Roll No: 5117060
Name: Shivam Tiwari

DC ASSIGNMENT 04

Q1) EXPLAIN THE FEATURES OF A GOOD DISTRIBUTED FILE SYSTEM.

Answer

A good distributed file system should have the features described below.

1. Transparency.

The following four types of transparencies are desirable:

- **Structure transparency.** Although not necessary, for performance, scalability, and reliability reasons, a distributed file system normally uses multiple file servers. Each file server is normally a user process or sometimes a kernel process that is responsible for controlling a set of secondary storage devices (used for file storage) of the node on which it runs. In multiple file servers, the multiplicity of file servers should be transparent to the clients of a distributed file system. In particular, clients should not know the number or locations of the file servers and the storage devices. Ideally, a distributed file system should look to its clients like a conventional file system offered by a centralized, time-sharing operating system.
- **Access transparency.** Both local and remote files should be accessible in the same way. That is, the file system interface should not distinguish between local and remote files, and the file system should automatically locate an accessed file and arrange for the transport of data to the client's site.
- **Naming transparency.** The name of a file should give no hint as to where the file is located. Furthermore, a file should be allowed to move from one node to another in a distributed system without having to change the name of the file.
- **Replication transparency.** If a file is replicated on multiple nodes, both the existence of multiple copies and their locations should be hidden from the clients.

2. User mobility.

In a distributed system, a user should not be forced to work on a specific node but should have the flexibility to work on different nodes at different times. Furthermore, the performance characteristics of the file system should not discourage users from accessing their files from workstations other than the one at which they usually work. One way to support user mobility is to automatically bring a user's environment (e.g., the user's home directory) at the time of login to the node where the user logs in.

3. Performance.
The performance of a file system is usually measured as the average amount of time needed to satisfy client requests. In centralized file systems, this time includes the time for accessing the secondary storage device on which the file is stored and the CPU processing time. In a distributed file system, however, this time also includes network communication overhead when the accessed file is remote. Although acceptable performance is hard to quantify, it is desirable that the performance of a distributed file system should be comparable to that of a centralized file system. Users should never feel the need to make explicit file placement decisions to improve performance.

4. Simplicity and ease of use.
Several issues influence the simplicity and ease of use of a distributed file system. The most important issue is that the semantics of the distributed file system should be easy to understand. This implies that the user interface to the file system must be simple and the number of commands should be as small as possible. In an ideal design, the semantics of a distributed file system should be the same as that of a file system for a conventional centralized time-sharing system. Another important issue for ease of use is that the file system should be able to support the whole range of applications.

5. Scalability.
It is inevitable that a distributed system will grow with time since expanding the network by adding new machines or interconnecting two networks together is commonplace. Therefore, a good distributed file system should be designed to easily cope with the growth of nodes and users in the system. That is, such growth should not cause serious disruption of service or significant loss of performance to users. In short, a scalable design should withstand a high service load, accommodate the growth of the user community, and enable simple integration of added resources.

6. High availability.
A distributed file system should continue to function even when partial failures occur due to the failure of one or more components, such as a communication link failure, a machine failure, or a storage device crash. When partial failures occur, the file system may show degradation in performance, functionality. However, the degradation should be proportional, in some sense, to the failed components. For instance, it is quite acceptable that the failure causes temporary loss of service to small groups of users. High availability and scalability are mutually related properties. Both properties call for a design in which both control and data are distributed. This is because centralized entities such as a central controller or a central data repository introduce both a severe point of

failure and a performance bottleneck. Therefore, a highly available and scalable distributed file system should have multiple and independent file servers controlling multiple and independent storage devices. Replication of files at multiple servers is the primary mechanism for providing high availability.

7. High reliability.
In a good distributed file system, the probability of loss of stored data should be minimized as far as practicable. That is, users should not feel compelled to make backup copies of their files because of the unreliability of the system. Rather, the file system should automatically generate backup copies of critical files that can be used in the event of loss of the original ones. Stable storage is a popular technique used by several file systems for high reliability.

8. Data integrity.
A file is often shared by multiple users. For a shared file, the file system must guarantee the integrity of data stored in it. That is, concurrent access requests from multiple users who are competing to access the file must be properly synchronized by the use of some form of concurrency control mechanism. Atomic transactions are a high-level concurrency control mechanism often provided to the users by a file system for data integrity.

9. Security.
A distributed file system should be secure so that its users can be confident of the privacy of their data. Necessary security mechanisms must be implemented to protect information stored in a file system against unauthorized access. Furthermore, passing rights to access a file should be performed safely; that is, the receiver of rights should not be able to pass them further if he or she is not allowed to do that. A consequence of large-scale distributed systems is that the casual attitude toward security is not acceptable. A fundamental question is who enforces security. For this, the general design principle is that a system whose security depends on the integrity of the fewest possible entities is more likely to remain secure as it grows.

10. Heterogeneity.
As a consequence of the large scale, heterogeneity becomes inevitable in distributed systems. Heterogeneous distributed systems provide the flexibility to their users to use different computer platforms for different applications. For example, a user may use a supercomputer for simulations, a Macintosh for document processing, and a UNIX workstation for program development. Easy access to shared data across these diverse

platforms would substantially improve usability. Therefore, a distributed file system should be designed to allow a variety of workstations to participate in the sharing of files via the distributed file system. Another heterogeneity issue in file systems is the ability to accommodate several different storage media. Therefore, a distributed file system should be designed to allow the integration of a new type of workstation or storage media in a relatively simple manner.

Q2) EXPLAIN FILE MODELS AND FILE ACCESS MODELS VIZ, UPLOAD/DOWNLOAD MODEL OR CACHING MODEL, REMOTE ACCESS OR REMOTE SERVICE MODEL

Answer

Different file systems use different conceptual models of a file. The two most commonly used criteria for file modeling are structure and modifiability. File models based on these criteria are described below.

Unstructured and Structured Files

According to the simplest model, a file is an unstructured sequence of data. In this model, there is no substructure known to the file server and the contents of each file of the file system appear to the file server as an uninterpreted sequence of bytes. The operating system is not interested in the information stored in the files. Hence, the interpretation of the meaning and structure of the data stored in the files are entirely up to the application programs. UNIX and MS-DOS use this file model. Another file model that is rarely used nowadays is the structured file model. In this model, a file appears to the file server as an ordered sequence of records. Records of different files of the same file system can be of different size. Therefore, many types of files exist in a file system, each having different properties. In this model, a record is the smallest unit of file data that can be accessed, and the file system read or write operations are carried out on a set of records.

Structured files are again of two types-files with non indexed records and files with indexed records. In the former model, a file record is accessed by specifying its position within the file, for example, the fifth record from the beginning of the file or the second record from the end of the file. In the latter model, records have one or more key fields and can be addressed by specifying the values of the key fields. In file systems that allow indexed records, a file is maintained as a B-tree or other suitable data structure or a hash table is used to locate records quick

According to the modifiability criteria, files are of two types-mutable and immutable.

Most existing operating systems use the mutable file model. In this model, an update performed on a file overwrites on its old contents to produce the new contents. That is, a file is represented as a single stored sequence that is altered by each update operation. On the other hand, some more recent file systems, such as the Cedar File System (CFS) [Gifford et at 1988], use the immutable file model. In this model, a file cannot be modified once it has been created except to be deleted. The file versioning approach is normally used to implement file updates, and each file is represented by a history of immutable versions. That is, rather than updating the same file, a new version of the file is created each time a change is made to the file contents and the old version is retained unchanged. In practice, the use of storage space may be reduced by keeping only a record of the differences between the old and new versions rather than creating the entire file once again

The manner in which a client's request to access a file is serviced depends on the file accessing model used by the file system. The file-accessing model of a distributed file system mainly depends on two factors-the method used for accessing remote files and the unit of data access.

Distributed File Systems Accessing Models

A distributed file system may use one of the following models to service a client's file access request when the accessed file is a remote file:

1. Remote service model.

In this model, the processing of the client's request is performed at the server's node. That is, the client's request for file access is delivered to the server, the server machine performs the access request, and finally the result is forwarded back to the client. The access requests from the client and the server replies for the client are transferred across the network as messages. Notice that the data packing and communication overheads per message can be significant. Therefore, if the remote service model is used, the file server interface and the communication protocols must be designed carefully to minimize the overhead of generating messages as well as the number of messages that must be exchanged in order to satisfy a particular request.

2. Data-caching model.

In the remote service model, every remote file access request results in network traffic. The data-caching model attempts to reduce the amount of network traffic by taking advantage of the locality feature found in file accesses. In this model, if the data needed to satisfy the client's access request is not present locally, it is copied from the server's node to the client's node and is cached there. The client's request is processed on the client's node itself by using the cached data. Recently accessed data is retained in the cache for some time so that repeated accesses to the same data can be handled locally. A replacement policy, such as the least recently used (LRU), is used to keep the cache size bounded.

