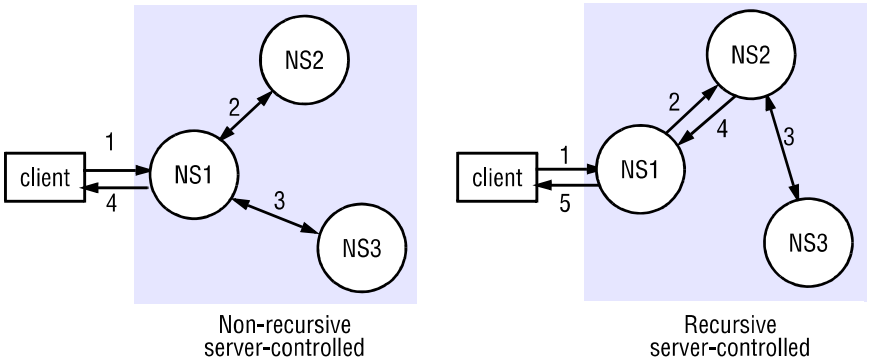
NFS also employs iterative navigation in the resolution of a file name, on a component-by-component basis (see Chapter 12). This is because the file service may encounter a symbolic link when resolving a name. A symbolic link must be interpreted in the client's file system name space because it may point to a file in a directory stored at another server. The client computer must determine which server this is, because only the client knows its mount points.

In *multicast navigation*, a client multicasts the name to be resolved and the required object type to the group of name servers. Only the server that holds the named attributes responds to the request. Unfortunately, however, if the name proves to be unbound, the request is greeted with silence. Cheriton and Mann [1989] describe a multicast-based navigation scheme in which a separate server is included in the group to respond when the required name is unbound.

Another alternative to the iterative navigation model is one in which a name server coordinates the resolution of the name and passes the result back to the user agent. Ma [1992] distinguishes *non-recursive* and *recursive server-controlled navigation* (Figure 13.3). Under non-recursive server-controlled navigation, any name server may be chosen by the client. This server communicates by multicast or iteratively with its peers in the style described above, as though it were a client. Under recursive server-controlled navigation, the client once more contacts a single server. If this server does not store the name, the server contacts a peer storing a (larger) prefix of the name, which in turn attempts to resolve it. This procedure continues recursively until the name is resolved.

If a name service spans distinct administrative domains, then clients executing in one administrative domain may be prohibited from accessing name servers belonging to another such domain. Moreover, even name servers may be prohibited from discovering the disposition of naming data across name servers in another administrative domain. Then, both client-controlled and non-recursive server-controlled navigation are inappropriate, and recursive server-controlled navigation must be used. Authorized name servers request name service data from designated name servers managed by different administrations, which return the attributes without revealing where the different parts of the naming database are stored.

Figure 13.3 Non-recursive and recursive server-controlled navigation



A name server NS1 communicates with other name servers on behalf of a client

Caching • In DNS and other name services, client name resolution software and servers maintain a cache of the results of previous name resolutions. When a client requests a name lookup, the name resolution software consults its cache. If it holds a recent result from a previous lookup for the name, it returns it to the client; otherwise, it sets about finding it from a server. That server, in turn, may return data cached from other servers.

Caching is key to a name service's performance and assists in maintaining the availability of both the name service and other services in spite of name server crashes. Its role in enhancing response times by saving communication with name servers is clear. Caching can be used to eliminate high-level name servers – the root server, in particular – from the navigation path, allowing resolution to proceed despite some server failures.

Caching by client name resolvers is widely applied in name services and is particularly successful because naming data are changed relatively rarely. For example, information such as computer or service addresses is liable to remain unchanged for months or years. However, the possibility exists of a name service returning out-of-date attributes – for example, an out-of-date address – during resolution.

13.2.3 The Domain Name System

The Domain Name System is a name service design whose main naming database is used across the Internet. It was devised principally by Mockapetris and specified in RFC 1034 [Mockapetris 1987] and RFC 1035. DNS replaced the original Internet naming scheme, in which all host names and addresses were held in a single central master file and downloaded by FTP to all computers that required them [Harrenstien *et al.* 1985]. This original scheme was soon seen to suffer from three major shortcomings:

- It did not scale to large numbers of computers.
- Local organizations wished to administer their own naming systems.
- A general name service was needed – not one that serves only for looking up computer addresses.

The objects named by the DNS are primarily computers – for which mainly IP addresses are stored as attributes – and what we have referred to in this chapter as naming domains are called simply *domains* in the DNS. In principle, however, any type of object can be named, and its architecture gives scope for a variety of implementations. Organizations and departments within them can manage their own naming data. Millions of names are bound by the Internet DNS, and lookups are made against it from around the world. Any name can be resolved by any client. This is achieved by hierarchical partitioning of the name database, by replication of the naming data, and by caching.

Domain names • The DNS is designed for use in multiple implementations, each of which may have its own name space. In practice, however, only one is in widespread use, and that is the one used for naming across the Internet. The Internet DNS name space is partitioned both organizationally and according to geography. The names are written with the highest-level domain on the right. The original top-level organizational domains (also called *generic domains*) in use across the Internet were:

com	–	Commercial organizations
edu	–	Universities and other educational institutions
gov	–	US governmental agencies
mil	–	US military organizations
net	–	Major network support centres
org	–	Organizations not mentioned above
int	–	International organizations

New top-level domains such as *biz* and *mobi* have been added since the early 2000s. A full list of current generic domain names is available from the Internet Assigned Numbers Authority [www.iana.org I].

In addition, every country has its own domains:

us	–	United States
uk	–	United Kingdom
fr	–	France
...	–	...

Countries, particularly those other than the US often use their own subdomains to distinguish their organizations. The UK, for example, has domains *co.uk* and *ac.uk*, which correspond to *com* and *edu* respectively (*ac* stands for 'academic community').

