MODULE 2

12.1 Introduction

In Chapters 1 and 2, we identified the sharing of resources as a key goal for distributed systems. The sharing of stored information is perhaps the most important aspect of distributed resource sharing. Mechanisms for data sharing take many forms and are described in several parts of this book. Web servers provide a restricted form of data sharing in which files stored locally, in file systems at the server or in servers on a local network, are made available to clients throughout the Internet. The design of large-scale wide area read-write file storage systems poses problems of load balancing, reliability, availability and security, whose resolution is the goal of the peer-to-peer file storage systems described in Chapter 10. Chapter 18 focuses on replicated storage systems that are suitable for applications requiring reliable access to data stored on systems where the availability of individual hosts cannot be guaranteed. In Chapter 20 we describe a media server that is designed to serve streams of video data to large numbers of users in real time. Chapter 21 describes a file system designed to support large-scale, data-intensive applications such as Internet search.

The requirements for sharing within local networks and intranets lead to a need for a different type of service – one that supports the persistent storage of data and programs of all types on behalf of clients and the consistent distribution of up-to-date data. The purpose of this chapter is to describe the architecture and implementation of these *basic* distributed file systems. We use the word 'basic' here to denote distributed file systems whose primary purpose is to emulate the functionality of a non-distributed file system for client programs running on multiple remote computers. They do not maintain multiple persistent replicas of files, nor do they support the bandwidth and timing guarantees required for multimedia data streaming – those requirements are addressed in later chapters. Basic distributed file systems provide an essential underpinning for organizational computing based on intranets.

File systems were originally developed for centralized computer systems and desktop computers as an operating system facility providing a convenient programming interface to disk storage. They subsequently acquired features such as access-control and file-locking mechanisms that made them useful for the sharing of data and programs. Distributed file systems support the sharing of information in the form of files and hardware resources in the form of persistent storage throughout an intranet. A well-designed file service provides access to files stored at a server with performance and reliability similar to, and in some cases better than, files stored on local disks. Their design is adapted to the performance and reliability characteristics of local networks, and hence they are most effective in providing shared persistent storage for use in intranets. The first file servers were developed by researchers in the 1970s [Birrell and Needham 1980, Mitchell and Dion 1982, Leach *et al.* 1983], and Sun's Network File System became available in the early 1980s [Sandberg *et al.* 1985, Callaghan 1999].

A file service enables programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer in an intranet. The concentration of persistent storage at a few servers reduces the need for local disk storage and (more importantly) enables economies to be made in the management and archiving of the persistent data owned by an organization. Other services, such as the name service, the user authentication service and the print service, can be more easily

Figure 12.1 Storage systems and their properties

	Sharing	*Persistence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	×	×	×	1	RAM
File system	×	✓	×	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Web server
Distributed shared memory	✓	×	✓	✓	Ivy (DSM, Ch. 6)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent State Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

Types of consistency:
1: strict one-copy ✓: slightly weaker guarantees 2: considerably weaker guarantees

implemented when they can call upon the file service to meet their needs for persistent storage. Web servers are reliant on filing systems for the storage of the web pages that they serve. In organizations that operate web servers for external and internal access via an intranet, the web servers often store and access the material from a local distributed file system.

With the advent of distributed object-oriented programming, a need arose for the persistent storage and distribution of shared objects. One way to achieve this is to serialize objects (in the manner described in Section 4.3.2) and to store and retrieve the serialized objects using files. But this method for achieving persistence and distribution is impractical for rapidly changing objects, so several more direct approaches have been developed. Java remote object invocation and CORBA ORBs provide access to remote, shared objects, but neither of these ensures the persistence of the objects, nor are the distributed objects replicated.

Figure 12.1 provides an overview of types of storage system. In addition to those already mentioned, the table includes distributed shared memory (DSM) systems and persistent object stores. DSM was described in Chapter 6. It provides an emulation of a shared memory by the replication of memory pages or segments at each host, but it does not necessarily provide automatic persistence. Persistent object stores were introduced in Chapter 5. They aim to provide persistence for distributed shared objects. Examples include the CORBA Persistent State Service (see Chapter 8) and persistent extensions to Java [Jordan 1996, java.sun.com VIII]. Some research projects have developed in platforms that support the automatic replication and persistent storage of objects (for example, PerDiS [Ferreira *et al*. 2000] and Khazana [Carter *et al*. 1998]). Peer-to-peer storage systems offer scalability to support client loads much larger than the systems described in this chaper, but they incur high performance costs in providing secure access control and consistency between updatable replicas.