Q3) WRITE CASE STUDIES ON: A) DISTRIBUTED FILE SYSTEM B) NETWORK FILE SYSTEM, C) ANDREW FILE SYSTEM

Answer

Distributed File System
Two of the popular commercial file systems for distributed computing systems are Sun Microsystems' Network File System (NFS) and Open Software Foundation's Distributed File Service (DFS). DFS is one of the many services supported by the Distributed Computing Environment (DeE).
As a case study of how the concepts and the mechanisms described in this chapter can be used to build a distributed file system, DFS is briefly described below. DFS has been designed to allow multiple file systems to simultaneously exist on a node of a DeE system. For example, UNIX, NFS, and DFS's own file system can coexist on a node to provide three different types of file systems to the users of that node. The local file system of DFS that provides DFS on a single node is called Episode [Chutaniet al. 1992). As may be expected, when multiple file systems coexist on a node, features specific to DFS are available only to those users who use the Episode file system. DFS features are not available to the users of other file systems. DFS FIl. Model Like UNIX, DFS uses the unstructured file model in which a file is an unstructured sequence of data. A DFS file server handles the contents of a file as an uninterpreted sequence of bytes. A single file can contain up to 242 bytes.Like UNIX, DFS also uses the mutable file model. That is, for each file there is just one stored sequence that is altered by update operations. Note that although DFS uses the mutable file model, it has a facility called cloning (described later) that allows two stored sequences of a file to exist simultaneously; one of these is the version of the file before cloning and the other contains the changes made to the file after cloning.

DFS File System Model
As shown in Figure 9.20, the DFS file system model has four levels of aggregation. At the lowest level are individual files. At the next level are directories. Each directory usually contains several files. Above directories are filesets. Each fileset usually contains several

directories. Finally, at the highest level is an aggregate that usually contains multiple filesets. Each disk partition holds exactly one aggregate.Like a file system of UNIX, a fileset is a group of files that are administered (moved, replicated, backed up, etc.) as a set. However, unlike UNIX, a fileset is normally a subtree of a file system and not the entire file system tree. For example,a fileset may contain all the files of a single user, or all the files of a group of related users. With this difference, DI~S allows multiple filesets per disk partition, a management advantage over UNIX or NFS, which allow only a single file system per disk partition. The main advantage is that disk space can be more efficiently utilized by dynamically rebalancing the space occupancy of different partitions by moving filesets from nearly full partitions to relatively empty partitions as and when needed.

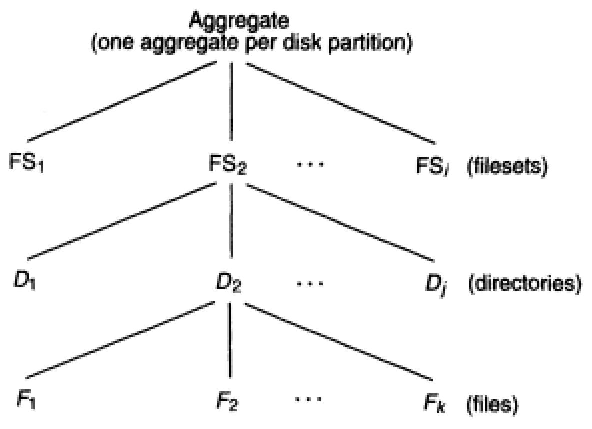


Fig. 9.20 Four levels of aggregation in the DFS file system model.

DFS File-Accessing Model

Distributed File Service relies on a client-server architecture and uses the data-caching model for file accessing. A machine in a DCE system is a DFS client, a DFS server, or both. A DFS client is a machine that uses files in filesets managed by DFS servers. The main software component of a DFS client machine is the DFS cache manager, which caches parts of recently used files to improve performance.On the other hand, a DFS server is a machine having its own disk storage that manages files in the files stored on its local disk and services requests received from DFS clients.A DFS server machine has the following software components:

1. Episode.
2. Token manager.
3. File exporter.
4. Fileset server.
5. Fileset location server.

6. Replication server.

DFS FILE.-Sharing Semantics

The strongest feature of DFS is that, in spite of using the data-caching model, it supports the single-site UNIX semantics. That is, every read operation on a file sees the effects of all previous write operations performed on that file. This is achieved in the manner described below.Recall that each DFS server has a component called token manger. The job of the token manager is to issue tokens to clients for file access requests and to keep track of which clients have been issued what types of tokens for which files. A client cannot perform the desired file operation on a piece of file data until it possesses the proper token.The use of a token-based approach to implement the single-site UNIX file-sharing semantics can best be illustrated with the help of an example (see Fig. 9.21). For simplicity, in this example we assume that there is only one type of token for all types of file access operations.

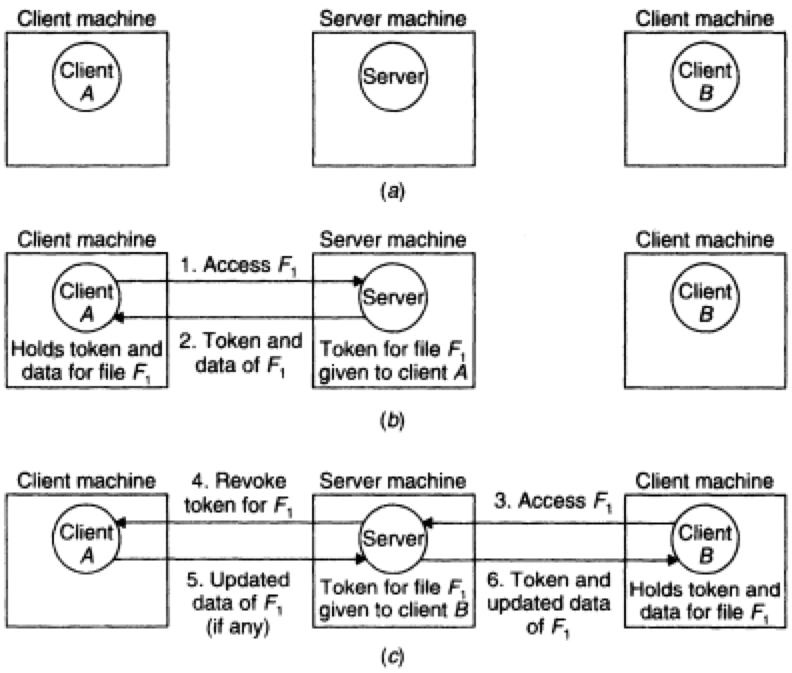

Client machine
Server machine
Client machine
Client A
Server
Client B
(a)
1. Access F_1
2. Token and data of F_1
Holds token and data for file F_1
Token for file F_1 given to client A
(b)
4. Revoke token for F_1
3. Access F_1
5. Updated data of F_1 (if any)
Token for file F_1 given to client B
6. Token and updated data of F_1
Holds token and data for file F_1
(c)

Fig. 9.21 Token-based approach of DFS for implementing the UNIX file-sharing semantics: (*a*) initial state of a server machine and two client machines; (*b*) state after client *A* receives the token and data for file F_1; (*c*) state after client *B* receives the token and data for file F_1.

Figure 9.21(a) shows the initial state of a server and two client machines. At this time, client A makes a request to the server for accessing file Fl. The server checks its state information to see if the token for file F1 has been given to any other client. It finds that it has not been given to any other client, so it sends the token and data of file FI to client A and makes a record of this information for future use. ClientA caches the received file data and then continues to perform file access operations on this data as many times as needed. Figure 9.21(b) shows the state of the server and client machines after client A receives the token and data for file Fl' Now suppose after some time that client B makes a request to the server for accessing the same file FJ. The server checks its state information and finds that the token for file F. has been given to client A. Therefore, it does not immediately send the token and data for file F1 to client B. Rather it first sends a message to client A asking back the token for file Fl. On receiving the revocation message, client A returns the token along with updated data of file F] (if any updates were made)and invalidates its cached data for file F). The server then updates its local copy of file F) (if needed) and now returns the token and up-to-date data of file F1 to client B. Client B caches the received file data and then continues to perform file access operations on this data as many times as needed. Figure 9.21(c) shows the state of the server and client machines after client B receives the token and data for file.F1• In this way, the single-system UNIX file-sharing semantics is achieved because the server issues the token for a file to only one client at a time. The client that possesses the token is assured exclusive access to the file.

File-Caching Scheme In DFS

We have already seen that in DFS recently accessed file data is cached by the cache manager of client machines. The local disk of a client machine is used for this purpose. However, in a diskless client machine, the local memory is used for caching file data. As shown in Figure 9.21, in DFS, modifications made to a cached file data are propagated to the file server only when the client receives a token revocation message for the file data. The same is true for cache validation scheme. That is, a cached file data of a client machine is invalidated (its cache entry is discarded) only when the client receives a token revocation message for the file data from the file server. As long as the client possesses the token for the specified operation on the file data, the cached data is valid and the client can continue to perform the specified operation on it. In effect, the approach used for cache validation is a server-initiated approach. The NFS schemes for modification propagation and cache validation have already been described in the previous section.