Note that, despite its geographic-sounding *uk* suffix, a domain such as *doit.co.uk* could have data referring to computers in the Spanish office of Doit Ltd., a notional British company. In other words, even geographic-sounding domain names are conventional and are completely independent of their physical locations.

DNS queries • The Internet DNS is primarily used for simple host name resolution and for looking up electronic mail hosts, as follows:

Host name resolution: In general, applications use the DNS to resolve host names into IP addresses. For example, when a web browser is given a URL containing the

domain name *www.dcs.qmul.ac.uk*, it makes a DNS enquiry and obtains the corresponding IP address. As was pointed out in Chapter 4, browsers then use HTTP to communicate with the web server at the given IP address, using a reserved port number if none is specified in the URL. FTP and SMTP services work in a similar way; for example, an FTP program may be given the domain name *ftp.dcs.qmul.ac.uk* and can make a DNS enquiry to get its IP address and then use TCP to communicate with it at the reserved port number.

Mail host location: Electronic mail software uses the DNS to resolve domain names into the IP addresses of mail hosts – i.e., computers that will accept mail for those domains. For example, when the address *tom@dcs.rnx.ac.uk* is to be resolved, the DNS is queried with the address *dcs.rnx.ac.uk* and the type designation 'mail'. It returns a list of domain names of hosts that can accept mail for *dcs.rnx.ac.uk*, if such exist (and, optionally, the corresponding IP addresses). The DNS may return more than one domain name so that the mail software can try alternatives if the main mail host is unreachable for some reason. The DNS returns an integer preference value for each mail host, indicating the order in which the mail hosts should be tried.

Some other types of query that are implemented in some installations but are less frequently used than those just given are:

Reverse resolution: Some software requires a domain name to be returned given an IP address. This is just the reverse of the normal host name query, but the name server receiving the query replies only if the IP address is in its own domain.

Host information: The DNS can store the machine architecture type and operating system with the domain names of hosts. It has been suggested that this option should not be used in public, because it provides useful information for those attempting to gain unauthorized access to computers.

In principle, the DNS can be used to store arbitrary attributes. A query is specified by a domain name, class and type. For domain names in the Internet, the class is IN. The type of query specifies whether an IP address, a mail host, a name server or some other type of information is required. A special domain, *in-addr.arpa*, exists to hold IP addresses for reverse lookups. The class attribute is used to distinguish, for example, the Internet naming database from other (experimental) DNS naming databases. A set of types is defined for a given database; those for the Internet database are given in Figure 13.5.

DNS name servers • The problems of scale are treated by a combination of partitioning the naming database and replicating and caching parts of it close to the points of need. The DNS database is distributed across a logical network of servers. Each server holds part of the naming database – primarily data for the local domain. Queries concerning computers in the local domain are satisfied by servers within that domain. However, each server records the domain names and addresses of other name servers, so that queries pertaining to objects outside the domain can be satisfied.

The DNS naming data are divided into *zones*. A zone contains the following data:

- Attribute data for names in a domain, less any subdomains administered by lower-level authorities. For example, a zone could contain data for Queen Mary, University of London – *qmul.ac.uk* – less the data held by departments (for example the Department of Computer Science – *dcs.qmul.ac.uk*).

- The names and addresses of at least two name servers that provide *authoritative* data for the zone. These are versions of zone data that can be relied upon as being reasonably up-to-date.
- The names of name servers that hold authoritative data for delegated subdomains; and 'glue' data giving the IP addresses of these servers.
- Zone-management parameters, such as those governing the caching and replication of zone data.

A server may hold authoritative data for zero or more zones. So that naming data are available even when a single server fails, the DNS architecture specifies that each zone must be replicated authoritatively in at least two servers.

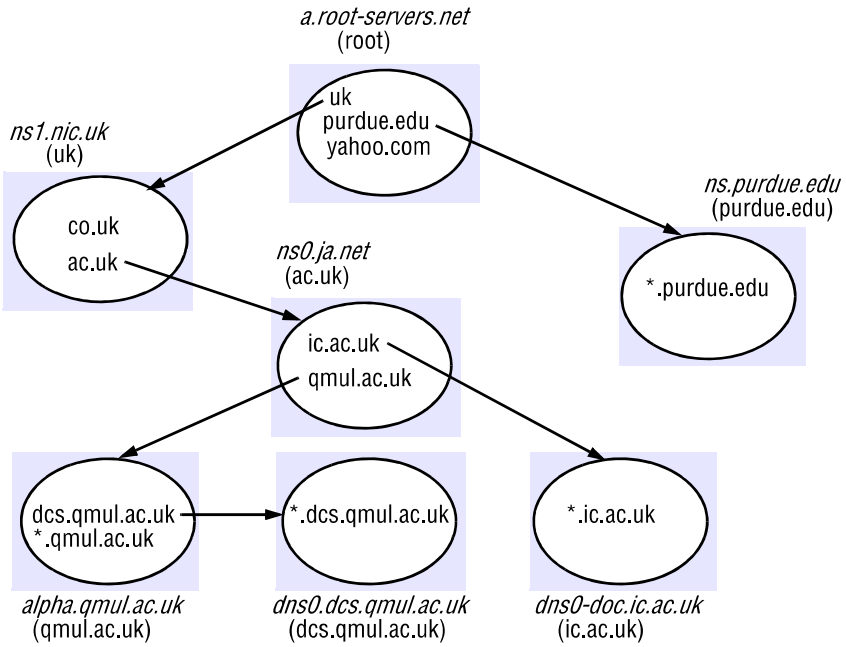
System administrators enter the data for a zone into a master file, which is the source of authoritative data for the zone. There are two types of server that are considered to provide authoritative data. A *primary* or *master server* reads zone data directly from a local master file. *Secondary servers* download zone data from a primary server. They communicate periodically with the primary server to check whether their stored version matches that held by the primary server. If a secondary's copy is out of date, the primary sends it the latest version. The frequency of the secondary's check is set by administrators as a zone parameter, and its value is typically once or twice a day.