Figure 12.2 File system modules

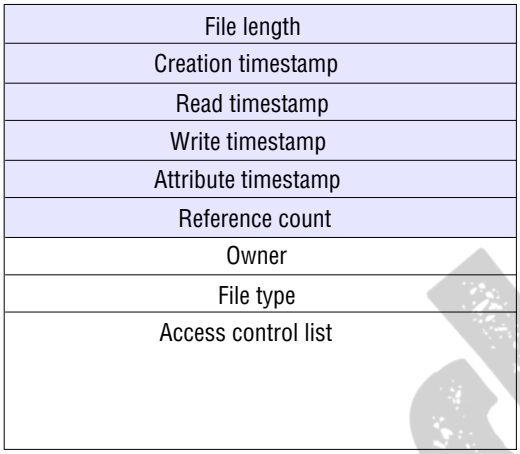
Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	performs disk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

The *consistency* column indicates whether mechanisms exist for the maintenance of consistency between multiple copies of data when updates occur. Virtually all storage systems rely on the use of caching to optimize the performance of programs. Caching was first applied to main memory and non-distributed file systems, and for those the consistency is strict (denoted by a '1', for one-copy consistency in Figure 12.1) – programs cannot observe any discrepancies between cached copies and stored data after an update. When distributed replicas are used, strict consistency is more difficult to achieve. Distributed file systems such as Sun NFS and the Andrew File System cache copies of portions of files at client computers, and they adopt specific consistency mechanisms to maintain an approximation to strict consistency – this is indicated by a tick (✓) in the consistency column of Figure 12.1. We discuss these mechanisms and the degree to which they deviate from strict consistency in Sections 12.3 and 12.4.

The Web uses caching extensively both at client computers and at proxy servers maintained by user organizations. The consistency between the copies stored at web proxies and client caches and the original server is only maintained by explicit user actions. Clients are not notified when a page stored at the original server is updated; they must perform explicit checks to keep their local copies up-to-date. This serves the purposes of web browsing adequately, but it does not support the development of cooperative applications such as a shared distributed whiteboard. The consistency mechanisms used in DSM systems are discussed in depth on the companion web site to the book [www.cdk5.net]. Persistent object systems vary considerably in their approach to caching and consistency. The CORBA and Persistent Java schemes maintain single copies of persistent objects, and remote invocation is required to access them, so the only consistency issue is between the persistent copy of an object on disk and the active copy in memory, which is not visible to remote clients. The PerDiS and Khazana projects that we mentioned above maintain cached replicas of objects and employ quite elaborate consistency mechanisms to produce forms of consistency similar to those found in DSM systems.

Having introduced some wider issues relating to storage and distribution of persistent and non-persistent data, we now return to the main topic of this chapter – the design of basic distributed file systems. We describe some relevant characteristics of (non-distributed) file systems in Section 12.1.1 and the requirements for distributed file systems in Section 12.1.2. Section 12.1.3 introduces the case studies that will be used throughout the chapter. In Section 12.2, we define an abstract model for a basic

Figure 12.3 File attribute record structure



distributed file service, including a set of programming interfaces. Sun NFS is described in Section 12.3; it shares many of the features of the abstract model. In Section 12.4 we describe the Andrew File System, a widely used system that employs substantially different caching and consistency mechanisms. Section 12.5 reviews some recent developments in the design of file services.

The systems described in this chapter do not cover the full spectrum of distributed file and data management systems. Several systems with more advanced characteristics will be described later in the book. Chapter 18 includes a description of Coda, a distributed file system that maintains persistent replicas of files for reliability, availability and disconnected working. Bayou, a distributed data management system that provides a weakly consistent form of replication for high availability, is also covered in Chapter 18. Chapter 20 covers the Tiger video file server, which is designed to provide timely delivery of streams of data to large numbers of clients. Chapter 21 describes the Google File System (GFS), a file system designed specifically to support large-scale, data-intensive applications including Internet search.

12.1.1 Characteristics of file systems

File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files. They provide a programming interface that characterizes the file abstraction, freeing programmers from concern with the details of storage allocation and layout. Files are stored on disks or other non-volatile storage media.

Files contain both *data* and *attributes*. The data consist of a sequence of data items (typically 8-bit bytes), accessible by operations to read and write any portion of the sequence. The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists. A typical attribute record structure is illustrated in Figure 12.3. The shaded attributes are managed by the file system and are not normally updatable by user programs.

Figure 12.4 UNIX file system operations

filedes = open(name, mode) *filedes = creat(name, mode)*	Opens an existing file with the given *name*. Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = close(filedes)	Closes the open file *filedes*.
count = read(filedes, buffer, n) *count = write(filedes, buffer, n)*	Transfers *n* bytes from the file referenced by *filedes* to *buffer*. Transfers *n* bytes to the file referenced by *filedes* from *buffer*. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to *offset* (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*).
status = stat(name, buffer)	Puts the file attributes for file *name* into *buffer*.

