CS2056 DISTRIBUTED SYSTEMS

UNIT III

Distributed File Systems-Introduction-File service architecture-Case Study:Sun Network File System-Enhancements and further developments.

Name Services-Introduction-Name Services and the Domain Name System-Directory Services-Case Study: Global Name Service.

Lecture Notes

A file system is responsible for the organization, storage, retrieval, naming, sharing, and protection of files. File systems provide directory services, which convert a file name (possibly a hierarchical one) into an internal identifier (e.g. inode, FAT index). They contain a representation of the file data itself and methods for accessing it (read/write). The file system is responsible for controlling access to the data and for performing low-level operations such as buffering frequently used data and issuing disk I/O requests.

A distributed file system is to present certain degrees of transparency to the user and the system:
Access transparency: Clients are unaware that files are distributed and can access them in the same way as local files are accessed.
Location transparency: A consistent name space exists encompassing local as well as remote files. The name of a file does not give it location.
Concurrency transparency: All clients have the same view of the state of the file system. This means that if one process is modifying a file, any other processes on the same system or remote systems that are accessing the files will see the modifications in a coherent manner.
Failure transparency: The client and client programs should operate correctly after a server failure.
Heterogeneity: File service should be provided across different hardware and operating system platforms.
Scalability: The file system should work well in small environments (1 machine, a dozen machines) and also scale gracefully to huge ones (hundreds through tens of thousands of systems).
Replication transparency: To support scalability, we may wish to replicate files across multiple servers. Clients should be unaware of this.
Migration transparency: Files should be able to move around without the client's knowledge.
Support fine-grained distribution of data: To optimize performance, we may wish to locate individual objects near the processes that use them.
Tolerance for network partitioning: The entire network or certain segments of it may be unavailable to a client during certain periods (e.g. disconnected operation of a laptop). The file system should be tolerant of this.

File service types

To provide a remote system with file service, we will have to select one of two models of operation. One of these is the upload/download model. In this model, there are two fundamental operations: *read file* transfers an entire file from the server to the requesting client, and *write file* copies the file back to the server. It is a simple model and efficient in that it provides local access to the file when it is being used. Three problems are evident. It can be wasteful if the client needs access to only a small amount of the file data. It can be problematic if the client doesn't have enough space to cache the entire file. Finally, what happens if others need to modify the same file?

The second model is a remote access model. The file service provides remote operations such as *open*, *close*, *read bytes*, *write bytes*, *get attributes*, etc. The file system itself runs on servers. The drawback in this approach is the servers are accessed for the duration of file access rather than once to download the file and again to upload it.

Another important distinction in providing file service is that of understanding the difference between *directory service* and *file service*. A directory service, in the context of file systems, maps human-friendly textual names for files to their internal locations, which can be used by the file service. The file service itself provides the file interface (this is mentioned above). Another component of file distributed file systems is the client module. This is the client-side interface for file and directory service. It provides a local file system interface to client software (for example, the vnode file system layer of a UNIX kernel).

Introduction

- File system were originally developed for centralized computer systems and desktop computers.
- File system was as an operating system facility providing a convenient programming interface to disk storage.
- Distributed file systems support the sharing of information in the form of files and hardware resources.
- With the advent of distributed object systems (CORBA, Java) and the web, the picture has become more complex.
- Figure 1 provides an overview of types of storage system.

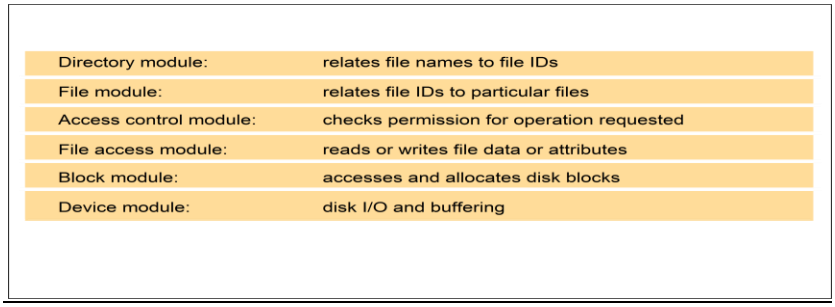
Figure 1. Storage systems and their properties

	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

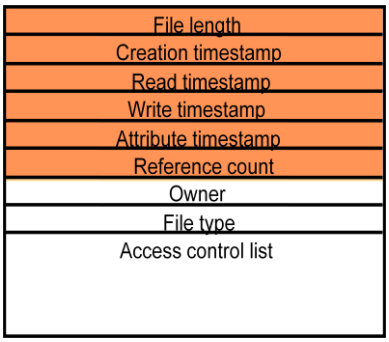
- Figure 2 shows a typical layered module structure for the implementation of a non-distributed file system in a conventional operating system.

Figure 2. File system modules



- File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files.
- Files contain both data and attributes.
- A typical attribute record structure is illustrated in Figure 3.

Figure 3. File attribute record structure



- Figure 4 summarizes the main operations on files that are available to applications in UNIX systems.

Figure 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)	Transfers *n* bytes from the file referenced by *filedes* to *buffer*.
count = write(filedes, buffer, n)	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)	Gets the file attributes for file *name* into *buffer*.

- Distributed File system requirements
  - Related requirements in distributed file systems are:
    - Transparency
    - Concurrency
    - Replication
    - Heterogeneity
    - Fault tolerance
    - Consistency
    - Security
    - Efficiency

Case studies

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. 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.

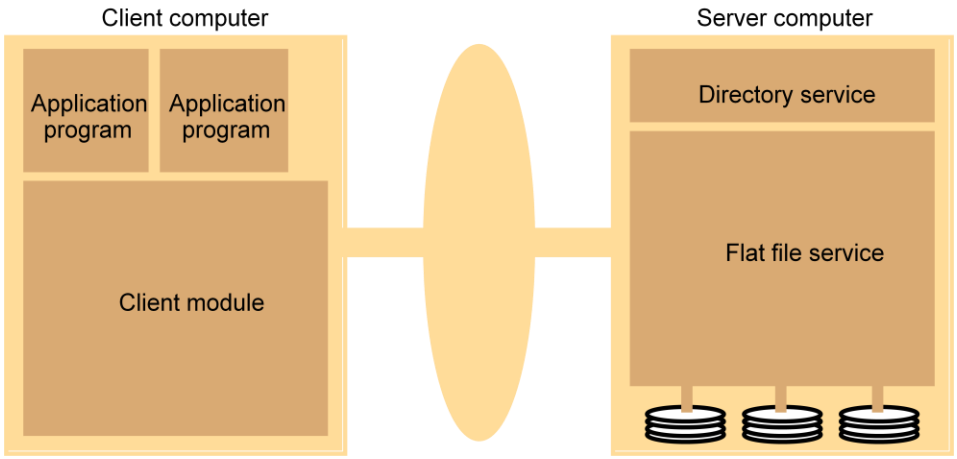
Andrew File System • Andrew is a distributed computing environment developed at Carnegie Mellon University (CMU) for use as a campus computing and information system. The design of the Andrew File System (henceforth abbreviated AFS) reflects an intention to support information sharing on a large scale 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.