Replication And Cloning in DFS

Distributed File Service provides the facility to replicate files on multiple file servers. The unit of replication is a fileset, That is, all files of a fileset are replicated together. The

existence of multiple replicas of a file is transparent to normal users (client applications). That is, a filename is mapped to all file servers having a replica of the file.Therefore, given a filename, the fileset location server returns the addresses of all the file servers that have a replica of the fileset containing the file. DFS uses the explicit replication mechanism for replication control. That is, the number of replicas for a fileset and their locations are decided by the system administrator. The fileset server has a single command for replicating an entire fileset. The replication server of a server machine is responsible for maintaining the consistency of the replicas of filesets. The primary-copy protocol is used for this purpose.That is, for each replicated fileset, one copy is designated as the primary copy and all the others are secondary copies. Read operations on a file in a replicated fileset can be performed using any copy of the fileset, primary or secondary. But all update operations on a file in the fileset are directly performed only on the primary copy of the fileset. The the replication server ofthe primary copy periodically sends the updated versions of modified files to the replication servers of the secondary copies, which then update their own replicas. of the fileset.
NFS does not provide the facility to replicate files on multiple servers.In addition to allowing replication of filesets, DFS also provides the facility to clone filesets. This facility allows the creation of a new virtual copy of the fileset in another disk partition and the old copy is marked read only. This facility may be used by the system administrator to maintain an old version of a fileset, allowing the recovery of the old version of an inadvertently deleted file. For example, the system administrator might instruct the system to clone a fileset every day at midnight so that the previous day's work always remains intact in an old version of all files. If a user inadvertently deletes a file, he or she can always get the old version of the file and once again perform the current day's updates on it. Cloning of a fileset does not take much time because only a virtual copy of the fileset is made. That is, only the data structures for the files in the fileset are copied to the new partition and the file data is not copied. The old data structures in the original partition are marked read only. Therefore, both sets of data structures point to the same data blocks. When a file in the new fileset is updated, new data blocks are allocated for writing the updated version of the data and the corresponding file data structure in the new partition is updated to point to the new data blocks. A request to update a file in the original fileset is refused with an error message.

Fault Tolerance.

In addition to allowing replication of filesets, another important feature of DFS that helps in improving its fault tolerance ability is the use of the write-ahead log approach for recording file updates in a recoverable manner. In DFS, for every update made to a file, a log is written to the disk. A log entry contains the old value and the new value of the modified part of the file. When the system comes up after a crash, the log is used to check which changes have already been made to the file and which changes have not yet been made. Those that have not been made are the ones that were lost due to a system crash. These changes are now made to the file to bring it to a consistent state. If an update is lost because the crash occurred before the log for the update was recorded on the disk, it does not create any inconsistency because the lost update is treated as if the update was never performed on the file. Therefore the file is always in a consistent state after recovery.Notice that in the log-based crash recovery approach used in DFS, the recovery time is proportional to the length of the log and is independent of the size of the disk partition. This allows faster recovery than traditional systems like UNIX, in which the recovery time of a file system is proportional to the size of its disk partition. For fault tolerance, the main approach used by NFS is to use stateless file servers. Notice from Figure 9.21 that DFS servers are stateful because they have to keep track of the tokens issued to the clients

Atomic Transactions

The DeE does not provide transaction processing facility either as a part of DFS or as an independent component. This is mainly because DCE currently does not possess services needed for developing and running mission critical, distributed on-line transaction processing (OLTP) applications. For instance, OLTP applications require guaranteed data integrity, application programming interface with simplified transaction semantics, and the ability to extend programs to support RPCs that allow multiple processes to work together over the network to perform a common task. Such services are not currently supported by DCE. However, users of DeE who need the transaction processing facility can use Transarc Corporation's Encina OLTP technology. Encina expands on the DCE framework and provides a set of standards-based distributed services for simplifying the construction of reliable, distributed OLTP systems with guaranteed data integrity. In particular, the services offered by Encina for distributed OLTP include full data integrity with a transactional two-phase commit protocol, a high-level application programming interface with simplified transaction semantics, and additional transactional semantics required for achieving deterministic results with RPCs.In addition to Encina, two other transaction processing environments gaining popularity are Customer Information Control System (CIC'S) and Information Management System (IMS), both from IBM. CICS is already being used by more than 20,000 customers in more than 90 countries worldwide. Encina offers interoperability with IBM's CICS. Therefore, CICS can be implemented on top of the DCE and Encina technology.

User Interfaces to DFS

Distributed File Service supports the following types of user interfaces for different types of users:
1. File service interface. DFS uses native operating system commands for directory and file operations so that users do not need to learn new commands. For example, users on UNIX systems will use cd to change directory, ls to list directory contents, mkdir to create a new directory, and so on. DFS also has several commands that work only for its own file system (Episode). These include commands to check quotas of different filesets, to locate the server of a file, and so on. To access a file, a client may specify the file by its global name or by its cell relative name.
2. Application programming interface. The application programming interface to
DFS is very similar to UNIX. Therefore, application programmers can use standard file system calls like fopen () for opening a file, tread () to read from a file, fwrite () to write to a file, [close () to close a file, and so on. In fact, most existing software will work immediately by simply recompiling with the DFS libraries.
3. Administrative interface. The administrative interface of DFS provides commands that allow the system administrator to handle filesets, to install or remove DFS file servers,and to manipulate ACLs associated with files and directories. The commands for handling filesets are used by the system administrator to create, delete, move, replicate, clone, back up, or restore filesets. For example, the system administrator may move filesets from one server machine to other server machines to balance the load across all file server machines in a cell.On the other hand, the commands to install or remove file servers allow the system administrator to dynamically reconfigure the system as needs change. For example, the system administrator may notice that the DFS performance of a cell is not so good because there are too many clients and only a few file servers. In this case, he or she may install a new file server in the cell and move some of the filesets from already existing file servers of the cell to this file server.
Finally, the commands to manipulate ACLs are used by the system administrator to revoke some of the access permissions already given to some users or to give additional permissions to some users.

Network File System

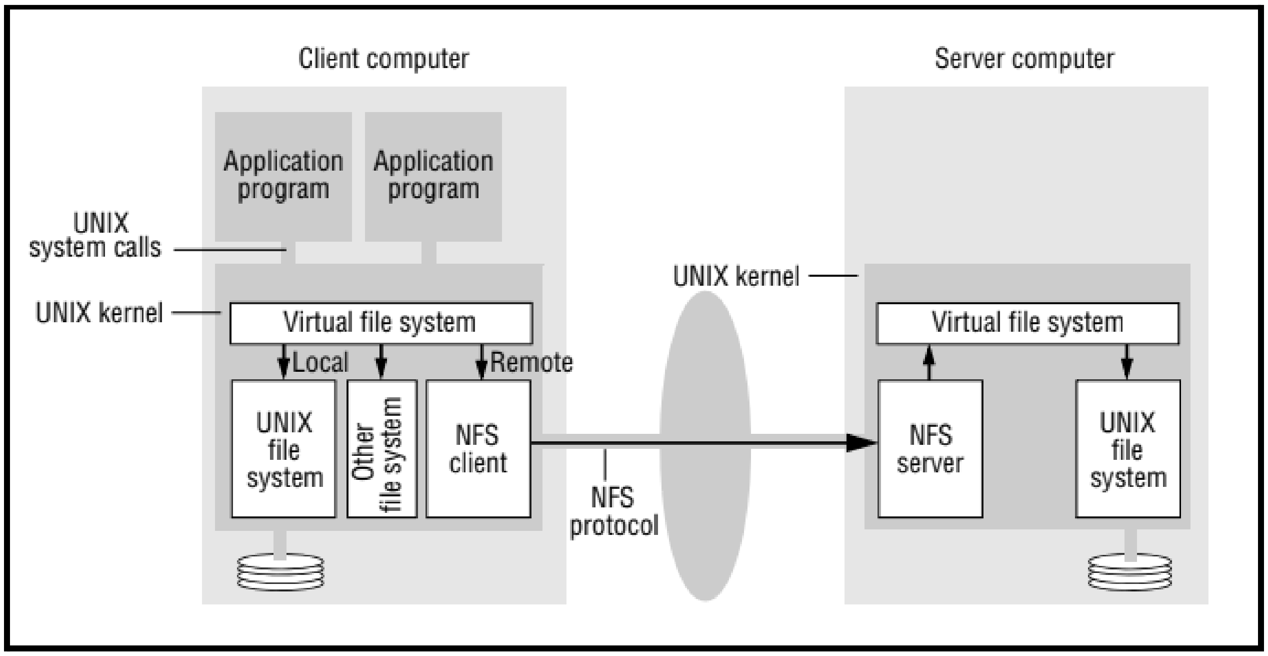


Figure 3.b.1: Architecture of Sun NFS

Figure 3.a.1 shows the architecture of Sun NFS. All implementations of NFS support the NFS protocol – a set of remote procedure calls that provide the means for clients to perform operations on a remote file store. The NFS protocol is operating system–independent but was originally developed for use in networks of UNIX systems, and we shall describe the UNIX implementation of the NFS protocol (version 3).