File systems are designed to store and manage large numbers of files, with facilities for creating, naming and deleting files. The naming of files is supported by the use of directories. A *directory* is a file, often of a special type, that provides a mapping from text names to internal file identifiers. Directories may include the names of other directories, leading to the familiar hierarchic file-naming scheme and the multi-part *pathnames* for files used in UNIX and other operating systems. File systems also take responsibility for the control of access to files, restricting access to files according to users' authorizations and the type of access requested (reading, updating, executing and so on).

The term *metadata* is often used to refer to all of the extra information stored by a file system that is needed for the management of files. It includes file attributes, directories and all the other persistent information used by the file system.

Figure 12.2 shows a typical layered module structure for the implementation of a non-distributed file system in a conventional operating system. Each layer depends only on the layers below it. The implementation of a distributed file service requires all of the components shown there, with additional components to deal with client-server communication and with the distributed naming and location of files.

File system operations • Figure 12.4 summarizes the main operations on files that are available to applications in UNIX systems. These are the system calls implemented by the kernel; application programmers usually access them through procedure libraries such as the C Standard Input/Output Library or the Java file classes. We give the primitives here as an indication of the operations that file services are expected to support and for comparison with the file service interfaces that we shall introduce below.

The UNIX operations are based on a programming model in which some file state information is stored by the file system for each running program. This consists of a list of currently open files with a read-write pointer for each, giving the position within the file at which the next read or write operation will be applied.

The file system is responsible for applying access control for files. In local file systems such as UNIX, it does so when each file is opened, checking the rights allowed for the user's identity in the access control list against the *mode* of access requested in the *open* system call. If the rights match the mode, the file is opened and the *mode* is recorded in the open file state information.

12.1.2 Distributed file system requirements

Many of the requirements and potential pitfalls in the design of distributed services were first observed in the early development of distributed file systems. Initially, they offered access transparency and location transparency; performance, scalability, concurrency control, fault tolerance and security requirements emerged and were met in subsequent phases of development. We discuss these and related requirements in the following subsections.

Transparency • The file service is usually the most heavily loaded service in an intranet, so its functionality and performance are critical. The design of the file service should support many of the transparency requirements for distributed systems identified in Section 1.5.7. The design must balance the flexibility and scalability that derive from transparency against software complexity and performance. The following forms of transparency are partially or wholly addressed by current file services:

Access transparency: Client programs should be unaware of the distribution of files. A single set of operations is provided for access to local and remote files. Programs written to operate on local files are able to access remote files without modification.

Location transparency: Client programs should see a uniform file name space. Files or groups of files may be relocated without changing their pathnames, and user programs see the same name space wherever they are executed.

Mobility transparency: Neither client programs nor system administration tables in client nodes need to be changed when files are moved. This allows file mobility – files or, more commonly, sets or volumes of files may be moved, either by system administrators or automatically.

Performance transparency: Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.

Scaling transparency: The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.

Concurrent file updates • Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file. This is the well-known issue of concurrency control, discussed in detail in Chapter 16. The need for concurrency control for access to shared data in many applications is widely accepted and techniques are known for its implementation, but they are costly. Most current file

services follow modern UNIX standards in providing advisory or mandatory file- or record-level locking.

File replication • In a file service that supports replication, a file may be represented by several copies of its contents at different locations. This has two benefits – it enables multiple servers to share the load of providing a service to clients accessing the same set of files, enhancing the scalability of the service, and it enhances fault tolerance by enabling clients to locate another server that holds a copy of the file when one has failed. Few file services support replication fully, but most support the caching of files or portions of files locally, a limited form of replication. The replication of data is discussed in Chapter 18, which includes a description of the Coda replicated file service.

Hardware and operating system heterogeneity • The service interfaces should be defined so that client and server software can be implemented for different operating systems and computers. This requirement is an important aspect of openness.

Fault tolerance • The central role of the file service in distributed systems makes it essential that the service continue to operate in the face of client and server failures. Fortunately, a moderately fault-tolerant design is straightforward for simple servers. To cope with transient communication failures, the design can be based on *at-most-once* invocation semantics (see Section 5.3.1); or it can use the simpler *at-least-once* semantics with a server protocol designed in terms of *idempotent* operations, ensuring that duplicated requests do not result in invalid updates to files. The servers can be *stateless*, so that they can be restarted and the service restored after a failure without any need to recover previous state. Tolerance of disconnection or server failures requires file replication, which is more difficult to achieve and will be discussed in Chapter 18.

Consistency • Conventional file systems such as that provided in UNIX offer *one-copy update semantics*. This refers to a model for concurrent access to files in which the file contents seen by all of the processes accessing or updating a given file are those that they would see if only a single copy of the file contents existed. When files are replicated or cached at different sites, there is an inevitable delay in the propagation of modifications made at one site to all of the other sites that hold copies, and this may result in some deviation from one-copy semantics.