Lecture Notes

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
  - A client module.
- The relevant modules and their relationship is shown in Figure 5.

Figure 5. File service architecture



- The Client module implements exported interfaces by flat file and directory services on server side.
- Responsibilities of various modules can be defined as follows:
  - Flat file service:
    - Concerned with the implementation of operations on the contents of file. Unique File Identifiers (UFIDs) are used to refer to files in all requests for

flat file service operations. UFIDs are long sequences of bits chosen so that each file has a unique among all of the files in a distributed system.

- Directory service:
  - Provides mapping between text names for the files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to directory service. Directory service supports functions needed generate directories, to add new files to directories.
- Client module:
  - It runs on each computer and provides integrated service (flat file and directory) as a single API to application programs. For example, in UNIX hosts, a client module emulates the full set of Unix file operations.
  - It holds information about the network locations of flat-file and directory server processes; and achieve better performance through implementation of a cache of recently used file blocks at the client.
- Flat file service interface:
  - Figure 6 contains a definition of the interface to a flat file service.

Figure 6. Flat file service operations

Read(FileId, i, n) -> Data -throws BadPosition	if *1≤i≤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≤i≤Length(File)+1*: Write a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() -> FileId	Creates a new file of length0 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 3.)

- Access control
  - 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.
- Directory service interface
  - Figure 7 contains a definition of the RPC interface to a directory service.

Figure 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, File) -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)	If *Name* is in the directory, the entry containing *Name* is removed 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*.

- 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.
- File Group
  - A file group is a collection of files that can be located on any server or moved between servers while maintaining the same names.
    - A similar construct is used in a UNIX file system.
    - It helps with distributing the load of file serving between several servers.
    - File groups have identifiers which are unique throughout the system (and hence for an open system, they must be globally unique).

To construct globally unique ID we use some unique attribute of the machine on which it is created. E.g: IP number, even though the file group may move subsequently.

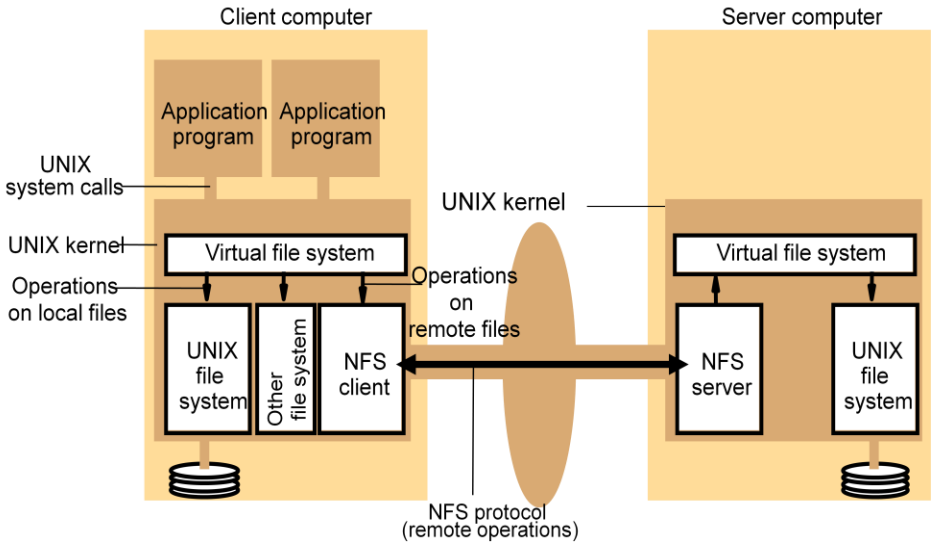
	32 bits	*16 bits*
file group identifier:	IP address	date

DFS: Case Studies

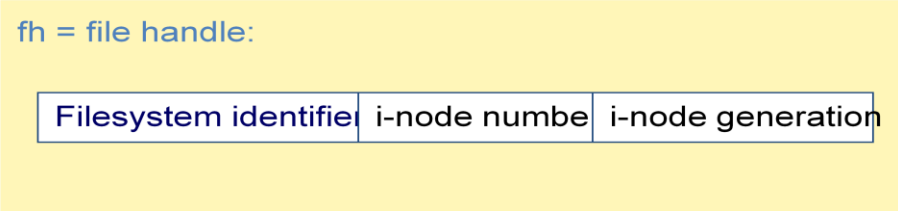
- NFS (Network File System)
  - Developed by Sun Microsystems (in 1985)
  - Most popular, open, and widely used.
  - NFS protocol standardized through IETF (RFC 1813)
- AFS (Andrew File System)
  - Developed by Carnegie Mellon University as part of Andrew distributed computing environments (in 1986)
  - A research project to create campus wide file system.
  - Public domain implementation is available on Linux (LinuxAFS)
  - It was adopted as a basis for the DCE/DFS file system in the Open Software Foundation (OSF, www.opengroup.org) DEC (Distributed Computing Environment

NFS architecture

Figure 8 shows the architecture of Sun NFS



- The file identifiers used in NFS are called file handles.



- A simplified representation of the RPC interface provided by NFS version 3 servers is shown in Figure 9.