Any server is free to cache data from other servers to avoid having to contact them when name resolution requires the same data again; it does this on the proviso that clients are told that such data is non-authoritative as supplied. Each entry in a zone has a time-to-live value. When a non-authoritative server caches data from an authoritative server, it notes the time to live. It will only provide its cached data to clients for up to this time; when queried after the time period has expired, it recontacts the authoritative server to check its data. This is a useful feature that minimizes the amount of network traffic while retaining flexibility for system administrators. When attributes are expected to change rarely, they can be given a correspondingly large time to live. If an administrator knows that attributes are likely to change soon, they can reduce the time to live accordingly.

Figure 13.4 shows the arrangement of some of the DNS database as it stood in the year 2001. This example is equally valid today even if some of the data has altered as systems have been reconfigured over time. Note that, in practice, root servers such as *a.root-servers.net* hold entries for several levels of domain, as well as entries for first-level domain names. This is to reduce the number of navigation steps required to resolve domain names. Root name servers hold authoritative entries for the name servers for the top-level domains. They are also authoritative name servers for the generic top-level domains, such as *com* and *edu*. However, the root name servers are not name servers for the country domains. For example, the *uk* domain has a collection of name servers, one of which is called *ns1.nic.net*. These name servers know the name servers for the second-level domains in the United Kingdom such as *ac.uk* and *co.uk*. The name servers for the domain *ac.uk* know the name servers for all of the university domains in the country, such as *qmul.ac.uk* or *ic.ac.uk*. In some cases, a university domain delegates some of its responsibilities to a subdomain, such as *dcs.qmul.ac.uk*.

The root domain information is replicated by a primary server to a collection of secondary servers, as described above. In spite of this, root servers serve thousands or

Figure 13.4 DNS name servers



Name server names are in italics, and the corresponding domains are in parentheses. Arrows denote name server entries

more queries per second. All DNS servers store the addresses of one or more root name servers, which do not change very often. They also usually store the address of an authoritative server for the parent domain. A query involving a three-component domain name such as *www.berkeley.edu* can be satisfied using at worst two navigation steps: one to a root server that stores an appropriate name server entry, and a second to the server whose name is returned.

Referring to Figure 13.4, the domain name *jeans-pc.dcs.qmul.ac.uk* can be looked up from within *dcs.qmul.ac.uk* using the local server *dns0.dcs.qmul.ac.uk*. This server does not store an entry for the web server *www.ic.ac.uk*, but it does keep a cached entry for *ic.ac.uk* (which it obtained from the authorized server *ns0.ja.net*). The server *dns0-doc.ic.ac.uk* can be contacted to resolve the full name.

Navigation and query processing • A DNS client is called a *resolver*. It is normally implemented as library software. It accepts queries, formats them into messages in the form expected under the DNS protocol and communicates with one or more name servers in order to satisfy the queries. A simple request-reply protocol is used, typically using UDP packets on the Internet (DNS servers use a well-known port number). The resolver times out and resends its query if necessary. The resolver can be configured to contact a list of initial name servers in order of preference in case one or more are unavailable.

Figure 13.5 DNS resource records

Record type	*Meaning*	*Main contents*
A	A computer address (IPv4)	IPv4 number
AAAA	A computer address (IPv6)	IPv6 number
NS	An authoritative name server	Domain name for server
CNAME	The canonical name for an alias	Domain name for alias
SOA	Marks the start of data for a zone	Parameters governing the zone
PTR	Domain name pointer (reverse lookups)	Domain name
HINFO	Host information	Machine architecture and operating system
MX	Mail exchange	List of <*preference*, *host*> pairs
TXT	Text string	Arbitrary text

The DNS architecture allows for recursive navigation as well as iterative navigation. The resolver specifies which type of navigation is required when contacting a name server. However, name servers are not bound to implement recursive navigation. As was pointed out above, recursive navigation may tie up server threads, meaning that other requests might be delayed.

In order to save on network communication, the DNS protocol allows for multiple queries to be packed into the same request message and for name servers correspondingly to send multiple replies in their response messages.

Resource records • Zone data are stored by name servers in files in one of several fixed types of resource record. For the Internet database, these include the types given in Figure 13.5. Each record refers to a domain name, which is not shown. The entries in the table refer to items already mentioned, except that *AAAA* records store IPv6 addresses whereas *A* records store IPv4 addresses, and *TXT* entries are included to allow arbitrary other information to be stored along with domain names.

The data for a zone starts with an *SOA*-type record, which contains the zone parameters that specify, for example, the version number and how often secondaries should refresh their copies. This is followed by a list of records of type *NS* specifying the name servers for the domain and a list of records of type *MX* giving the domain names of mail hosts, each prefixed by a number expressing its preference. For example, part of the database for the domain *dcs.qmul.ac.uk* at one point is shown in Figure 13.6, where the time to live *1D* means 1 day.

Further records of type *A* later in the database give the IP addresses for the two name servers *dns0* and *dns1*. The IP addresses of the mail hosts and the third name server are given in the databases corresponding to their domains.