Security • Virtually all file systems provide access-control mechanisms based on the use of access control lists. In distributed file systems, there is a need to authenticate client requests so that access control at the server is based on correct user identities and to protect the contents of request and reply messages with digital signatures and (optionally) encryption of secret data. We discuss the impact of these requirements in the case studies later in this chapter.

Efficiency • A distributed file service should offer facilities that are of at least the same power and generality as those found in conventional file systems and should achieve a comparable level of performance. Birrell and Needham [1980] expressed their design aims for the Cambridge File Server (CFS) in these terms:

> We would wish to have a simple, low-level file server in order to share an expensive resource, namely a disk, whilst leaving us free to design the filing system most appropriate to a particular client, but we would wish also to have available a high-level system shared between clients.

The changed economics of disk storage have reduced the significance of their first goal, but their perception of the need for a range of services addressing the requirements of clients with different goals remains and can best be addressed by a modular architecture of the type outlined above.

The techniques used for the implementation of file services are an important part of the design of distributed systems. A distributed file system should provide a service that is comparable with, or better than, local file systems in performance and reliability. It must be convenient to administer, providing operations and tools that enable system administrators to install and operate the system conveniently.

12.1.3 Case studies

We have constructed an abstract model for a file service to act as an introductory example, separating implementation concerns and providing a simplified model. We describe the Sun Network File System in some detail, drawing on our simpler abstract model to clarify its architecture. The Andrew File System is then described, providing a view of a distributed file system that takes a different approach to scalability and consistency maintenance.

File service architecture • This is an abstract architectural model that underpins both NFS and AFS. It is based upon a division of responsibilities between three modules – a client module that emulates a conventional file system interface for application programs, and server modules, that perform operations for clients on directories and on files. The architecture is designed to enable a *stateless* implementation of the server module.

SUN NFS • Sun Microsystems's *Network File System* (NFS) has been widely adopted in industry and in academic environments since its introduction in 1985. The design and development of NFS were undertaken by staff at Sun Microsystems in 1984 [Sandberg *et al.* 1985, Sandberg 1987, Callaghan 1999]. Although several distributed file services had already been developed and used in universities and research laboratories, NFS was the first file service that was designed as a product. The design and implementation of NFS have achieved success both technically and commercially.

To encourage its adoption as a standard, the definitions of the key interfaces were placed in the public domain [Sun 1989], enabling other vendors to produce implementations, and the source code for a reference implementation was made available to other computer vendors under licence. It is now supported by many vendors, and the NFS protocol (version 3) is an Internet standard, defined in RFC 1813 [Callaghan *et al.* 1995]. Callaghan's book on NFS [Callaghan 1999] is an excellent source on the design and development of NFS and related topics.

NFS provides transparent access to remote files for client programs running on UNIX and other systems. The client-server relationship is symmetrical: each computer in an NFS network can act as both a client and a server, and the files at every machine can be made available for remote access by other machines. Any computer can be a server, exporting some of its files, and a client, accessing files on other machines. But it is common practice to configure larger installations with some machines as dedicated servers and others as workstations.

An important goal of NFS is to achieve a high level of support for hardware and operating system heterogeneity. The design is operating system–independent: client and server implementations exist for almost all operating systems and platforms, including all versions of Windows, Mac OS, Linux and every other version of UNIX. Implementations of NFS on high-performance multiprocessor hosts have been developed by several vendors, and these are widely used to meet storage requirements in intranets with many concurrent users.

Andrew File System • Andrew is a distributed computing environment developed at Carnegie Mellon University (CMU) for use as a campus computing and information system [Morris *et al.* 1986]. The design of the Andrew File System (henceforth abbreviated AFS) reflects an intention to support information sharing on a large scale by minimizing client-server communication. This is achieved by transferring whole files between server and client computers and caching them at clients until the server receives a more up-to-date version. We shall describe AFS-2, the first 'production' implementation, following the descriptions by Satyanarayanan [1989a, 1989b]. More recent descriptions can be found in Campbell [1997] and [Linux AFS].

AFS was initially implemented on a network of workstations and servers running BSD UNIX and the Mach operating system at CMU and was subsequently made available in commercial and public-domain versions. A public-domain implementation of AFS is available in the Linux operating system [Linux AFS]. AFS was adopted as the basis for the DCE/DFS file system in the Open Software Foundation's Distributed Computing Environment (DCE) [www.opengroup.org]. The design of DCE/DFS went beyond AFS in several important respects, which we outline in Section 12.5.