The NFS server module resides in the kernel on each computer that acts as an NFS server. Requests referring to files in a remote file system are translated by the client module to NFS protocol operations and then passed to the NFS server module at the computer holding the relevant file system.

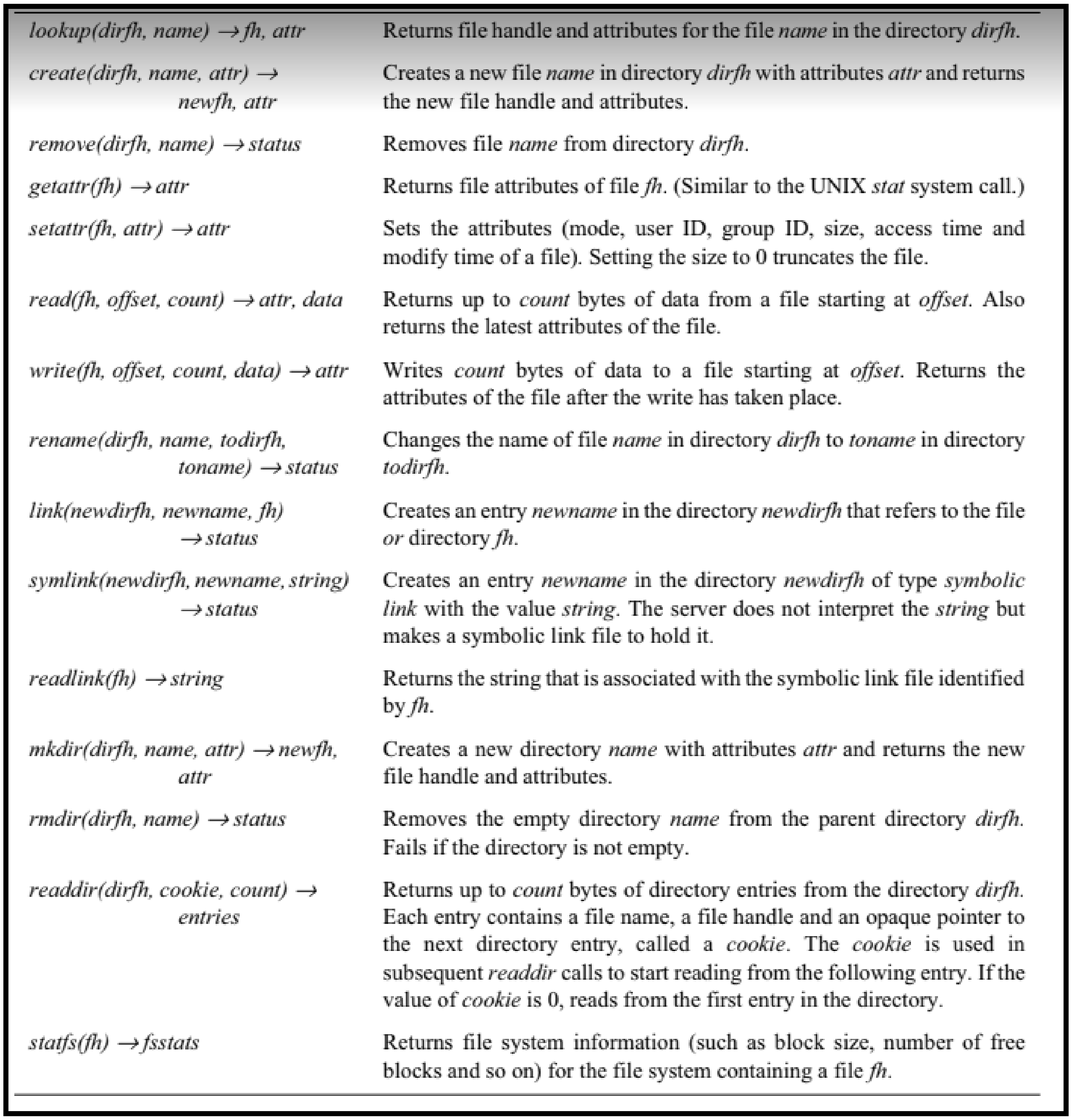
The NFS client and server modules communicate using remote procedure calls. Sun's RPC system was developed for use in NFS. It can be configured to use either UDP or TCP, and the NFS protocol is compatible with both. A port mapper service is included to enable clients to bind to services in a given host by name. The RPC interface to the NFS server is open: any process can send requests to an NFS server; if the requests are valid and they include valid user credentials, they will be acted upon. The submission of signed user credentials can be required as an optional security feature, as can the encryption of data for privacy and integrity.

Virtual file system • Figure 3.a.1 makes it clear that NFS provides access transparency: user programs can issue file operations for local or remote files without distinction. Other distributed file systems may be present that support UNIX system calls, and if so, they could be integrated in the same way.

The integration is achieved by a virtual file system (VFS) module, which has been added to the UNIX kernel to distinguish between local and remote files and to translate between the UNIX-independent file identifiers used by NFS and the internal file identifiers normally used in UNIX and other file systems. In addition, VFS keeps track of the file systems that are currently available both locally and remotely, and it passes each request to the appropriate local system module (the UNIX file system, the NFS client module or the service module for another file system).

Client integration • The NFS client module plays the role described for the client module in our architectural model, supplying an interface suitable for use by conventional application programs. But unlike our model client module, it emulates the semantics of the standard UNIX file system primitives precisely and is integrated with the UNIX kernel. It is integrated with the kernel and not supplied as a library for loading into client processes so that:

- user programs can access files via UNIX system calls without recompilation or reloading;
- a single client module serves all of the user-level processes, with a shared cache of recently used blocks (described below);
- the encryption key used to authenticate user IDs passed to the server can be retained in the kernel, preventing impersonation by user-level clients.



lookup(dirfh, name) → fh, attr	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) → newfh, attr	Creates a new file *name* in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) → status	Removes file *name* from directory *dirfh*.
getattr(fh) → attr	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
setattr(fh, attr) → attr	Sets the attributes (mode, user ID, group ID, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) → attr, data	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
write(fh, offset, count, data) → attr	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) → status	Changes the name of file *name* in directory *dirfh* to *toname* in directory *todirfh*.
link(newdirfh, newname, fh) → status	Creates an entry *newname* in the directory *newdirfh* that refers to the file *or* directory *fh*.
symlink(newdirfh, newname, string) → status	Creates an entry *newname* in the directory *newdirfh* of type *symbolic link* with the value *string*. The server does not interpret the *string* but makes a symbolic link file to hold it.
readlink(fh) → string	Returns the string that is associated with the symbolic link file identified by *fh*.
mkdir(dirfh, name, attr) → newfh, attr	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) → status	Removes the empty directory *name* from the parent directory *dirfh*. Fails if the directory is not empty.
readdir(dirfh, cookie, count) → entries	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle and an opaque pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the following entry. If the value of *cookie* is 0, reads from the first entry in the directory.
statfs(fh) → fsstats	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

Figure 3.b.2: NFS server operations (NFS version 3 protocol, simplified)

Access control and authentication • Unlike the conventional UNIX file system, the NFS server is stateless and does not keep files open on behalf of its clients. So the server must check the user's identity against the file's access permission attributes afresh on each request, to see whether the user is permitted to access the file in the manner requested. The Sun RPC protocol requires clients to send user authentication information (for example, the conventional UNIX 16-bit user ID and group ID) with each request and this is checked against the access permission in the file attributes.

NFS server interface • A simplified representation of the RPC interface provided by NFS version 3 servers (defined in RFC 1813 [Callaghan et al. 1995]) is shown in Figure 3.b.2. The NFS file access operations read, write, getattr and setattr are almost identical to the Read, Write, GetAttributes and SetAttributes operations defined for the flat file service model. The lookup operation and most of the other directory operations defined in Figure 3.b.2 are similar to those of a directory service model.

Mount service • The mounting of subtrees of remote filesystems by clients is supported by a separate mount service process that runs at user level on each NFS server computer. On each server, there is a file with a well-known name (/etc/exports) containing the names of local filesystems that are available for remote mounting. An access list is associated with each filesystem name indicating which hosts are permitted to mount the filesystem.