Figure 13.6 DNS zone data records

domain name	*time to live*	*class*	*type*	*value*
dcs.qmul.ac.uk	*1D*	*IN*	*NS*	*dns0*
dcs.qmul.ac.uk	*1D*	*IN*	*NS*	*dns1*
dcs.qmul.ac.uk	*1D*	*IN*	*MX*	*1 mail1.qmul.ac.uk*
dcs.qmul.ac.uk	*1D*	*IN*	*MX*	*2 mail2.qmul.ac.uk*

The majority of the remainder of the records in a lower-level zone like *dcs.qmul.ac.uk* will be of type *A* and map the domain name of a computer onto its IP address. They may contain some aliases for the well-known services, for example:

domain name	*time to live*	*class*	*type*	*value*
www	*1D*	*IN*	*CNAME*	traffic
traffic	*1D*	*IN*	*A*	138.37.95.150

If the domain has any subdomains, there will be further records of type *NS* specifying their name servers, which will also have individual *A* entries. For example, at one point the database for *qmul.ac.uk* contained the following records for the name servers in its subdomain *dcs.qmul.ac.uk*:

domain name	*time to live*	*class*	*type*	*value*
dcs	*1D*	*IN*	*NS*	*dns0.dcs*
dns0.dcs	*1D*	*IN*	*A*	*138.37.88.249*
dcs	*1D*	*IN*	*NS*	*dns1.dcs*
dns1.dcs	*1D*	*IN*	*A*	*138.37.94.248*

Load sharing by name servers: At some sites, heavily used services such as the Web and FTP are supported by a group of computers on the same network. In this case, the same domain name is used for each member of the group. When a domain name is shared by several computers, there is one record for each computer in the group, giving its IP address. By default, the name server responds to queries for which multiple records match the requested name by returning the IP addresses according to a round-robin schedule. Successive clients are given access to different servers so that the servers can share the workload. Caching has a potential for spoiling this scheme, for once a non-authoritative name server or a client has the server's address in its cache it will continue to use it. To counteract this effect, the records are given a short time to live.

The BIND implementation of the DNS • The Berkeley Internet Name Domain (BIND) is an implementation of the DNS for computers running UNIX. Client programs link in library software as the resolver. DNS name server computers run the *named* daemon.

BIND allows for three categories of name server: primary servers, secondary servers and caching-only servers. The *named* program implements just one of these types, according to the contents of a configuration file. The first two categories are as described above. Caching-only servers read in from a configuration file sufficient names and addresses of authoritative servers to resolve any name. Thereafter, they only store this data and data that they learn by resolving names for clients.

A typical organization has one primary server, with one or more secondary servers that provide name serving on different local area networks at the site. Additionally, individual computers often run their own caching-only server, to reduce network traffic and speed up response times still further.

Discussion of the DNS • The DNS Internet implementation achieves relatively short average response times for lookups, considering the amount of naming data and the scale of the networks involved. We have seen that it achieves this by a combination of partitioning, replicating and caching naming data. The objects named are primarily computers, name servers and mail hosts. Computer (host) name–to–IP address mappings change relatively rarely, as do the identities of name servers and mail hosts, so caching and replication occur in a relatively clement environment.

The DNS allows naming data to become inconsistent. That is, if naming data is changed, then other servers may provide clients with stale data for periods on the order of days. None of the replication techniques explored in Chapter 18 is applied. However, inconsistency is of no consequence until such time as a client attempts to use stale data. The DNS does not address itself to how staleness of addresses is detected.

Apart from computers, the DNS also names one particular type of service – the mail service – on a per-domain basis. DNS assumes there to be only one mail service per addressed domain, so users do not have to include the name of this service explicitly in names. Electronic mail applications transparently select this service by using the appropriate type of query when contacting DNS servers.

In summary, the DNS stores a limited variety of naming data, but this is sufficient in so far as applications such as electronic mail impose their own naming schemes on top of domain names. It might be argued that the DNS database represents the lowest common denominator of what would be considered useful by the many user communities on the Internet. The DNS was not designed to be the only name service in the Internet; it coexists with local name and directory services that store data most pertinent to local needs (such as Sun's Network Information Service, which stores encoded passwords, for example, or Microsoft's Active Directory Services [www.microsoft.com I], which stores detailed information about all the resources within a domain).

What remains as a potential problem for the DNS design is its rigidity with respect to changes in the structure of the name space, and the lack of ability to customize the name space to suit local needs. These aspects of naming design are taken up by the case study of the Global Name Service in Section 13.4. But before that, we consider directory services.

13.3 Directory services

We have described how name services store collections of <*name, attribute*> pairs, and how the attributes are looked up from a name. It is natural to consider the dual of this arrangement, in which *attributes* are used as values to be looked up. In these services, textual names can be considered to be just another attribute. Sometimes users wish to find a particular person or resource, but they do not know its name, only some of its other attributes. For example, a user may ask: 'What is the name of the user with telephone number 020-555 9980?' Likewise, sometimes users require a service, but they are not concerned with what system entity supplies that service, as long as the service is conveniently accessible. For example, a user might ask, 'Which computers in this building are Macintoshes running the Mac OS X operating system?' or 'Where can I print a high-resolution colour image?'

A service that stores collections of bindings between names and attributes and that looks up entries that match attribute-based specifications is called a *directory service*. Examples are Microsoft's Active Directory Services, X.500 and its cousin LDAP (described in Section 13.5), Univers [Bowman *et al.* 1990] and Profile [Peterson 1988]. Directory services are sometimes called *yellow pages services*, and conventional name services are correspondingly called *white pages services*, in an analogy with the traditional types of telephone directory. Directory services are also sometimes known as *attribute-based name services*.

A directory service returns the sets of attributes of any objects found to match some specified attributes. So, for example, the request 'TelephoneNumber = 020 555 9980' might return {'Name = John Smith', 'TelephoneNumber = 020 555 9980', 'emailAddress = john@dcs.gormenghast.ac.uk', ...}. The client may specify that only a subset of the attributes is of interest – for example, just the email addresses of matching objects. X.500 and some other directory services also allow objects to be looked up by conventional hierarchic textual names. The Universal Directory and Discovery Service (UDDI), which was presented in Section 9.4, provides both white pages and yellow pages services to provide information about organizations and the web services they offer.

UDDI aside, the term *discovery service* normally denotes the special case of a directory service for services provided by devices in a spontaneous networking environment. As Section 1.3.2 described, devices in spontaneous networks are liable to connect and disconnect unpredictably. One core difference between a discovery service and other directory services is that the address of a directory service is normally well known and preconfigured in clients, whereas a device entering a spontaneous networking environment has to resort to multicast navigation, at least the first time it accesses the local discovery service. Section 19.2.1 describes discovery services in detail.