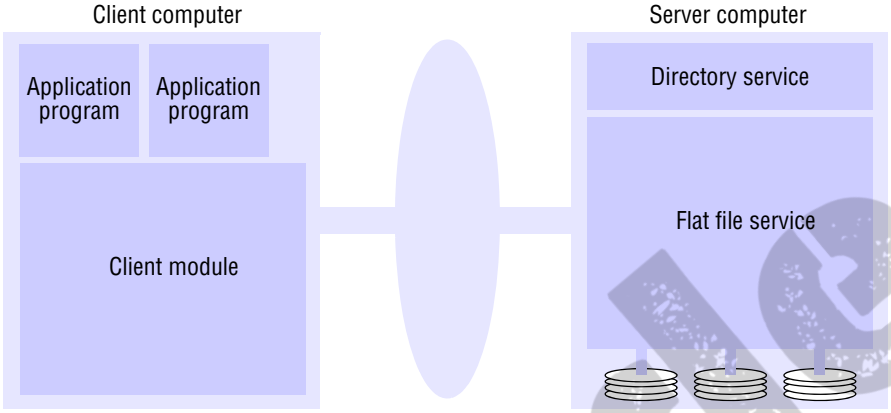
12.2 File service architecture

An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components – a *flat file service*, a *directory service* and a *client module*. The relevant modules and their relationships are shown in Figure 12.5. The flat file service and the directory service each export an interface for use by client programs, and their RPC interfaces, taken together, provide a comprehensive set of operations for access to files. The client module provides a single programming interface with operations on files similar to those found in conventional file systems. The design is *open* in the sense that different client modules can be used to implement different programming interfaces, simulating the file operations of a variety of different operating systems and optimizing the performance for different client and server hardware configurations.

The division of responsibilities between the modules can be defined as follows:

Flat file service • The flat file service is concerned with implementing operations on the contents of files. *Unique file identifiers* (UFIDs) are used to refer to files in all requests for flat file service operations. The division of responsibilities between the file service and the directory service is based upon the use of UFIDs. UFIDs are long sequences of bits chosen so that each file has a UFID that is unique among all of the files in a

Figure 12.5 File service architecture



distributed system. When the flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.

Directory service • The directory service provides a mapping between *text names* for files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to the directory service. The directory service provides the functions needed to generate directories, to add new file names to directories and to obtain UFIDs from directories. It is a client of the flat file service; its directory files are stored in files of the flat file service. When a hierarchic file-naming scheme is adopted, as in UNIX, directories hold references to other directories.

Client module • A client module runs in each client computer, integrating and extending the operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers. For example, in UNIX hosts, a client module would be provided that emulates the full set of UNIX file operations, interpreting UNIX multi-part file names by iterative requests to the directory service. The client module also holds information about the network locations of the flat file server and directory server processes. Finally, the client module can play an important role in achieving satisfactory performance through the implementation of a cache of recently used file blocks at the client.

Flat file service interface • Figure 12.6 contains a definition of the interface to a flat file service. This is the RPC interface used by client modules. It is not normally used directly by user-level programs. A *FileId* is invalid if the file that it refers to is not present in the server processing the request or if its access permissions are inappropriate for the operation requested. All of the procedures in the interface except *Create* throw exceptions if the *FileId* argument contains an invalid UFID or the user doesn't have sufficient access rights. These exceptions are omitted from the definition for clarity.

The most important operations are those for reading and writing. Both the *Read* and the *Write* operation require a parameter *i* specifying a position in the file. The *Read* operation copies the sequence of *n* data items beginning at item *i* from the specified file

Figure 12.6 Flat file service operations

Read(FileId, i, n) → *Data* — throws *BadPosition*	If $1 \le i \le Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \le i \le Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() → *FileId*	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) → *Attr*	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 12.3).

into *Data*, which is then returned to the client. The *Write* operation copies the sequence of data items in *Data* into the specified file beginning at item *i*, replacing the previous contents of the file at the corresponding position and extending the file if necessary.

Create creates a new, empty file and returns the UFID that is generated. *Delete* removes the specified file.

GetAttributes and *SetAttributes* enable clients to access the attribute record. *GetAttributes* is normally available to any client that is allowed to read the file. Access to the *SetAttributes* operation would normally be restricted to the directory service that provides access to the file. The values of the length and timestamp portions of the attribute record are not affected by *SetAttributes*; they are maintained separately by the flat file service itself.

Comparison with UNIX: Our interface and the UNIX file system primitives are functionally equivalent. It is a simple matter to construct a client module that emulates the UNIX system calls in terms of our flat file service and the directory service operations described in the next section.

In comparison with the UNIX interface, our flat file service has no *open* and *close* operations – files can be accessed immediately by quoting the appropriate UFID. The *Read* and *Write* requests in our interface include a parameter specifying a starting point within the file for each transfer, whereas the equivalent UNIX operations do not. In UNIX, each *read* or *write* operation starts at the current position of the read-write pointer, and the read-write pointer is advanced by the number of bytes transferred after each *read* or *write*. A *seek* operation is provided to enable the read-write pointer to be explicitly repositioned.