Pathname translation • UNIX file systems translate multi-part file path names to i-node references in a step-by-step process whenever the open, creator stat system calls are used. In NFS, pathnames cannot be translated at a server, because the name may cross a 'mount point' at the client – directories holding different parts of a multi-part name may reside in file systems at different servers. So pathnames are parsed, and their translation is performed in an iterative manner by the client. Each part of a name that refers to a remote-mounted directory is translated to a file handle using a separate lookup request to the remote server.

Automounter • The automounter was added to the UNIX implementation of NFS in order to mount a remote directory dynamically whenever an 'empty' mount point is referenced by a client. The original implementation of the automounter ran as a user-level UNIX process in each client computer. The automounter maintains a table of mount points (pathnames) with a reference to one or more NFS servers listed against each. It behaves like a local NFS server at the client machine.When the NFS client module attempts to resolve a pathname that includes one of these mount points, it passes to the local automounter a lookup() request that locates the required file system in its table and sends a 'probe' request to each server listed. The filesystem on the first server to respond is then mounted at the client using the normal mount service. The mounted filesystem is linked to the mount point using a symbolic link, so access to it will not result in further requests to the automounter. File access then proceeds in the normal way without further reference to the automounter unless there are no references to the symbolic link for several minutes. In the latter case, the automounter unmounts the remote filesystem.

Server caching • Caching in both the client and the server computer are indispensable features of NFS implementations in order to achieve adequate performance.

Client caching • The NFS client module caches the results of read, write, getattr, lookup and readdir operations in order to reduce the number of requests transmitted to servers. Client caching introduces the potential for different versions of files or portions of files to exist in different client nodes, because writes by a client do not result in the immediate updating of cached copies of the same file in other clients. Instead, clients are responsible for polling the server to check the currency of the cached data that they hold.

Case study: The Andrew File System

Like NFS, AFS provides transparent access to remote shared files for UNIX programs running on workstations. Access to AFS files is via the normal UNIX file primitives, enabling existing UNIX programs to access AFS files without modification or recompilation. AFS is compatible with NFS. AFS servers hold 'local' UNIX files, but the filing system in the servers is NFS-based, so files are referenced by NFS-style file handles rather than i-node numbers, and the files may be remotely accessed via NFS. AFS differs markedly from NFS in its design and implementation. The differences are primarily attributable to the identification of scalability as the most important design goal. AFS is designed to perform well with larger numbers of active users than other distributed file systems. The key strategy for achieving scalability is the caching of whole files in client nodes. AFS has two unusual design characteristics:

Whole-file serving:

The entire contents of directories and files are transmitted to client computers by AFS servers (in AFS-3, files larger than 64 kbytes are transferred in 64-kbyte chunks).

Whole-file caching:

Once a copy of a file or a chunk has been transferred to a client computer it is stored in a cache on the local disk. The cache contains several hundred of the files most recently used on that computer. The cache is permanent, surviving reboots of the client computer. Local copies of files are used to satisfy clients' open requests in preference to remote copies whenever possible.

Scenario

• Here is a simple scenario illustrating the operation of AFS:

1. When a user process in a client computer issues an open system call for a file in the shared file space and there is not a current copy of the file in the local cache, the server holding the file is located and is sent a request for a copy of the file.
2. The copy is stored in the local UNIX file system in the client computer. The copy is then opened and the resulting UNIX file descriptor is returned to the client.

3. Subsequent read, write and other operations on the file by processes in the client computer are applied to the local copy.
4. When the process in the client issues a close system call, if the local copy has been updated its contents are sent back to the server. The server updates the file contents and the timestamps on the file.

The copy on the client's local disk is retained in case it is needed again by a user-level process on the same workstation. We discuss the observed performance of AFS below, but we can make some general observations and predictions here based on the design characteristics described above:

• For shared files that are infrequently updated (such as those containing the code of UNIX commands and libraries) and for files that are normally accessed by only a single user (such as most of the files in a user's home directory and its subtree), locally cached copies are likely to remain valid for long periods – in the first case because they are not updated and in the second because if they are updated, the updated copy will be in the cache on the owner's workstation. These classes of file account for the overwhelming majority of file accesses.

Q4) WRITE A NOTE ON : A) NAME SERVICES AND B) DOMAIN NAME SYSTEM

Answer

Name Services

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 a 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.

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 namespace. 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.

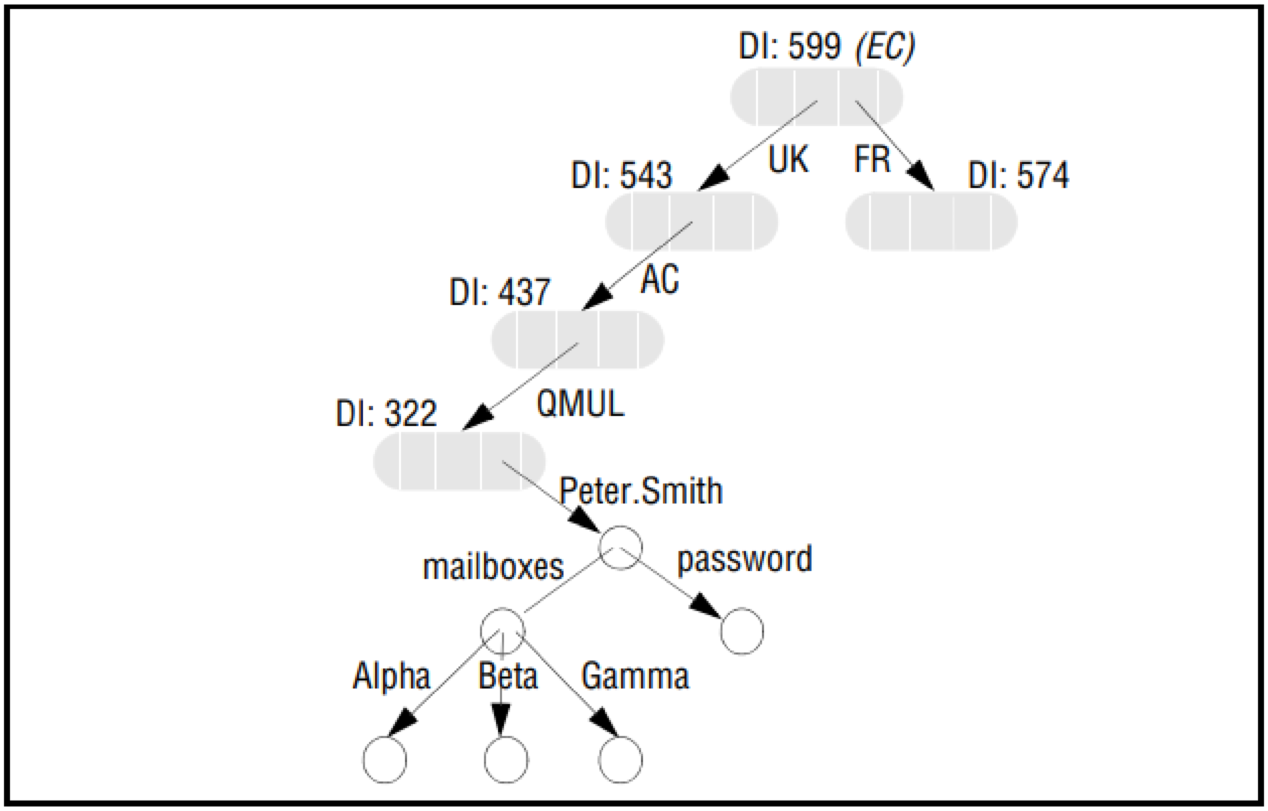
Q5) WRITE CASE STUDIES ON: A) THE GLOBAL NAME SERVICE B) THE X.500 DIRECTORY SERVICE C) DESIGNING DISTRIBUTED SYSTEMS : GOOGLE CASE STUDY.