Attributes are clearly more powerful than names as designators of objects: programs can be written to select objects according to precise attribute specifications where names might not be known. Another advantage of attributes is that they do not expose the structure of organizations to the outside world, as do organizationally partitioned names. However, the relative simplicity of use of textual names makes them unlikely to be replaced by attribute-based naming in many applications.

13.4 Case study: The Global Name Service

A Global Name Service (GNS) was designed and implemented by Lampson and colleagues at the DEC Systems Research Center [Lampson 1986] to provide facilities for resource location, mail addressing and authentication. The design goals of the GNS have already been listed at the end of Section 13.1; they reflect the fact that a name service for use in an internetwork must support a naming database that may extend to include the names of millions of computers and (eventually) email addresses for billions of users. The designers of the GNS also recognized that the naming database is likely to have a long lifetime and that it must continue to operate effectively while it grows from small to large scale and while the network on which it is based evolves. The structure of the name space may change during that time to reflect changes in organizational structures. The service should accommodate changes in the names of the individuals, organizations and groups that it holds, and changes in the naming structure such as those that occur when one company is taken over by another. In this description, we focus on those features of the design that enable it to accommodate such changes.

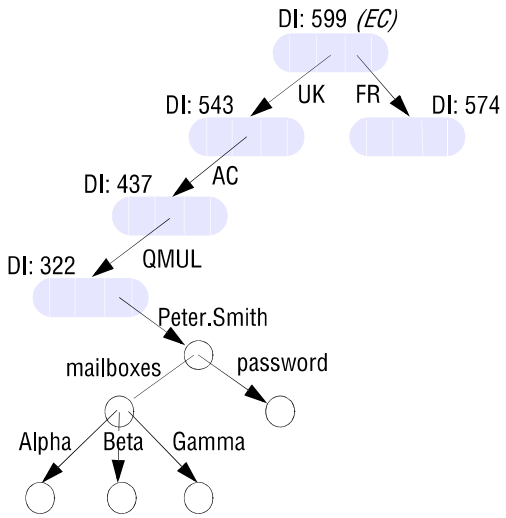
The potentially large naming database and the scale of the distributed environment in which the GNS is intended to operate make the use of caching essential and render it extremely difficult to maintain complete consistency between all copies of a database entry. The cache consistency strategy adopted relies on the assumption that updates to the database will be infrequent and that slow dissemination of updates is acceptable, since clients can detect and recover from the use of out-of-date naming data.

The GNS manages a naming database that is composed of a tree of directories holding names and values. Directories are named by multi-part pathnames referred to a root, or relative to a working directory, much like file names in a UNIX file system. Each directory is also assigned an integer, which serves as a unique *directory identifier* (DI). In this section, we use names in italics when referring to the DI of a directory, so that *EC* is the identifier of the EC directory. A directory contains a list of names and references. The values stored at the leaves of the directory tree are organized into *value trees*, so that the attributes associated with names can be structured values.

Names in the GNS have two parts: <*directory name*, *value name*>. The first part identifies a directory; the second refers to a value tree, or some portion of a value tree. For example, see Figure 13.7, in which the DIs are illustrated as small integers (although they are actually chosen from a range of integers to ensure uniqueness). The attributes of a user Peter.Smith in the directory QMUL would be stored in the value tree named <*EC*/UK/AC/QMUL, Peter.Smith>. The value tree includes a password, which can be referenced as <*EC*/UK/AC/QMUL, Peter.Smith/password>, and several mail addresses, each of which would be listed in the value tree as a single node with the name <*EC*/UK/AC/QMUL, Peter.Smith/mailboxes>.

The directory tree is partitioned and stored in many servers, with each partition replicated in several servers. The consistency of the tree is maintained in the face of two or more concurrent updates – for example, two users may simultaneously attempt to create entries with the same name, and only one should succeed. Replicated directories present a second consistency problem; this is addressed by an asynchronous update distribution algorithm that ensures eventual consistency, but with no guarantee that all copies are always current.

Figure 13.7 GNS directory tree and value tree for user Peter.Smith



Accommodating change • We now turn to the aspects of the design that are concerned with accommodating growth and change in the structure of the naming database. At the level of clients and administrators, growth is accommodated through extension of the directory tree in the usual manner. But we may wish to integrate the naming trees of two previously separate GNS services. For example, how could we integrate the database rooted at the *EC* directory shown in Figure 13.7 with another database for *NORTH AMERICA*? Figure 13.8 shows a new root, *WORLD*, introduced above the existing roots of the two trees to be merged. This is a straightforward technique, but how does it affect clients that continue to use names that are referred to what was 'the root' before integration took place? For example, </UK/AC/QMUL, Peter.Smith> is a name used by clients before integration. It is an absolute name (since it begins with the symbol for the root, '/'), but the root it refers to is *EC*, not *WORLD*. *EC* and *NORTH AMERICA* are *working roots* – initial contexts against which names beginning with the root '/' are to be looked up.