The interface to our flat file service differs from the UNIX file system interface mainly for reasons of fault tolerance:

Repeatable operations: With the exception of *Create*, the operations are *idempotent*, allowing the use of *at-least-once* RPC semantics – clients may repeat calls to which they receive no reply. Repeated execution of *Create* produces a different new file for each call.

Stateless servers: The interface is suitable for implementation by *stateless* servers. Stateless servers can be restarted after a failure and resume operation without any need for clients or the server to restore any state.

The UNIX file operations are neither idempotent nor consistent with the requirement for a stateless implementation. A read-write pointer is generated by the UNIX file system whenever a file is opened, and it is retained, together with the results of access-control checks, until the file is closed. The UNIX *read* and *write* operations are not idempotent; if an operation is accidentally repeated, the automatic advance of the read-write pointer results in access to a different portion of the file in the repeated operation. The read-write pointer is a hidden, client-related state variable. To mimic it in a file server, *open* and *close* operations would be needed, and the read-write pointer's value would have to be retained by the server as long as the relevant file is open. By eliminating the read-write pointer, we have eliminated most of the need for the file server to retain state information on behalf of specific clients.

Access control • In the UNIX file system, the user's access rights are checked against the access *mode* (read or write) requested in the *open* call (Figure 12.4 shows the UNIX file system API) and the file is opened only if the user has the necessary rights. The user identity (UID) used in the access rights check is retrieved during the user's earlier authenticated login and cannot be tampered with in non-distributed implementations. The resulting access rights are retained until the file is closed, and no further checks are required when subsequent operations on the same file are requested.

In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files. A user identity has to be passed with requests, and the server is vulnerable to forged identities. Furthermore, if the results of an access rights check were retained at the server and used for future accesses, the server would no longer be stateless. Two alternative approaches to the latter problem can be adopted:

- An access check is made whenever a file name is converted to a UFID, and the results are encoded in the form of a capability (see Section 11.2.4), which is returned to the client for submission with subsequent requests.
- A user identity is submitted with every client request, and access checks are performed by the server for every file operation.

Both methods enable stateless server implementation, and both have been used in distributed file systems. The second is more common; it is used in both NFS and AFS. Neither of these approaches overcomes the security problem concerning forged user identities, but we saw in Chapter 11 that this can be addressed by the use of digital signatures. Kerberos is an effective authentication scheme that has been applied to both NFS and AFS.

In our abstract model, we make no assumption about the method by which access control is implemented. The user identity is passed as an implicit parameter and can be used whenever it is needed.

Directory service interface • Figure 12.7 contains a definition of the RPC interface to a directory service. The primary purpose of the directory service is to provide a service for translating text names to UFIDs. In order to do so, it maintains directory files containing

Figure 12.7 Directory service operations

Lookup(Dir, Name) → *FileId* — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds *(Name, File)* to the directory and updates the file's attribute record. If *Name* is already in the directory, throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory, removes the entry containing *Name* from the directory. If *Name* is not in the directory, throws an exception.
GetNames(Dir, Pattern) → *NameSeq*	Returns all the text names in the directory that match the regular expression *Pattern*.

the mappings between text names for files and UFIDs. Each directory is stored as a conventional file with a UFID, so the directory service is a client of the file service.

We define only operations on individual directories. For each operation, a UFID for the file containing the directory is required (in the *Dir* parameter). The *Lookup* operation in the basic directory service performs a single *Name* → *UFID* translation. It is a building block for use in other services or in the client module to perform more complex translations, such as the hierarchic name interpretation found in UNIX. As before, exceptions caused by inadequate access rights are omitted from the definitions.

There are two operations for altering directories: *AddName* and *UnName*. *AddName* adds an entry to a directory and increments the reference count field in the file's attribute record.

UnName removes an entry from a directory and decrements the reference count. If this causes the reference count to reach zero, the file is removed. *GetNames* is provided to enable clients to examine the contents of directories and to implement pattern-matching operations on file names such as those found in the UNIX shell. It returns all or a subset of the names stored in a given directory. The names are selected by pattern matching against a regular expression supplied by the client.

The provision of pattern matching in the *GetNames* operation enables users to determine the names of one or more files by giving an incomplete specification of the characters in the names. A regular expression is a specification for a class of strings in the form of an expression containing a combination of literal substrings and symbols denoting variable characters or repeated occurrences of characters or substrings.