Figure 9. NFS server operations (NFS Version 3 protocol, simplified)

- *read(fh, offset, count) -> attr, data*
- *write(fh, offset, count, data) -> attr*
- *create(dirfh, name, attr) -> newfh, attr*
- *remove(dirfh, name) status*
- *getattr(fh) -> attr*
- *setattr(fh, attr) -> attr*
- *lookup(dirfh, name) -> fh, attr*
- *rename(dirfh, name, todirfh, toname)*
- *link(newdirfh, newname, dirfh, name)*
- *readdir(dirfh, cookie, count) -> entries*
- *symlink(newdirfh, newname, string) -> status*
- *readlink(fh) -> string*
- *mkdir(dirfh, name, attr) -> newfh, attr*
- *rmdir(dirfh, name) -> status*
- *statfs(fh) -> fsstats*

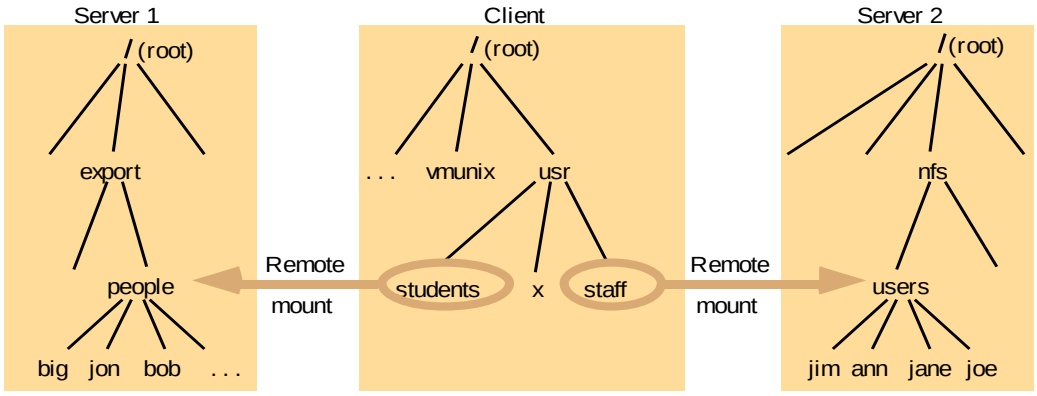
- NFS access control and authentication
  - The NFS server is stateless server, so the user's identity and access rights must be checked by the server on each request.
    - In the local file system they are checked only on the file's access permission attribute.
  - Every client request is accompanied by the userID and groupID
    - It is not shown in the Figure 8.9 because they are inserted by the RPC system.
  - Kerberos has been integrated with NFS to provide a stronger and more comprehensive security solution.

- Mount service
  - Mount operation:

    mount(remotehost, remotedirectory, localdirectory)

- Server maintains a table of clients who have mounted filesystems at that server.
- Each client maintains a table of mounted file systems holding:
  < IP address, port number, file handle>
- Remote file systems may be hard-mounted or soft-mounted in a client computer.
- Figure 10 illustrates a Client with two remotely mounted file stores.

Figure 10. Local and remote file systems accessible on an NFS client



- 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.
    - Automounter has a table of mount points with a reference to one or more NFS servers listed against each.
    - it sends a probe message to each candidate server and then uses the mount service to mount the file system at the first server to respond.
  - Automounter keeps the mount table small.
  - Automounter Provides a simple form of replication for read-only file systems.
    - E.g. if there are several servers with identical copies of /usr/lib then each server will have a chance of being mounted at some clients.
- Server caching
  - Similar to UNIX file caching for local files:
    - pages (blocks) from disk are held in a main memory buffer cache until the space is required for newer pages. Read-ahead and delayed-write optimizations.
    - For local files, writes are deferred to next sync event (30 second intervals).
    - Works well in local context, where files are always accessed through the local cache, but in the remote case it doesn't offer necessary synchronization guarantees to clients.

- NFS v3 servers offers two strategies for updating the disk:
  - Write-through - altered pages are written to disk as soon as they are received at the server. When a write() RPC returns, the NFS client knows that the page is on the disk.
  - Delayed commit - pages are held only in the cache until a commit() call is received for the relevant file. This is the default mode used by NFS v3 clients. A commit() is issued by the client whenever a file is closed.

- Client caching
  - Server caching does nothing to reduce RPC traffic between client and server
    - further optimization is essential to reduce server load in large networks.
    - NFS client module caches the results of read, write, getattr, lookup and readdir operations
    - synchronization of file contents (one-copy semantics) is not guaranteed when two or more clients are sharing the same file.
  - Timestamp-based validity check
    - It reduces inconsistency, but doesn't eliminate it.
    - It is used for validity condition for cache entries at the client:

$$(T - Tc < t) \; v \; (Tmclient = Tmserver)$$

t	freshness guarantee
Tc	time when cache entry was last validated
Tm	time when block was last updated at server
T	current time

    - it is configurable (per file) but is typically set to 3 seconds for files and 30 secs. for directories.
    - it remains difficult to write distributed applications that share files with NFS.

- Other NFS optimizations
  - Sun RPC runs over UDP by default (can use TCP if required).
  - Uses UNIX BSD Fast File System with 8-kbyte blocks.
  - reads() and writes() can be of any size (negotiated between client and server).
  - The guaranteed freshness interval t is set adaptively for individual files to reduce getattr() calls needed to update Tm.
  - File attribute information (including Tm) is piggybacked in replies to all file requests.
- NFS performance
  - Early measurements (1987) established that:
    - Write() operations are responsible for only 5% of server calls in typical UNIX environments.
      - hence write-through at server is acceptable.

- Lookup() accounts for 50% of operations -due to step-by-step pathname resolution necessitated by the naming and mounting semantics.
- More recent measurements (1993) show high performance.
  - see www.spec.org for more recent measurements.

- NFS summary
  - NFS is an excellent example of a simple, robust, high-performance distributed service.
  - Achievement of transparencies are other goals of NFS:
    - Access transparency:
      - The API is the UNIX system call interface for both local and remote files.
    - Location transparency:
      - Naming of filesystems is controlled by client mount operations, but transparency can be ensured by an appropriate system configuration.
    - Mobility transparency:
      - Hardly achieved; relocation of files is not possible, relocation of filesystems is possible, but requires updates to client configurations.
    - Scalability transparency:
      - File systems (file groups) may be subdivided and allocated to separate servers.
    - Replication transparency:
      - Limited to read-only file systems; for writable files, the SUN Network Information Service (NIS) runs over NFS and is used to replicate essential system files.
    - Hardware and software operating system heterogeneity:
      - NFS has been implemented for almost every known operating system and hardware platform and is supported by a variety of filling systems.
    - Fault tolerance:
      - Limited but effective; service is suspended if a server fails. Recovery from failures is aided by the simple stateless design.
    - Consistency:
      - It provides a close approximation to one-copy semantics and meets the needs of the vast majority of applications.
      - But the use of file sharing via NFS for communication or close coordination between processes on different computers cannot be recommended.
    - Security:
      - Recent developments include the option to use a secure RPC implementation for authentication and the privacy and security of the data transmitted with read and write operations.
      - Efficiency:
        - NFS protocols can be implemented for use in situations that generate very heavy loads.