Case study: The Global Name Service

A Global Name Service (GNS) was designed and implemented by Lampson and colleagues at the DEC Systems Research Center [Lampson 1986] to provide facilities for

resource location, mail addressing and authentication. The design goals of the GNS have already been listed at the end of Section 13.1; they reflect the fact that a name service for use in an internetwork must support a naming database that may extend to include the names of millions of computers and (eventually) email addresses for billions of users. The designers of the GNS also recognized that the naming database is likely to have a long lifetime and that it must continue to operate effectively while it grows from small to large scale and while the network on which it is based evolves. The structure of the name space may change during that time to reflect changes in organizational structures. The service should accommodate changes in the names of the individuals, organizations and groups that it holds, and changes in the naming structure such as those that occur when one company is taken over by another. In this description, we focus on those features of the design that enable it to accommodate such changes.
The potentially large naming database and the scale of the distributed environment in which the GNS is intended to operate make the use of caching essential and render it extremely difficult to maintain complete consistency between all copies of a database entry. The cache consistency strategy adopted relies on the assumption that updates to the database will be infrequent and that slow dissemination of updates is acceptable, since clients can detect and recover from the use of out-of-date naming data.
The GNS manages a naming database that is composed of a tree of directories holding names and values. Directories are named by multi-part path names referred to a root, or relative to a working directory, much like file names in a UNIX file system. Each directory is also assigned an integer, which serves as a unique directory identifier (DI). In this section, we use names in italics when referring to the DI of a directory, so that EC is the identifier of the EC directory. A directory contains a list of names and references. The values stored at the leaves of the directory tree are organized into value trees, so that the attributes associated with names can be structured values.
Names in the GNS have two parts: . The first part identifies a directory; the second refers to a value tree, or some portion of a value tree. For example, see Figure 13.7, in which the DIs are illustrated as small integers (although they are actually chosen from a range of integers to ensure uniqueness). The attributes of a user Peter.Smith in the directory QMUL would be stored in the value tree named . The value tree includes a password, which can be referenced as , and several mail addresses, each of which would be listed in the value tree as a single node with the name . The directory tree is partitioned and stored in many servers, with each partition replicated in several servers. The consistency of the tree is maintained in the face of two or more concurrent updates – for example, two users may simultaneously attempt to create entries with the same name, and only one should succeed. Replicated directories present a second consistency problem; this

is addressed by an asynchronous update distribution algorithm that ensures eventual consistency, but with no guarantee that all copies are always current.



Accommodating change

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

Figure 13.8 Merging trees under a new root

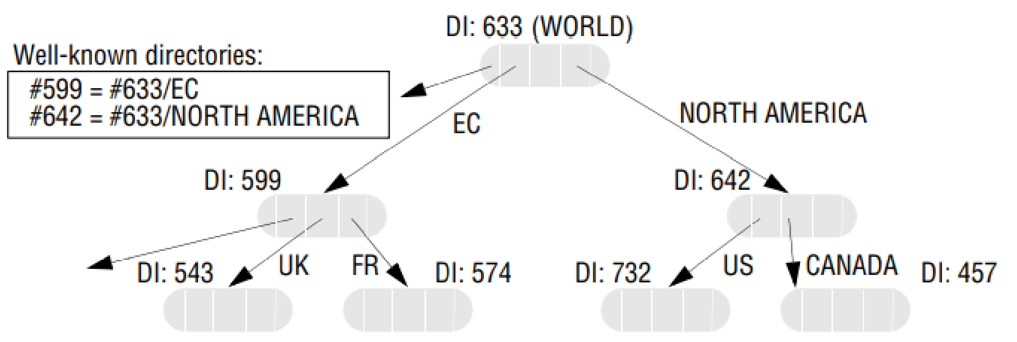
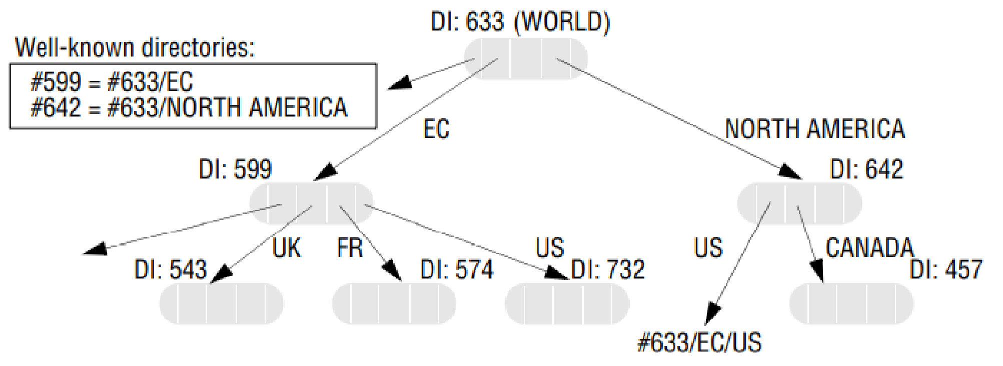


Figure 13.9 Restructuring the directory



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

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

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

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

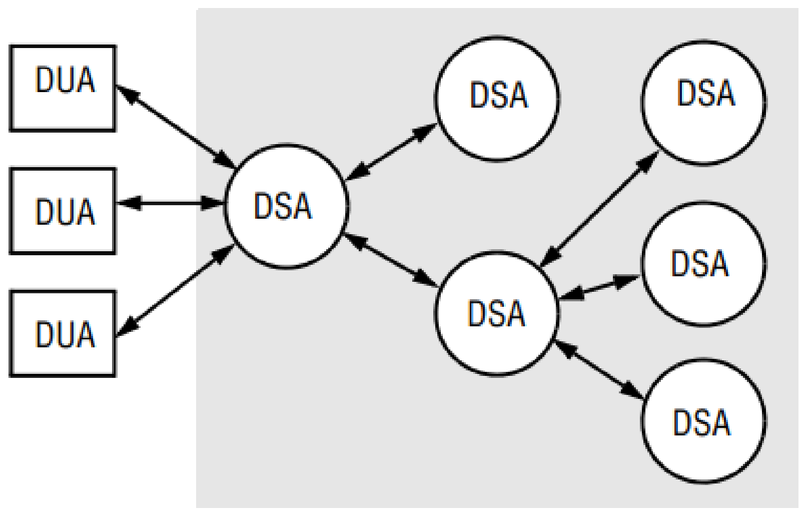
Case study: The X.500 Directory Service

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

telephone numbers of garages specializing in the repair of a particular make of car, to the use of the directory to access personal details such as job roles, dietary habits or even photographic images of the individuals.
Such queries may originate from users, in the ‘yellow pages' example mentioned above, or from processes, when they may be used to identify services to meet a functional requirement.
Individuals and organizations can use a directory service to make available a wide range of information about themselves and the resources that they wish to offer for use in the network. Users can search the directory for specific information with only partial knowledge of its name, structure or content.

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

X.500 service architecture



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

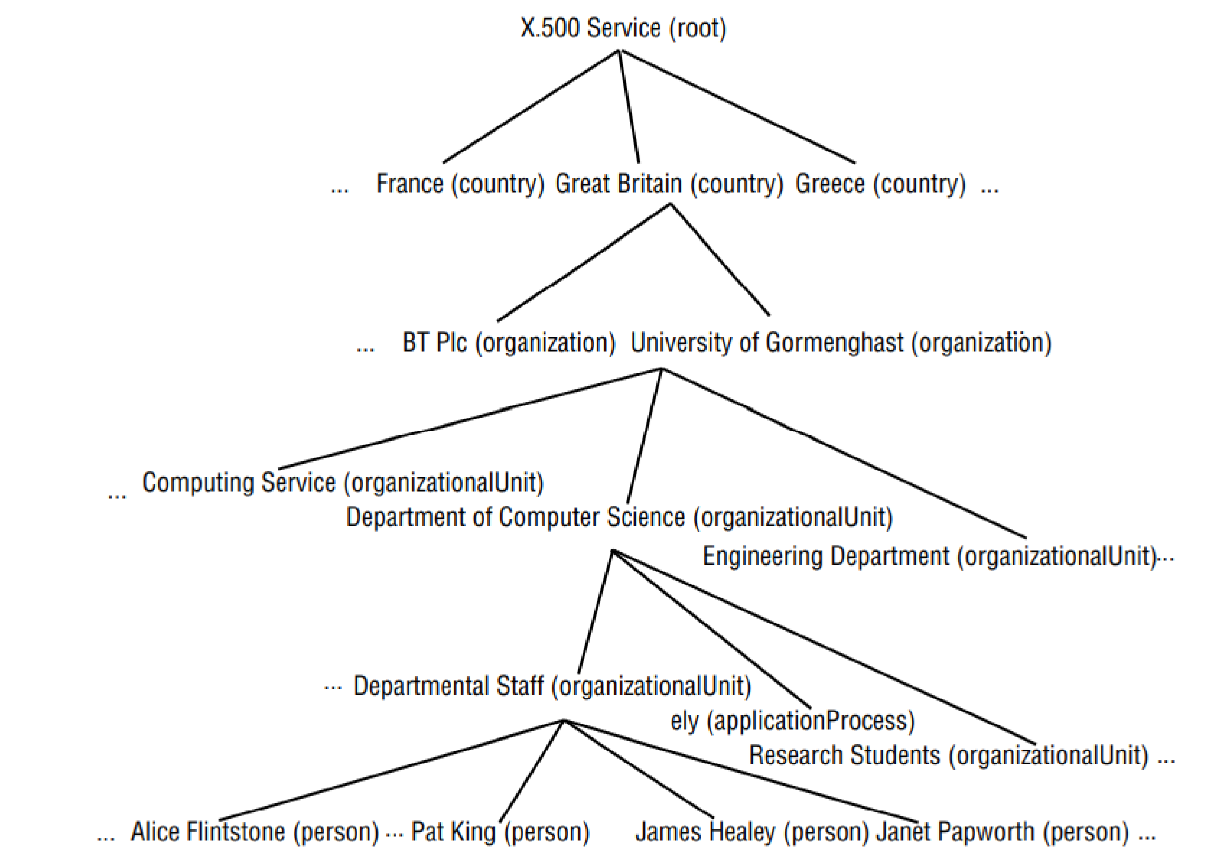
The X.500 name tree is called the Directory Information Tree (DIT), and the entire directory structure including the data associated with the nodes, is called the Directory Information Base (DIB). There is intended to be a single integrated DIB containing information provided by organizations throughout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