Hierarchic file system • A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure. Each directory holds the names of the files and other directories that are accessible from it. Any file or directory can be referenced using a *pathname* – a multi-part name that represents a path through

the tree. The root has a distinguished name, and each file or directory has a name in a directory. The UNIX file-naming scheme is not a strict hierarchy – files can have several names, and they can be in the same or different directories. This is implemented by a *link* operation, which adds a new name for a file to a specified directory.

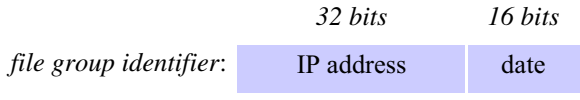
A UNIX-like file-naming system can be implemented by the client module using the flat file and directory services that we have defined. A tree-structured network of directories is constructed with files at the leaves and directories at the other nodes of the tree. The root of the tree is a directory with a 'well-known' UFID. Multiple names for files can be supported using the *AddName* operation and the reference count field in the attribute record.

A function can be provided in the client module that gets the UFID of a file given its pathname. The function interprets the pathname starting from the root, using *Lookup* to obtain the UFID of each directory in the path.

In a hierarchic directory service, the file attributes associated with files should include a type field that distinguishes between ordinary files and directories. This is used when following a path to ensure that each part of the name, except the last, refers to a directory.

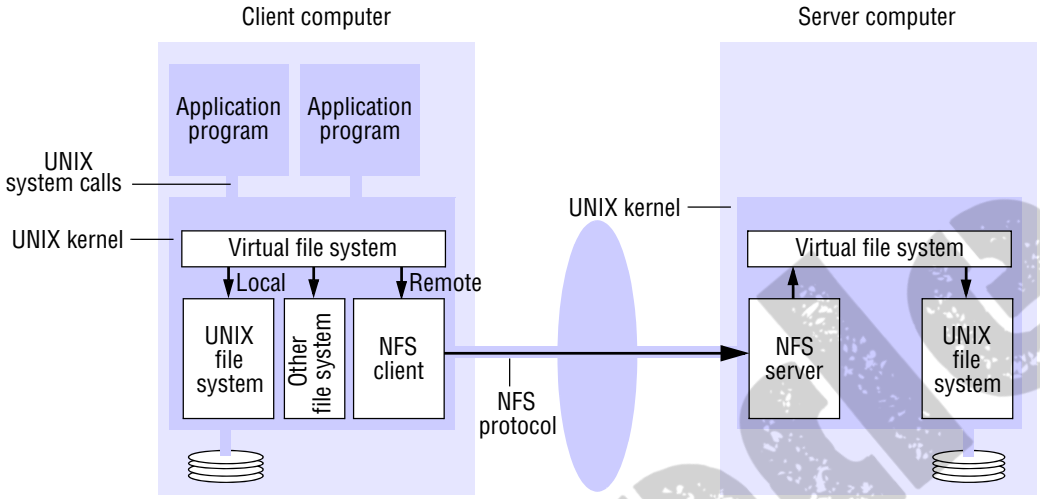
File groups • A *file group* is a collection of files located on a given server. A server may hold several file groups, and groups can be moved between servers, but a file cannot change the group to which it belongs. A similar construct called a *filesystem* is used in UNIX and in most other operating systems. (Terminology note: the single word *filesystem* refers to the set of files held in a storage device or partition, whereas the words *file system* refer to a software component that provides access to files.) File groups were originally introduced to support facilities for moving collections of files stored on removable media between computers. In a distributed file service, file groups support the allocation of files to file servers in larger logical units and enable the service to be implemented with files stored on several servers. In a distributed file system that supports file groups, the representation of UFIDs includes a file group identifier component, enabling the client module in each client computer to take responsibility for dispatching requests to the server that holds the relevant file group.

File group identifiers must be unique throughout a distributed system. Since file groups can be moved and distributed systems that are initially separate can be merged to form a single system, the only way to ensure that file group identifiers will always be distinct in a given system is to generate them with an algorithm that ensures global uniqueness. For example, whenever a new file group is created, a unique identifier can be generated by concatenating the 32-bit IP address of the host creating the new group with a 16-bit integer derived from the date, producing a unique 48-bit integer:



Note that the IP address *cannot* be used for the purpose of locating the file group, since it may be moved to another server. Instead, a mapping between group identifiers and servers should be maintained by the file service.

Figure 12.8 NFS architecture



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.