Case Study: The Andrew File System (AFS)

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.

- Like NFS, AFS provides transparent access to remote shared files for UNIX programs running on workstations.
- AFS is implemented as two software components that exist at UNIX processes called Vice and Venus.

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

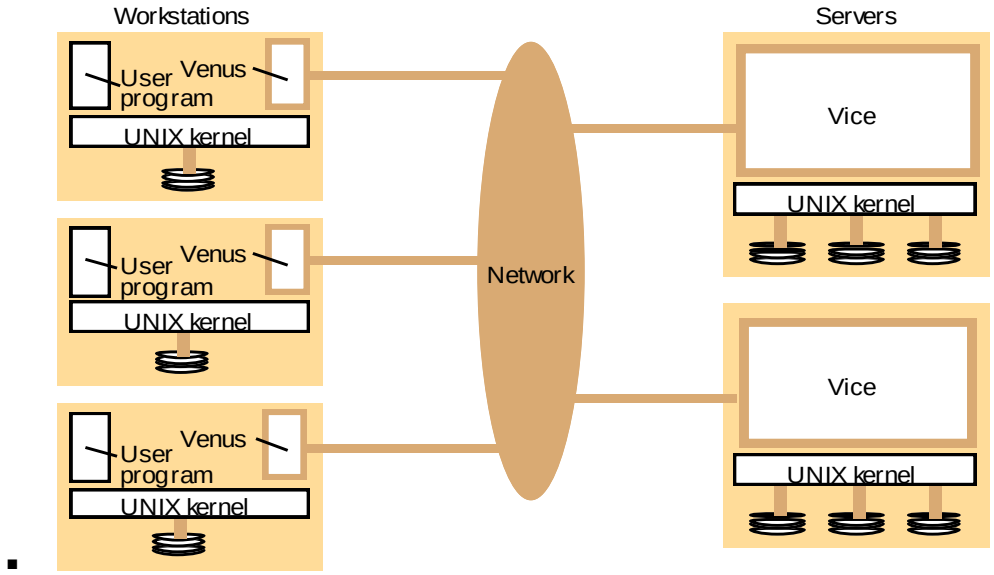
1. When a user process in a client computer issues an *open* system call for a file in the shared file space and there is not a current copy of the file in the local cache, the server holding the file is located and is sent a request for a copy of the file.

2. The copy is stored in the local UNIX file system in the client computer. The copy is then *open*ed 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.

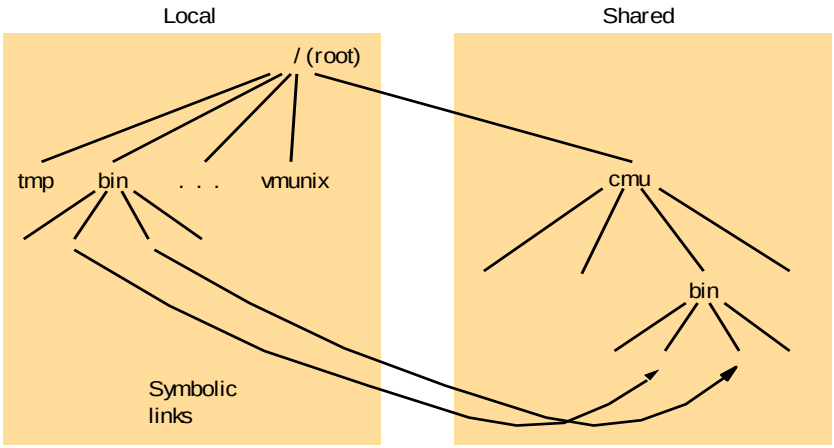
4. When the process in the client issues a *close* system call, if the local copy has been updated its contents are sent back to the server. The server updates the file contents and the timestamps on the file. The copy on the client's local disk is retained in case it is needed again by a user-level process on the same workstation.

Figure 11. Distribution of processes in the Andrew File System



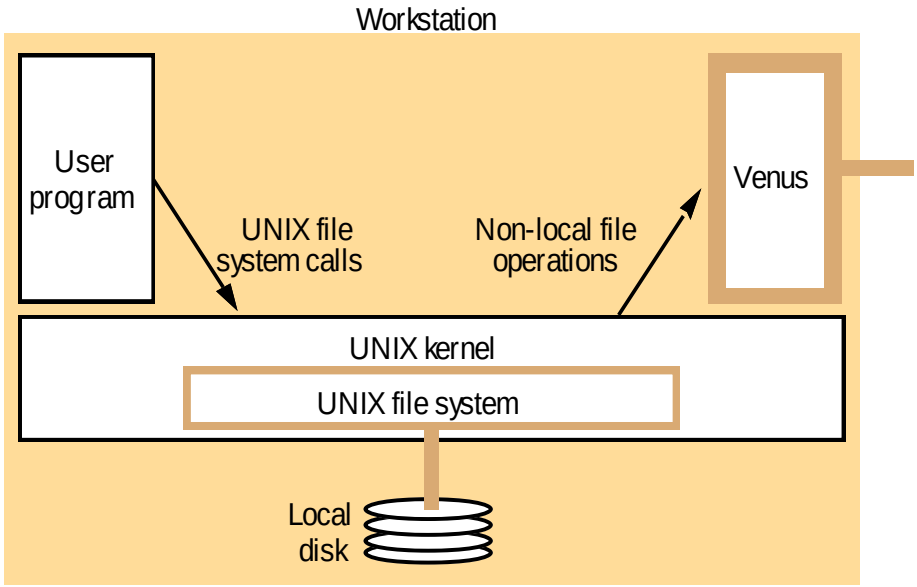
- The files available to user processes running on workstations are either local or shared.
- Local files are handled as normal UNIX files.
- They are stored on the workstation's disk and are available only to local user processes.
- Shared files are stored on servers, and copies of them are cached on the local disks of workstations.
- The name space seen by user processes is illustrated in Figure 12.