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

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

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

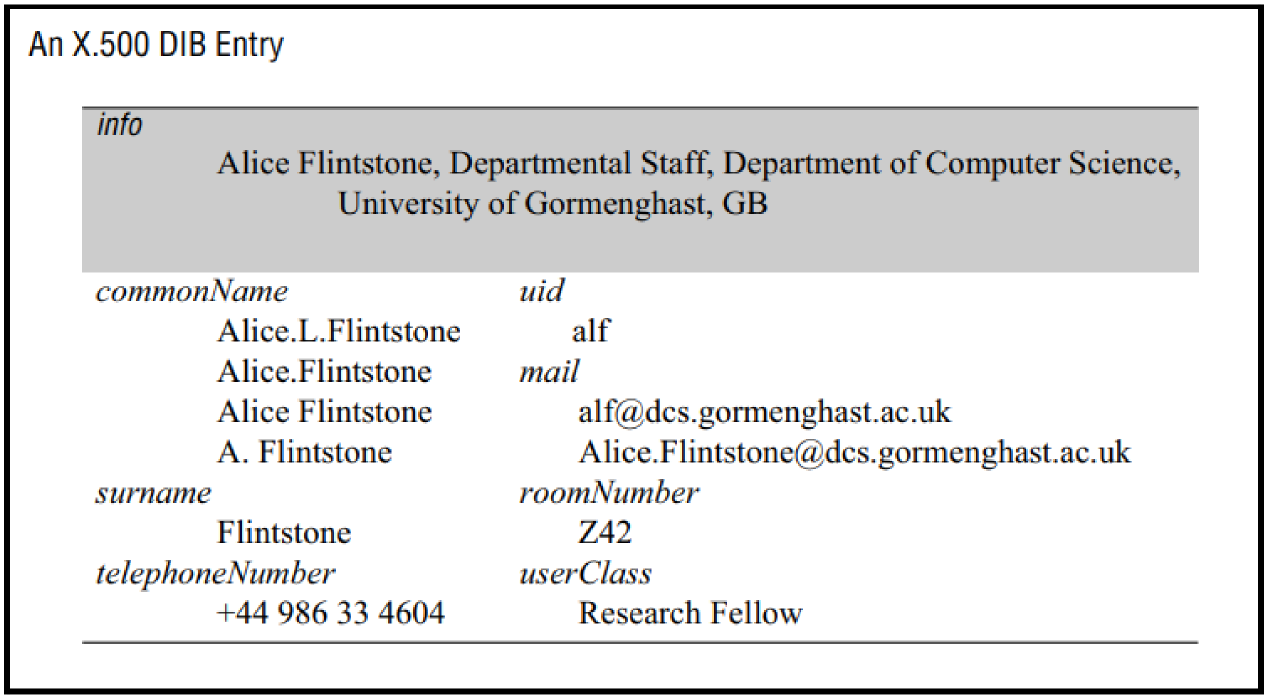
Figure 13.11 Part of the X.500 Directory Information Tree


X.500 Service (root)
... France (country) Great Britain (country) Greece (country) ...
... BT Plc (organization) University of Gormenghast (organization)
... Computing Service (organizationalUnit)
Department of Computer Science (organizationalUnit)
Engineering Department (organizationalUnit) ...
... Departmental Staff (organizationalUnit)
ely (applicationProcess)
Research Students (organizationalUnit) ...
... Alice Flintstone (person) ... Pat King (person) James Healey (person) Janet Papworth (person) ...

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

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

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



An X.500 DIB Entry

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

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

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

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

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

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

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

The DIB is partitioned, with the expectation that each organization will provide at least one server holding the details of the entities in that organization. Portions of the DIB may be replicated in several servers.
As a standard (or a 'recommendation' in CCITT terminology), X.500 does not address implementation issues. However, it is quite clear that any implementation involving multiple servers in a wide area network must rely on extensive use of replication and caching techniques to avoid too much redirection of queries.
One implementation, described by Rose [1992], is a system developed at University College, London, known as QUIPU [Kille 1991]. In this implementation, both caching and replication are performed at the level of individual DIB entries, and at the level of collections of entries descended from the same node. It is assumed that values may become inconsistent after an update, and the time interval in which the consistency is restored may be several minutes. This form of update dissemination is generally considered acceptable for directory service applications.

Lightweight Directory Access Protocol
• X.500's assumption that organizations would provide information about themselves in public directories within a common system has proved largely unfounded. Equally, its complexity has meant that its uptake has been relatively modest.
A group at the University of Michigan proposed a more lightweight approach called the Lightweight Directory Access Protocol (LDAP), in which a DUA accesses X.500 directory services directly over TCP/IP instead of the upper layers of the ISO protocol stack. This is described in RFC 2251 [Wahl et al. 1997]. LDAP also simplifies the interface to X.500 in other ways: for example, it provides a relatively simple API and it replaces ASN.1 encoding with textual encoding.
Although the LDAP specification is based on X.500, LDAP does not require it. An implementation may use any other directory server that obeys the simpler LDAP

specification, as opposed to the X.500 specification. For example, Microsoft's Active Directory Services provides an LDAP interface.
Unlike X.500, LDAP has been widely adopted, particularly for intranet directory services. It provides secure access to directory data through authentication.