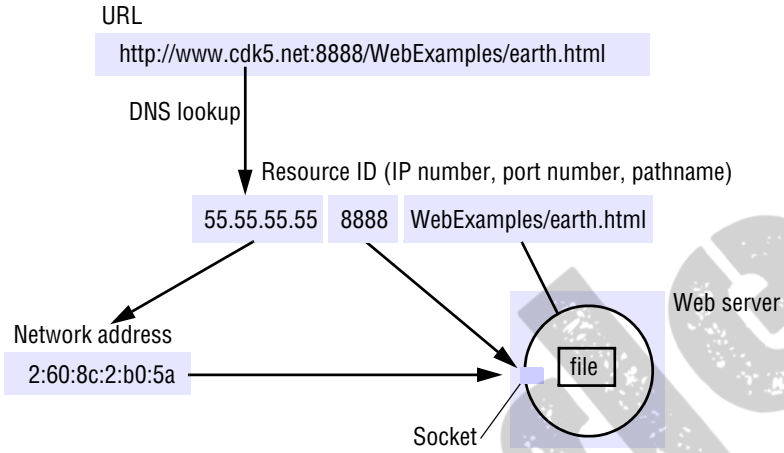
13.1.1 Names, addresses and other attributes

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.

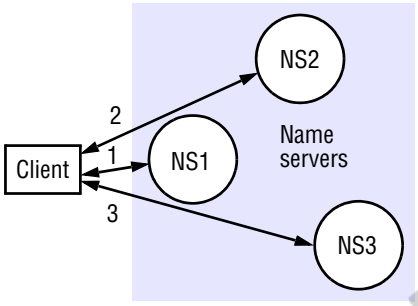
The partitioning of data implies that the local name server cannot answer all enquiries without the help of other name servers. For example, a name server in the *dcs.qmul.ac.uk* domain would not be able to supply the IP address of a computer in the domain *cs.purdue.edu* unless it was cached – certainly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

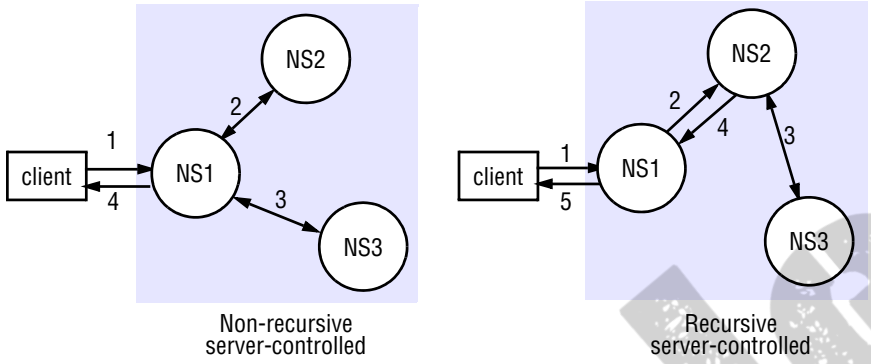
NFS also employs iterative navigation in the resolution of a file name, on a component-by-component basis (see Chapter 12). This is because the file service may encounter a symbolic link when resolving a name. A symbolic link must be interpreted in the client's file system name space because it may point 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.

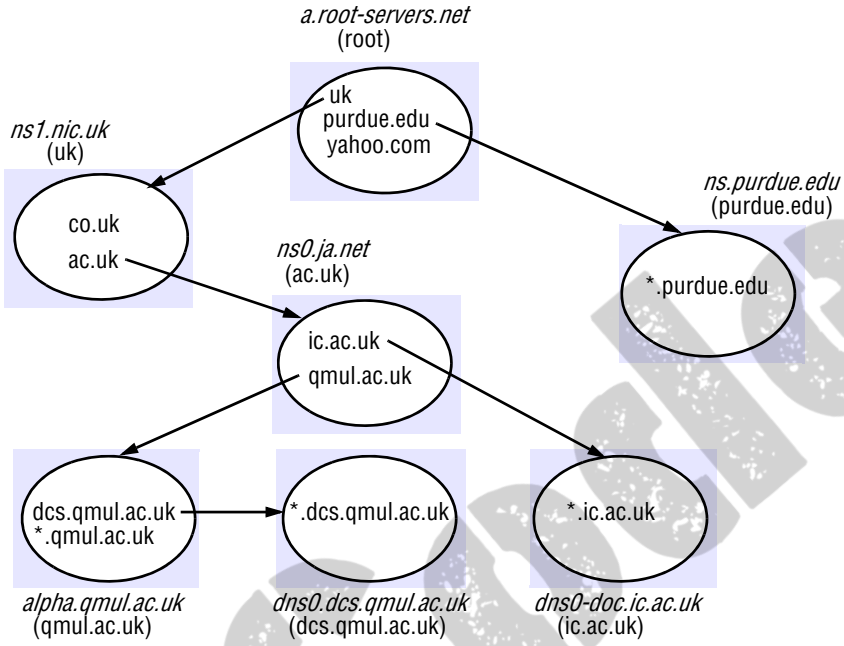
System administrators enter the data for a zone into a master file, which is the source of authoritative data for the zone. There are two types of server that are considered to provide authoritative data. A *primary* or *master server* reads zone data directly from a local master file. *Secondary servers* 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.