Figure 13.8 Merging trees under a new root

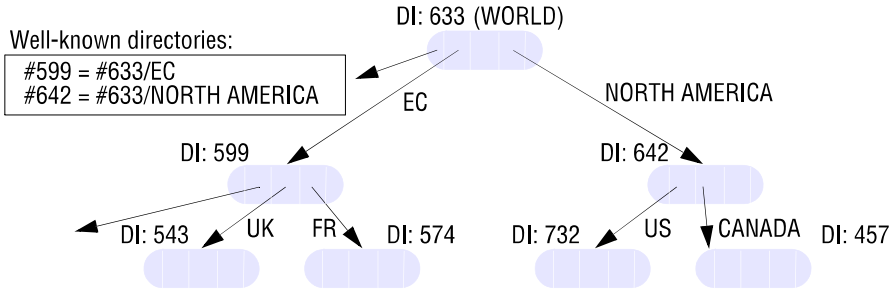
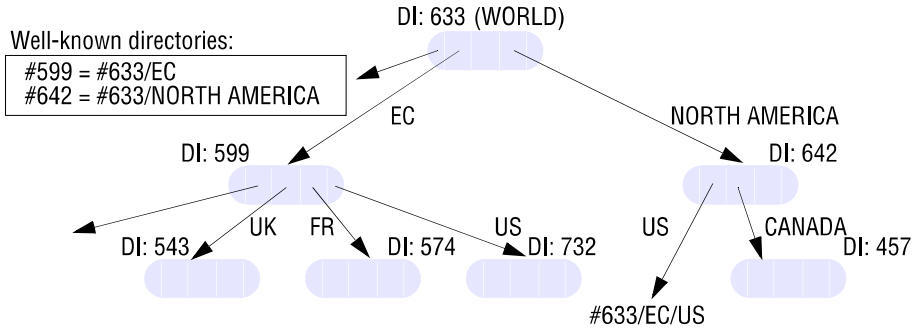


Figure 13.9 Restructuring the directory



The existence of unique directory identifiers can be used to solve this problem. The working root for each program must be identified as part of its execution environment (much as is done for a program's working directory). When a client in the European Community uses a name of the form </UK/AC/QMUL, Peter.Smith>, its local user agent, which is aware of the working root, prefixes the directory identifier *EC* (#599), thus producing the name <#599/UK/AC/QMUL, Peter.Smith>. The user agent passes this derived name in the lookup request to a GNS server. The user agent may deal similarly with relative names referred to working directories. Clients that are aware of the new configuration may also supply absolute names to the GNS server, which are referred to the conceptual super-root directory containing all directory identifiers – for example, <*WORLD*/EC/UK/AC/QMUL, Peter.Smith> – but the design cannot assume that all clients will be updated to take account of such a change.

The technique described above solves the logical problem, allowing users and client programs to continue to use names that are defined relative to an old root even when a new real root is inserted, but it leaves an implementation problem: in a distributed naming database that may contain millions of directories, how can the GNS service locate a directory given only its identifier, such as #599? The solution adopted by the GNS is to list those directories that are used as working roots, such as EC, in a table of 'well-known directories' held in the current real root directory of the naming database. Whenever the real root of the naming database changes, as it does in Figure 13.8, all GNS servers are informed of the new location of the real root. They can then interpret names of the form WORLD/EC/UK/AC/QMUL (referred to the real root) in the usual way, and they can interpret names of the form #599/UK/AC/QMUL by using the table of 'well-known directories' to translate them to full pathnames beginning at the real root.

The GNS also supports the restructuring of the database to accommodate organizational change. Suppose that the United States becomes part of the European Community (!). Figure 13.9 shows the new directory tree. But if the US subtree is simply moved to the EC directory, names beginning *WORLD*/NORTH AMERICA/US will no longer work. The solution adopted by the GNS is to insert a 'symbolic link' in place of the original US entry (shown in bold in Figure 13.9). The GNS directory lookup procedure interprets the link as a redirection to the US directory in its new location.

Discussion of the GNS • The GNS is descended from Grapevine [Birrell *et al.* 1982] and Clearinghouse [Oppen and Dalal 1983], two successful naming systems developed primarily for the purposes of mail delivery by the Xerox Corporation. The GNS successfully addresses needs for scalability and reconfigurability, but the solution adopted for merging and moving directory trees results in a requirement for a database (the table of well-known directories) that must be replicated at every node. In a large-scale network, reconfigurations may occur at any level, and this table could grow to a large size, conflicting with the scalability goal.

13.5 Case study: The X.500 Directory Service

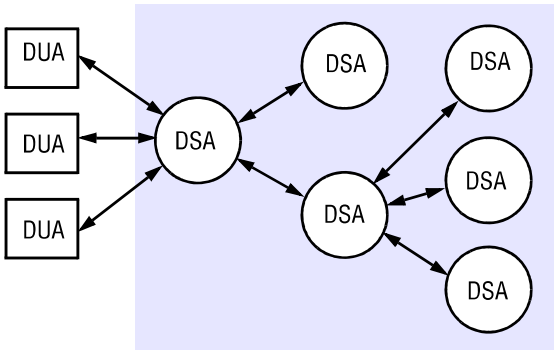
X.500 is a directory service in the sense defined in Section 13.3. It can be used in the same way as a conventional name service, but it is primarily used to satisfy descriptive queries and is designed to discover the names and attributes of other users or system resources. Users may have a variety of requirements for searching and browsing in a directory of network users, organizations and system resources to obtain information about the entities that the directory contains. The uses for such a service are likely to be quite diverse. They range from enquiries that are directly analogous to the use of telephone directories, such as a simple 'white pages' access to obtain a user's electronic mail address or a 'yellow pages' query aimed, for example, at obtaining the names and telephone numbers of garages specializing in the repair of a particular make of car, to the use of the directory to access personal details such as job roles, dietary habits or even photographic images of the individuals.

Such queries may originate from users, in the 'yellow pages'example mentioned above, or from processes, when they may be used to identify services to meet a functional requirement.

Individuals and organizations can use a directory service to make available a wide range of information about themselves and the resources that they wish to offer for use in the network. Users can search the directory for specific information with only partial knowledge of its name, structure or content.

The ITU and ISO standards organizations defined the *X.500 Directory Service* [ITU/ISO 1997] as a network service intended to meet these requirements. The standard refers to it as a service for access to information about 'real-world entities', but it is also likely to be used for access to information about hardware and software services and devices. X.500 is specified as an application-level service in the Open Systems Interconnection (OSI) set of standards, but its design does not depend to any significant extent on the other OSI standards, and it can be viewed as a design for a general-purpose directory service. We outline the design of the X.500 directory service and its implementation here. Readers interested in a more detailed description of X.500 and methods for its implementation are advised to study Rose's book on the subject [Rose 1992]. X.500 is also the basis for LDAP (discussed below), and it is used in the DCE directory service [OSF 1997].