Figure 12. File name space seen by clients of AFS



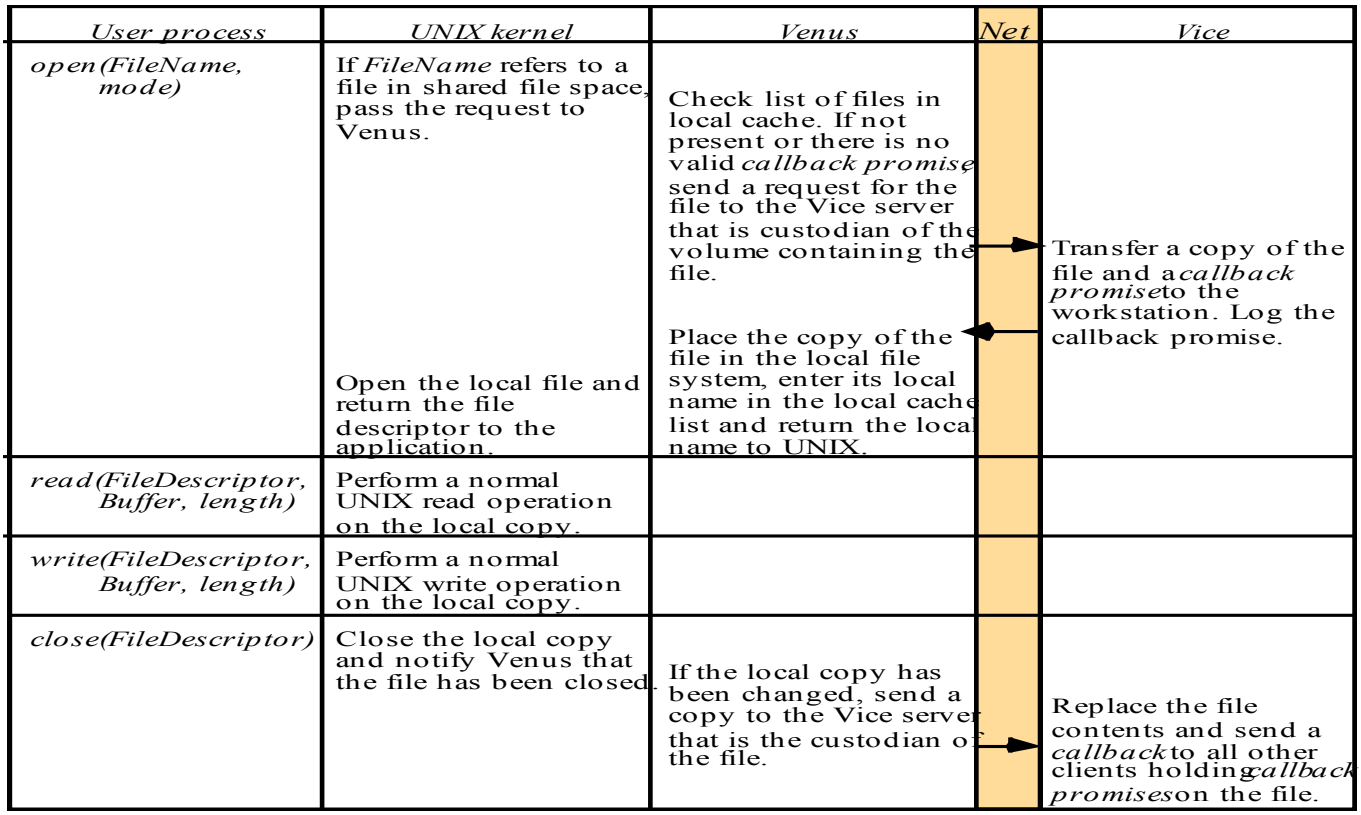
- The UNIX kernel in each workstation and server is a modified version of BSD UNIX.
- The modifications are designed to intercept open, close and some other file system calls when they refer to files in the shared name space and pass them to the Venus process in the client computer. (Figure 13)

Figure 13. System call interception in AFS



- Figure 14 describes the actions taken by Vice, Venus and the UNIX kernel when a user process issues system calls.

Figure 14. implementation of file system calls in AFS



- Figure 15 shows the RPC calls provided by AFS servers for operations on files.

Figure 15. The main components of the Vice service interface

Fetch(fid) -> attr, data	Returns the attributes (status) and, optionally, the contents of file identified by the *fid* and records a callback promise on it.
Store(fid, attr, data)	Updates the attributes and (optionally) the contents of a specified file.
Create() -> fid	Creates a new file and records a callback promise on it.
Remove(fid)	Deletes the specified file.
SetLock(fid, mode)	Sets a lock on the specified file or directory. The mode of the lock may be shared or exclusive. Locks that are not removed expire after 30 minutes.
ReleaseLock(fid)	Unlocks the specified file or directory.
RemoveCallback(fid)	Informs server that a Venus process has flushed a file from its cache.
BreakCallback(fid)	This call is made by a Vice server to a Venus process. It cancels the callback promise on the relevant file.

Other aspects

AFS introduces several other interesting design developments and refinements that we outline here, together with a summary of performance evaluation results:

1. UNIX kernel modifications
2. Location database
3. Threads
4. Read-only replicas
5. Bulk transfers
6. Partial file caching
7. Performance
8. Wide area support

Naming Services

Which one is easy for humans and machines? and why?

- 74.125.237.83 or **google.com**
- 128.250.1.22 or distributed systems website
- 128.250.1.25 or Prof. Buyya
- Disk 4, Sector 2, block 5 OR /usr/raj/hello.c

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 users.
- Naming is fundamental issue in DS design as it facilitates communication and resource sharing.
  - A name in the form of 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 within one another via a DS unless they can name one another, with email address.
- Names are not the only useful means of identification: descriptive attributes are another.

What are Naming Services?

- How do Naming Services facilitate communication and resource sharing?
  - An URL facilitates the localization of a resource exposed on the Web.
    - *e.g., abc.net.au means it is likely to be an Australian entity?*
  - A consistent and uniform naming helps processes in a distributed system to interoperate and manage resources.
    - *e.g., commercials use .com; non-profit organizations use .org*
  - Users refers to each other by means of their names (i.e. email) rather than their system ids
  - Naming Services are not only useful to locate resources but also to gather additional information about them such as attributes

What are Naming Services?

In a Distributed System, a Naming Service is a specific service whose aim is to provide a consistent and uniform naming of resources, thus allowing other programs or services to localize them and obtain the required metadata for interacting with them.

Key benefits

- Resource localization
- Uniform naming
- Device independent address (e.g., you can move domain name/web site from one server to another server seamlessly).