Designing Distributed systems :THE GOOGLE CASE STUDY

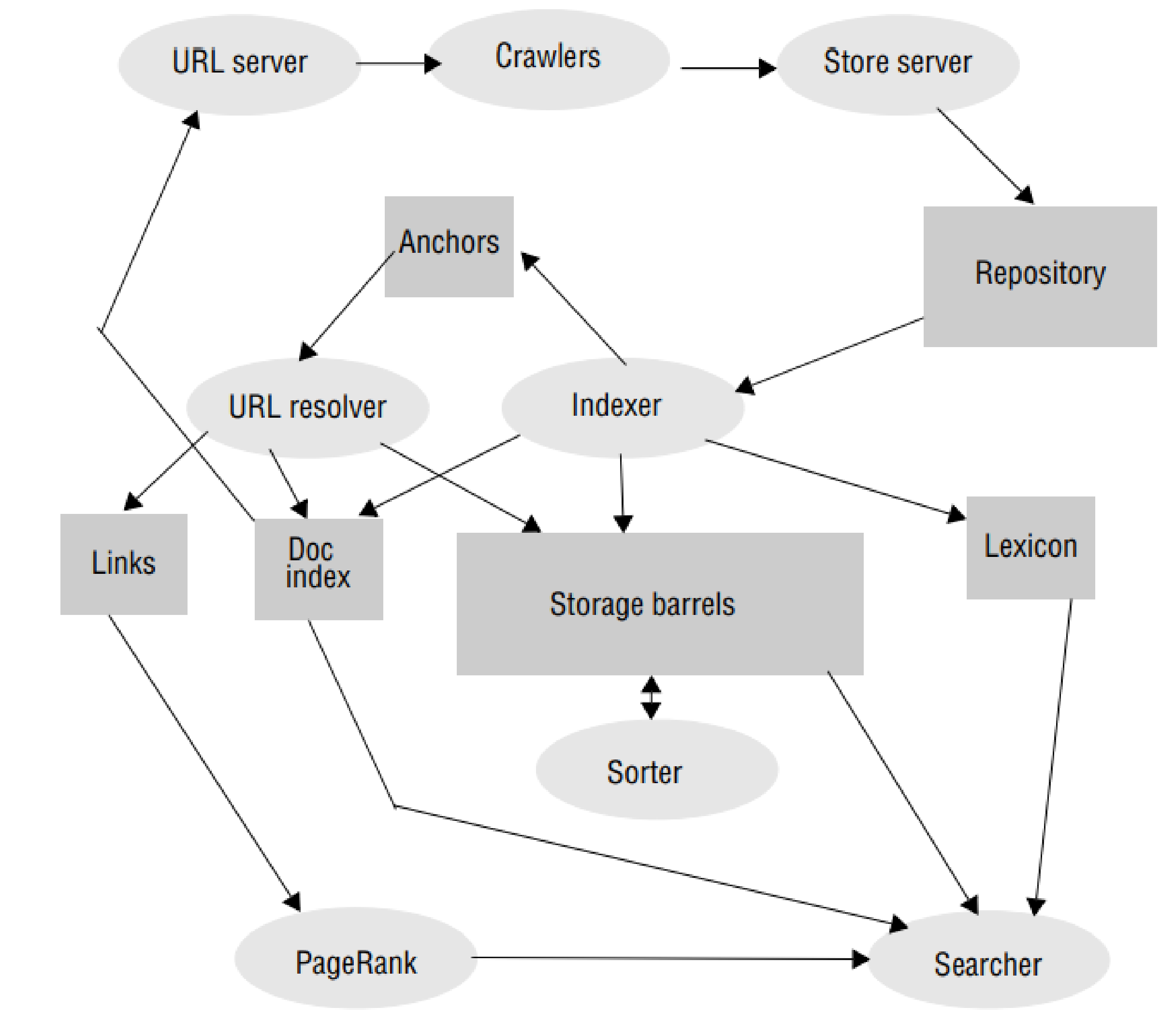
Google [www.google.com III] is a US-based corporation with its headquarters in Mountain View, California (the Googleplex), offering Internet search and broader web applications and earning revenue largely from advertising associated with such services. The name is a play on the word googol, the number 10100 (or 1 followed by a hundred zeros), emphasizing the sheer scale of information available in the Internet today. Google's mission is to tame this huge body of information: 'to organize the world's information and make it universally accessible and useful' [www.google.com III].
Google was born out of a research project at Stanford University, with the company launched in 1998. Since then, it has grown to have a dominant share of the Internet search market, largely due to the effectiveness of the underlying ranking algorithm used in its search engine (discussed further below). Significantly, Google has diversified, and as well as providing a search engine is now a major player in cloud computing. From a distributed systems perspective, Google provides a fascinating case study with extremely demanding requirements, particularly in terms of scalability, reliability, performance and openness (see the discussion of these challenges in Section 1.5). For example, in terms of search, it is noteworthy that the underlying system has successfully scaled with the growth of the company from its initial production system in 1998 to dealing with over 88 billion queries a month by the end of 2010, that the main search engine has never experienced an outage in all that time and that users can expect query results in around 0.2 seconds [googleblog.blogspot.com I].
The case study we present here will examine the strategies and design decisions behind that success, and provide insight into design of complex distributed systems. Before proceeding to the case study, though, it is instructive to look in more detail at the search engine and also at Google as a cloud provider
The Google search engine • The role of the Google search engine is, as for any web search engine, to take a given query and return an ordered list of the most relevant results that match that query by searching the content of the Web. The challenges stem from the size of the Web and its rate of change, as well as the requirement to provide the most relevant results from the perspective of its users. We provide a brief overview of the operation of Google search below; a fuller description of the operation of the Google

search engine can be found in Langville and Meyer [2006]. As a running example, we consider how the search engine responds to the query ‘distributed systems book’.
The underlying search engine consists of a set of services for crawling the Web and indexing and ranking the discovered pages, as discussed below.
Crawling: The task of the crawler is to locate and retrieve the contents of the Web and pass the contents onto the indexing subsystem. This is performed by a software service called Googlebot, which recursively reads a given web page, harvesting all the links from that web page and then scheduling further crawling operations for the harvested links (a technique known as deep searching that is highly effective in reaching practically all pages in the Web). In the past, because of the size of the Web, crawling was generally performed once every few weeks. However, for certain web pages this was insufficient.
For example, it is important for search engines to be able to report accurately on breaking news or changing share prices. Googlebot therefore took note of the change history of web pages and revisited frequently changing pages with a period roughly proportional to how often the pages change. With the introduction of Caffeine in 2010 [googleblog.blogspot.com II], Google has moved from a batch approach to a more continuous process of crawling intended to offer more freshness in terms of search results. Caffeine is built using a new infrastructure service called Percolator that supports the incremental updating of large datasets [Peng and Dabek 2010]. Indexing: While crawling is an important function in terms of being aware of the content of the Web, it does not really help us with our search for occurrences of ‘distributed systems book’. To understand how this is processed, we need to have a closer look at indexing. The role of indexing is to produce an index for the contents of the Web that is similar to an index at the back of a book, but on a much larger scale.
More precisely, indexing produces what is known as an inverted index mapping words appearing in web pages and other textual web resources (including documents in .pdf, .doc and other formats) onto the positions where they occur in documents, including the precise position in the document and other relevant information such as the font size and capitalization (which is used to determine importance, as will be seen below). The index is also sorted to support efficient queries for words against locations. As well as maintaining an index of words, the Google search engine also maintains an index of links, keeping track of which pages link to a given site. This is used by the PageRank algorithm, as discussed below. Let us return to our example query. This inverted index will allow us to discover web pages that include the search terms ‘distributed’, ‘systems’ and ‘book’ and, by careful analysis, we will be able to discover pages that include all of these terms.
For example, the search engine will be able to identify that the three terms can all be foundin amazon.com, www.cdk5.net and indeed many other web sites. Using the index, it

is therefore possible to narrow down the set of candidate web pages from billions to perhaps tens of thousands, depending on the level of discrimination in the keywords chosen.

Ranking: The problem with indexing on its own is that it provides no information about the relative importance of the web pages containing a particular set of keywords – yet this is crucial in determining the potential relevance of a given page. All modern search engines therefore place significant emphasis on a system of ranking whereby a higher rank is an indication of the importance of a page and it is used to ensure that important pages are returned nearer to the top of the list of results than lower-ranked pages. As mentioned above, much of the success of Google can be traced back to the effectiveness of its ranking algorithm, PageRank [Langville and Meyer 2006]. PageRank is inspired by the system of ranking academic papers based on citation analysis. In the academic world, a paper is viewed as important if it has a lot of citations by other academics in the field. Similarly, in PageRank, a page will be viewed as important if it is linked to by a large number of other pages (using the link data mentioned above). PageRank also goes beyond simple 'citation' analysis by looking at the importance of the sites that contain links to a given page. For example, a link from bbc.co.uk will be viewed as more important than a link from Gordon Blair's personal web page. Ranking in Google also takes a number of other factors into account, including the proximity of keywords on a page and whether they are in a large font or are capitalized (based on the information stored in the inverted index). Returning to our example, after performing an index lookup for each of the three words in the query, the search function ranks all the resulting page references according to perceived importance. For example, the ranking will pick out certain page references under amazon.com and www.cdk5.net because of the large number of links to those pages from other 'important sites. The ranking will also prioritize pages where the terms 'distributed', 'systems', and 'book' appear in close proximity. Similarly, the ranking should pull out pages where the words appear near the start of the page or in capitals, perhaps indicating a list of distributed systems textbooks. The end result should be a ranked list of results where the entries at the top are the most important results. Anatomy of a search engine: The founders of Google, Sergey Brin and Larry Page wrote a seminal paper on the 'anatomy' of the Google search engine in 1998 [Brin and Page 1998], providing interesting insights into how their search engine was implemented. The overall architecture described in this paper is illustrated in Figure 21.1, redrawn from the original. In this diagram, we distinguish between services directly supporting web search, drawn as ovals, and the underlying storage infrastructure components, illustrated as rectangles. While it is not the purpose of this chapter to present this architecture in detail, a brief overview will aid comparison with the more sophisticated Google infrastructure available today. The core function of crawling was described above. This takes as input lists of URLs to be fetched, provided by the URL server, with the resultant fetched pages placed into the store server. This data is then compressed and placed in the repository for further analysis, in particular creating the index for searching. The indexing function is performed in two stages. Firstly, the indexer uncompresses the data in the repository.

Outline architecture of the original Google search engine [Brin and Page 1998]



And produces a set of hits, where a hit is represented by the document ID, the word, the position in the document, and other information such as word size and capitalization. This data is then stored in a set of barrels, a key storage element in the initial architecture. This information is sorted by the document ID. The sorter then takes this data and sorts it by word ID to produce the necessary inverted index (as discussed above). The indexer also performs two other crucial functions as it parses the data: it extracts information about links in documents storing this information in an anchors file, and it produces a lexicon for the analyzed data (which at the time the initial architecture was used,consisted of 14 million words). The anchors file is processed by a URL resolver, which performs a number of functions on this data including resolving relative URLs into absolute URLs before producing a links database, as an important input into PageRank calculations. The URL resolver also creates a doc index, which provides input to the URL server in terms of further pages to crawl. Finally, the searcher implements the core Google search capability, taking input from the doc index, PageRank, the inverted index held in the barrels, and also the lexicon.

One thing that is striking about this architecture is that, while specific details of the architecture have changed, the key services supporting web search – that is, crawling, indexing (including sorting), and ranking (through PageRank) – remain the same