Figure 13.10 X.500 service architecture



The data stored in X.500 servers is organized in a tree structure with named nodes, as in the case of the other name servers discussed in this chapter, but in X.500 a wide range of attributes are stored at each node in the tree, and access is possible not just by name but also by searching for entries with any required combination of attributes.

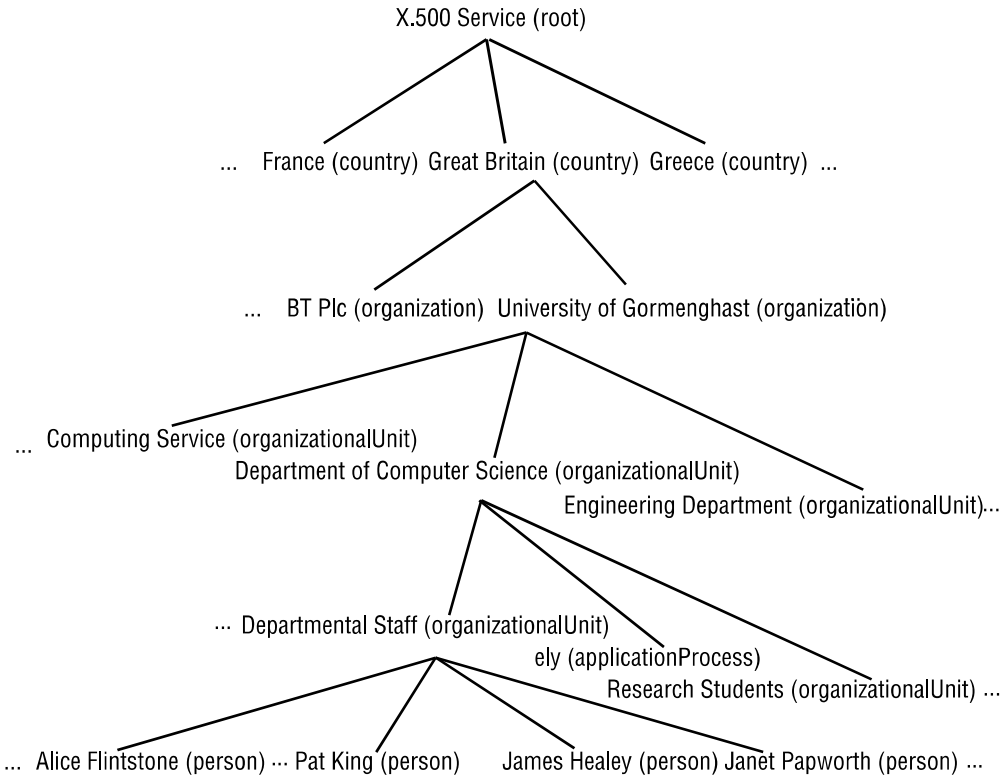
The X.500 name tree is called the *Directory Information Tree* (DIT), and the entire directory structure including the data associated with the nodes, is called the *Directory Information Base* (DIB). There is intended to be a single integrated DIB containing information provided by organizations throughout the world, with portions of the DIB located in individual X.500 servers. Typically, a medium-sized or large organization would provide at least one server. Clients access the directory by establishing a connection to a server and issuing access requests. Clients can contact any server with an enquiry. If the data required are not in the segment of the DIB held by the contacted server, it will either invoke other servers to resolve the query or redirect the client to another server.

In the terminology of the X.500 standard, servers are *Directory Service Agents* (DSAs), and their clients are termed *Directory User Agents* (DUAs). Figure 13.10 shows the software architecture and one of the several possible navigation models, with each DUA client process interacting with a single DSA process, which accesses other DSAs as necessary to satisfy requests.

Each entry in the DIB consists of a name and a set of attributes. As in other name servers, the full name of an entry corresponds to a path through the DIT from the root of the tree to the entry. In addition to full or *absolute* names, a DUA can establish a context, which includes a base node, and then use shorter relative names that give the path from the base node to the named entry.

Figure 13.11 shows the portion of the Directory Information Tree that includes the notional University of Gormenghast in Great Britain, and Figure 13.12 is one of the associated DIB entries. The data structure for the entries in the DIB and the DIT is very flexible. A DIB entry consists of a set of attributes, where an attribute has a *type* and one or more *values*. The type of each attribute is denoted by a type name (for example, *countryName*, *organizationName*, *commonName*, *telephoneNumber*, *mailbox*, *objectClass*). New attribute types can be defined if they are required. For each distinct

Figure 13.11 Part of the X.500 Directory Information Tree



type name there is a corresponding type definition, which includes a type description and a syntax definition in the ASN.1 notation (a standard notation for syntax definitions) defining representations for all permissible values of the type.

DIB entries are classified in a manner similar to the object class structures found in object-oriented programming languages. Each entry includes an *objectClass* attribute, which determines the class (or classes) of the object to which an entry refers. *Organization*, *organizationalPerson* and *document* are all examples of *objectClass* values. Further classes can be defined as they are required. The definition of a class determines which attributes are mandatory and which are optional for entries of the given class. The definitions of classes are organized in an inheritance hierarchy in which all classes except one (called *topClass*) must contain an *objectClass* attribute, and the value of the *objectClass* attribute must be the names of one or more classes. If there are several *objectClass* values, the object inherits the mandatory and optional attributes of each of the classes.

The name of a DIB entry (the name that determines its position in the DIT) is determined by selecting one or more of its attributes as *distinguished attributes*. The attributes selected for this purpose are referred to as the entry's *Distinguished Name* (DN).