The role of names and name services

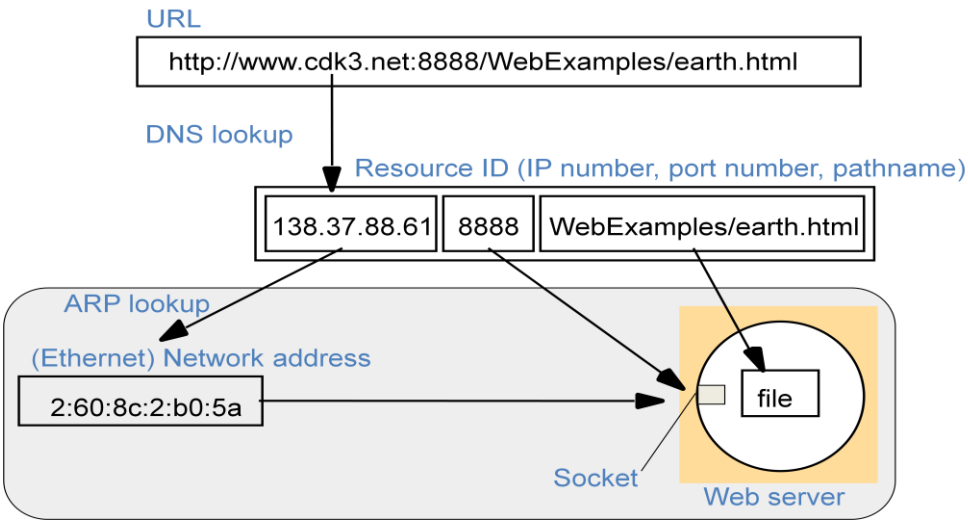
- Resources are accessed using *identifier* or *reference*
  - An identifier can be stored in variables and retrieved from tables quickly

  - Identifier includes or can be transformed to an address for an object
    - *E.g. NFS file handle, Corba remote object reference*
  - A *name* is human-readable value (usually a string) that can be *resolved* to an identifier or address
    - *Internet domain name, file pathname, process number*
    - *E.g ./etc/passwd, http://www.cdk3.net/*
- For many purposes, names are preferable to identifiers
  - because the binding of the named resource to a physical location is deferred and can be changed
  - because they are more meaningful to users
- Resource names are *resolved* by name services
  - to give identifiers and other useful attributes

Requirements for name spaces

- Allow simple but meaningful names to be used
- Potentially infinite number of names
- Structured
  - to allow similar subnames without clashes
  - to group related names
- Allow re-structuring of name trees
  - for some types of change, old programs should continue to work
- Management of trust

Composed naming domains used to access a resource from a URL



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 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.

Name Services and the Domain Name System

- A name service stores a collection of one or more naming contexts, sets of bindings between textual names and attributes for objects such as computers, services, and users.
- The major operation that a name service supports is to resolve names.

Uniform Resource Identifiers

Uniform Resource Identifiers (URIs) 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.

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.

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

Navigation

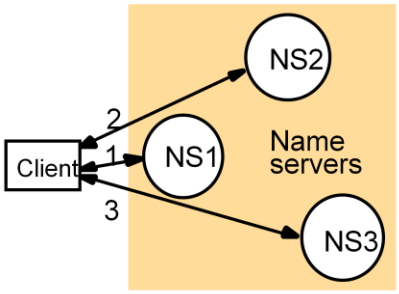
Navigation is the act of chaining multiple Naming Services in order to resolve a single name to the corresponding resource.

- Namespaces allows for structure in names.
- URLs provide a default structure that decompose the location of a resource in
  - protocol used for retrieval
  - internet end point of the service exposing the resource
  - service specific path
- This decomposition facilitates the resolution of the name into the corresponding resource
- Moreover, structured namespaces allows for iterative navigation…

Iterative navigation

Reason for NFS iterative name resolution

This is because the file service may encounter a symbolic link (i.e. an *alias*) 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



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

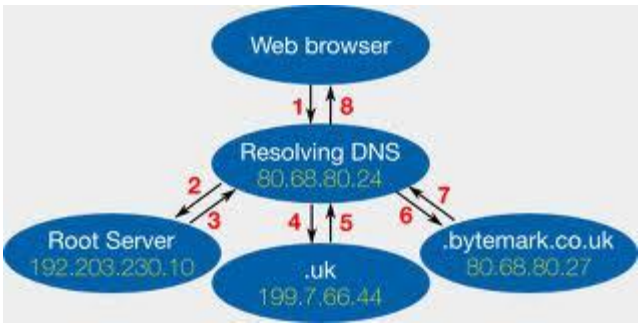
Server controlled navigation

- In an alternative model, name server coordinates naming resolution and returns the results to the client. It can be:
  - Recursive:
    - *it is performed by the naming server*
    - *the server becomes like a client for the next server*
    - *this is necessary in case of client connectivity constraints*
  - Non recursive:
    - *it is performed by the client or the first server*
    - *the server bounces back the next hop to its client*

Non-recursive and recursive server-controlled navigation

DNS offers recursive navigation as an option, but iterative is the standard technique. Recursive navigation must be used in domains that limit client access to their DNS information for security reasons.

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



7. **Lecture Notes: (To be attached)**

8. **Textbook :**

1. George Coulouris, Jean Dollimore, Tim Kindberg, , "Distributed Systems: Concepts and Design", 4th Edition, Pearson Education, 2005. **PP. 350-356.**

9. **Application**

Lecture Notes

The Domain Name System is a name service design whose main naming database is used across the Internet.

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.

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
... – ...

DNS - The Internet Domain Name System

- A distributed naming database (specified in RFC 1034/1305)
- Name structure reflects administrative structure of the Internet
- Rapidly resolves domain names to IP addresses
  - exploits caching heavily
  - typical query time ~100 milliseconds
- Scales to millions of computers
  - partitioned database
  - caching
- Resilient to failure of a server
  - Replication

Basic DNS algorithm for name resolution (domain name -> IP number)

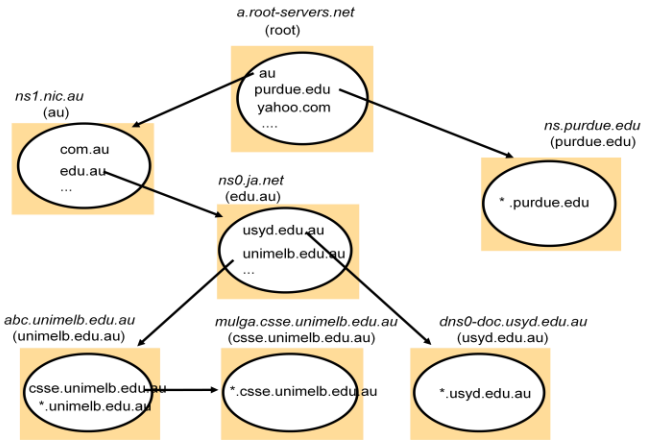
- Look for the name in the local cache
- Try a superior DNS server, which responds with:

– another recommended DNS server

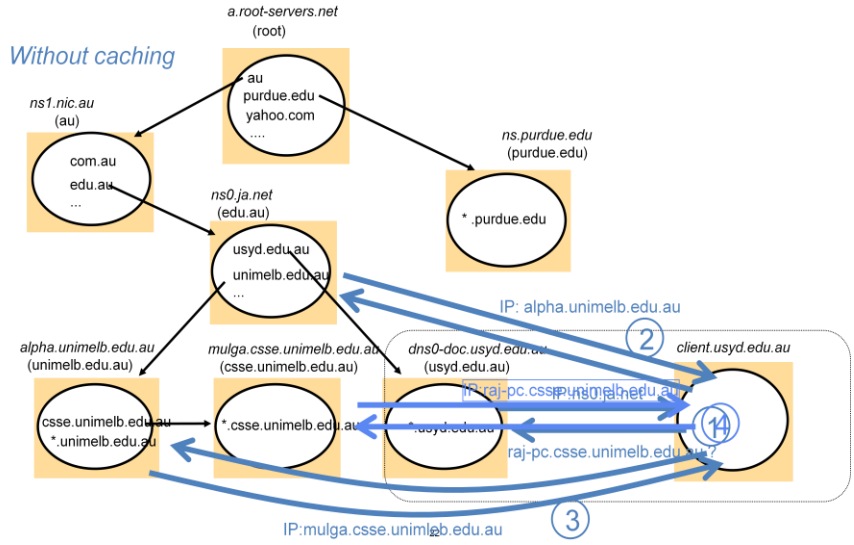
– the IP address (which may not be entirely up to date)

DNS name servers: Hierarchical organisation

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



DNS in typical operation



DNS server functions and configuration

- Main function is to resolve domain names for computers, i.e. to get their IP addresses
  - caches the results of previous searches until they pass their 'time to live'
- Other functions:
  - get *mail host* for a domain
  - reverse resolution - get domain name from IP address
  - Host information - type of hardware and OS
  - Well-known services - a list of well-known services offered by a host
  - Other attributes can be included (optional)

DNS resource records

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.

Record type	*Meaning*	*Main contents*
A	A computer address	IP 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
WKS	A well-known service description	List of service names and protocols
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 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 the following figure where the time to live *1D* means 1 day.

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

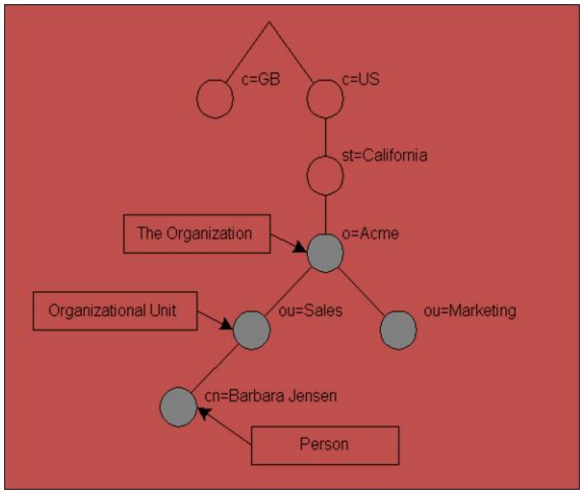
If the domain has any subdomains, there will be further records of type *NS* specifying their name servers, which will also have individual *A* entries. For example, at one point the database for *qmul.ac.uk* contained the following records for the name servers in its subdomain *dcs.qmul.ac.uk*:

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

DNS issues

- Name tables change infrequently, but when they do, caching can result in the delivery of stale data.
  - Clients are responsible for detecting this and recovering
- Its design makes changes to the structure of the name space difficult. For example:
  - merging previously separate domain trees under a new root
  - moving subtrees to a different part of the structure (e.g. if Scotland became a separate country, its domains should all be moved to a new country-level domain.)

- Directory service: 'yellow pages' for the resources in a network
  Retrieves the set of names that satisfy a given description
  - e.g. X.500, LDAP, MS Active Directory Services
    - *(DNS holds some descriptive data, but:*
      - the data is very incomplete
      - DNS isn't organised to search it)
- Discovery service:- a directory service that also:
  is automatically updated as the network configuration changes
  meets the needs of clients in spontaneous networks (Section 2.2.3)
  discovers services required by a client (who may be mobile) within the current *scope*, for example, to find the most suitable printing service for image files after arriving at a hotel.
  Examples of discovery services: Jini discovery service, the 'service location protocol', the 'simple service discovery protocol' (part of UPnP), the 'secure discovery service'.



The 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, Univers and Profile.

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 5559980' 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.

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.

Discovery service

- A database of services with lookup based on service description or type, location and other criteria, E.g.
    1. Find a printing service in this hotel compatible with a Nikon camera
    2. Send the video from my camera to the digital TV in my room.
- Automatic registration of new services
- Automatic connection of guest's clients to the discovery service

Global Name Service (GNS)

- Designed and implemented by Lampson and colleagues at the DEC Systems Research Center (1986)
- Provide facilities for resource location, email addressing and authentication
- When the naming database grows from small to large scale, the structure of name space may change
  → the service should accommodate it
- Cache consistency ?

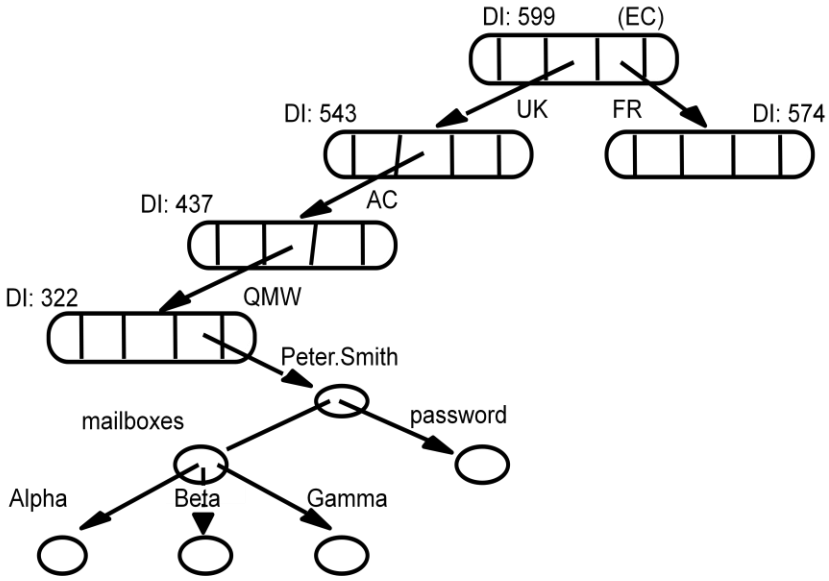
The GNS manages a naming database that is composed of a tree of directories holding names and values. Directories are named by multi-part pathnames referred to a root, or relative to a working directory, much like file names in a UNIX file system. Each directory is also assigned an integer, which serves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

Names in the GNS have two parts: <*directory name*, *value name*>. The first part identifies a directory; the second refers to a value tree, or some portion of a value tree.