Figure 13.12 An X.500 DIB Entry

info	
Alice Flintstone, Departmental Staff, Department of Computer Science, University of Gormenghast, GB	
commonName	*uid*
Alice.L.Flintstone	alf
Alice.Flintstone	*mail*
Alice Flintstone	alf@dcs.gormenghast.ac.uk
A. Flintstone	Alice.Flintstone@dcs.gormenghast.ac.uk
surname	*roomNumber*
Flintstone	Z42
telephoneNumber	*userClass*
+44 986 33 4604	Research Fellow

Now we can consider the methods by which the directory is accessed. There are two main types of access request:

read: An absolute or relative name (a *domain name* in X.500 terminology) for an entry is given, together with a list of attributes to be read (or an indication that all attributes are required). The DSA locates the named entry by navigating in the DIT, passing requests to other DSA servers where it does not hold relevant parts of the tree. It retrieves the required attributes and returns them to the client.

search: This is an attribute-based access request. A base name and a filter expression are supplied as arguments. The base name specifies the node in the DIT from which the search is to commence; the filter expression is a boolean expression that is to be evaluated for every node below the base node. The filter specifies a search criterion: a logical combination of tests on the values of any of the attributes in an entry. The *search* command returns a list of names (domain names) for all of the entries below the base node for which the filter evaluates to *TRUE*.

For example, a filter might be constructed and applied to find the *commonNames* of members of staff who occupy room Z42 in the Department of Computer Science at the University of Gormenghast (Figure 13.12). A *read* request could then be used to obtain any or all of the attributes of those DIB entries.

Searching can be quite costly when it is applied to large portions of the directory tree (which may reside in several servers). Additional arguments can be supplied to *search* to restrict the scope, the time for which the search is allowed to continue and the size of the list of entries that is returned.

Administration and updating of the DIB • The DSA interface includes operations for adding, deleting and modifying entries. Access control is provided for both queries and updating operations, so access to parts of the DIT may be restricted to certain users or classes of user.

The DIB is partitioned, with the expectation that each organization will provide at least one server holding the details of the entities in that organization. Portions of the DIB may be replicated in several servers.

As a standard (or a 'recommendation' in CCITT terminology), X.500 does not address implementation issues. However, it is quite clear that any implementation involving multiple servers in a wide area network must rely on extensive use of replication and caching techniques to avoid too much redirection of queries.

One implementation, described by Rose [1992], is a system developed at University College, London, known as QUIPU [Kille 1991]. In this implementation, both caching and replication are performed at the level of individual DIB entries, and at the level of collections of entries descended from the same node. It is assumed that values may become inconsistent after an update, and the time interval in which the consistency is restored may be several minutes. This form of update dissemination is generally considered acceptable for directory service applications.

Lightweight Directory Access Protocol • X.500's assumption that organizations would provide information about themselves in public directories within a common system has proved largely unfounded. Equally, its compexity has meant that its uptake has been relatively modest.

A group at the University of Michigan proposed a more lightweight approach called the *Lightweight Directory Access Protocol* (LDAP), in which a DUA accesses X.500 directory services directly over TCP/IP instead of the upper layers of the ISO protocol stack. This is described in RFC 2251 [Wahl *et al.* 1997]. LDAP also simplifies the interface to X.500 in other ways: for example, it provides a relatively simple API and it replaces ASN.1 encoding with textual encoding.

Although the LDAP specification is based on X.500, LDAP does not require it. An implementation may use any other directory server that obeys the simpler LDAP specification, as opposed to the X.500 specification. For example, Microsoft's Active Directory Services provides an LDAP interface.

Unlike X.500, LDAP has been widely adopted, particularly for intranet directory services. It provides secure access to directory data through authentication.

13.6 Summary

This chapter has described the design and implementation of name services in distributed systems. Name services store the attributes of objects in a distributed system – in particular, their addresses – and return these attributes when a textual name is supplied to be looked up.

The main requirements for the name service are an ability to handle an arbitrary number of names, a long lifetime, high availability, the isolation of faults and the tolerance of mistrust.

The primary design issue is the structure of the name space – the syntactic rules governing names. A related issue is the resolution model, which sets out the rules by which a multi-component name is resolved to a set of attributes. The set of bound names must be managed. Most designs consider the name space to be divided into domains –

discrete sections of the name space, each of which is associated with a single authority controlling the binding of names within it.

The implementation of the name service may span different organizations and user communities. The collection of bindings between names and attributes, in other words, is stored at multiple name servers, each of which stores at least part of the set of names within a naming domain. The question of navigation therefore arises – by what procedure can a name be resolved when the necessary information is stored at several sites? The types of navigation that are supported are iterative, multicast, recursive server-controlled and non-recursive server-controlled.

Another important aspect of the implementation of a name service is the use of replication and caching. Both of these assist in making the service highly available, and both also reduce the time taken to resolve a name.

This chapter has considered two main cases of name service design and implementation. The Domain Name System is widely used for naming computers and addressing electronic mail across the Internet; it achieves good response times through replication and caching. The Global Name Service is a design that has tackled the issue of reconfiguring the name space as organizational changes occur.

The chapter also considered directory services, which provide data about matching objects and services when clients supply attribute-based descriptions. X.500 is a model for directory services that can range in scope from individual organizations to global directories. It has been taken up more widely for use in intranets since the arrival of the LDAP software.

EXERCISES

13.1 Describe the names (including identifiers) and attributes used in a distributed file service such as NFS (see Chapter 12). *page 566*

13.2 Discuss the problems raised by the use of aliases in a name service, and indicate how, if at all, these may be overcome. *page 571*

13.3 Explain why iterative navigation is necessary in a name service in which different name spaces are partially integrated, such as the file-naming scheme provided by NFS. *page 574*

13.4 Describe the problem of unbound names in multicast navigation. What is implied by the installation of a server for responding to lookups of unbound names? *page 575*

13.5 How does caching help a name service's availability? *page 576*

13.6 Discuss the absence of a syntactic distinction (such as the use of a final '.') between absolute and relative names in the DNS. *page 571*

13.7 Investigate your local configuration of DNS domains and servers. You may find a program such as *dig* or *nslookup* installed, which enables you to carry out individual name server queries. *page 578*