GNS Structure

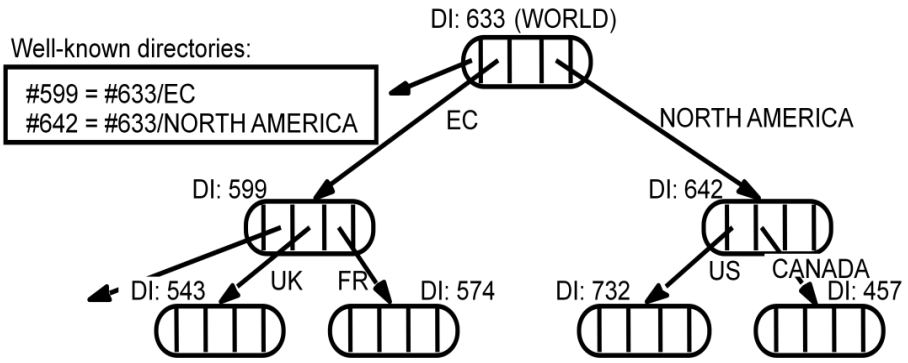
- Tree of directories holding names and values
- Muti-part pathnames refer to the root or relative working directory (like Unix file system)
- Unique Directory Identifier (DI)
- A directory contains list of names and references
- Leaves of tree contain value trees (structured values)

GNS directory tree and value tree



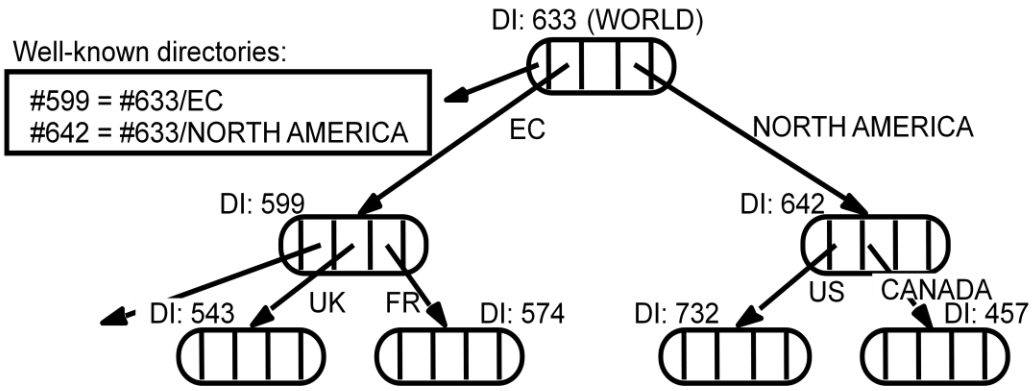
Accommodating changes

- How to integrate naming trees of two previously separate GNS services
- But what is for '/UK/AC/QMV, Peter.Smith' ?



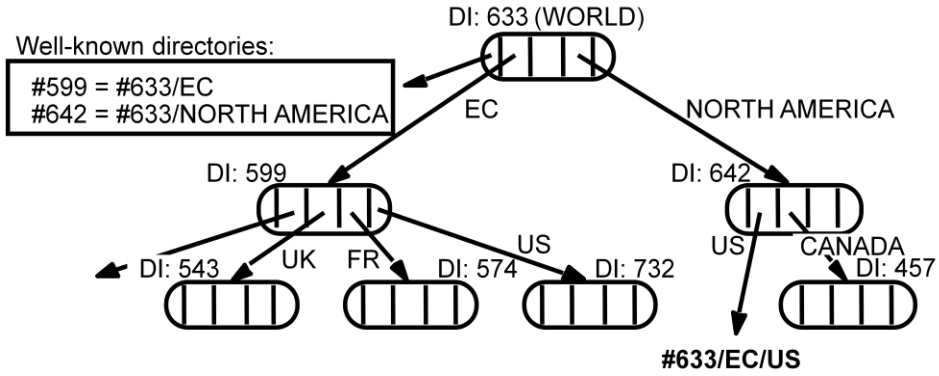
Using DI to solve changes

- Using the name '#599/UK/AC/QMV, Peter.Smith'



Restructuring of database

- Using symbolic links



X500 Directory Service

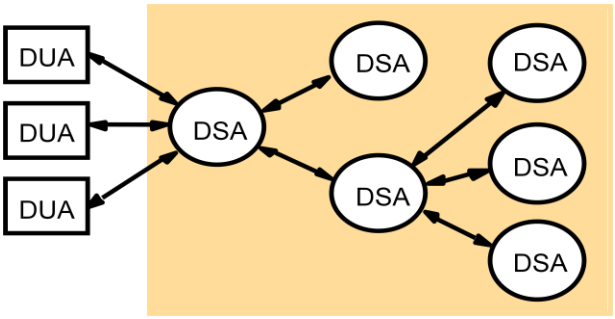
X.500 is a directory service used in the same way as a conventional name service, but it is primarily used to satisfy descriptive queries and is designed to discover the names and attributes of other users or system resources. Users may have a variety of requirements for searching and browsing in a directory 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.

- Standard of ITU and ISO organizations
- Organized in a tree structure with name nodes as in the case of other name servers
- A wide range of attributes are stored in each node
- Directory Information Tree (DIT)
- Directory Information Base (DIB)

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.

- **Directory Server Agent (DSA)**
- **Directory User Agent (DUA)**



In the terminology of the X.500 standard, servers are *Directory Service Agents* (DSAs), and their clients are termed *Directory User Agents* (DUAs). 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.

An X.500 DIB Entry

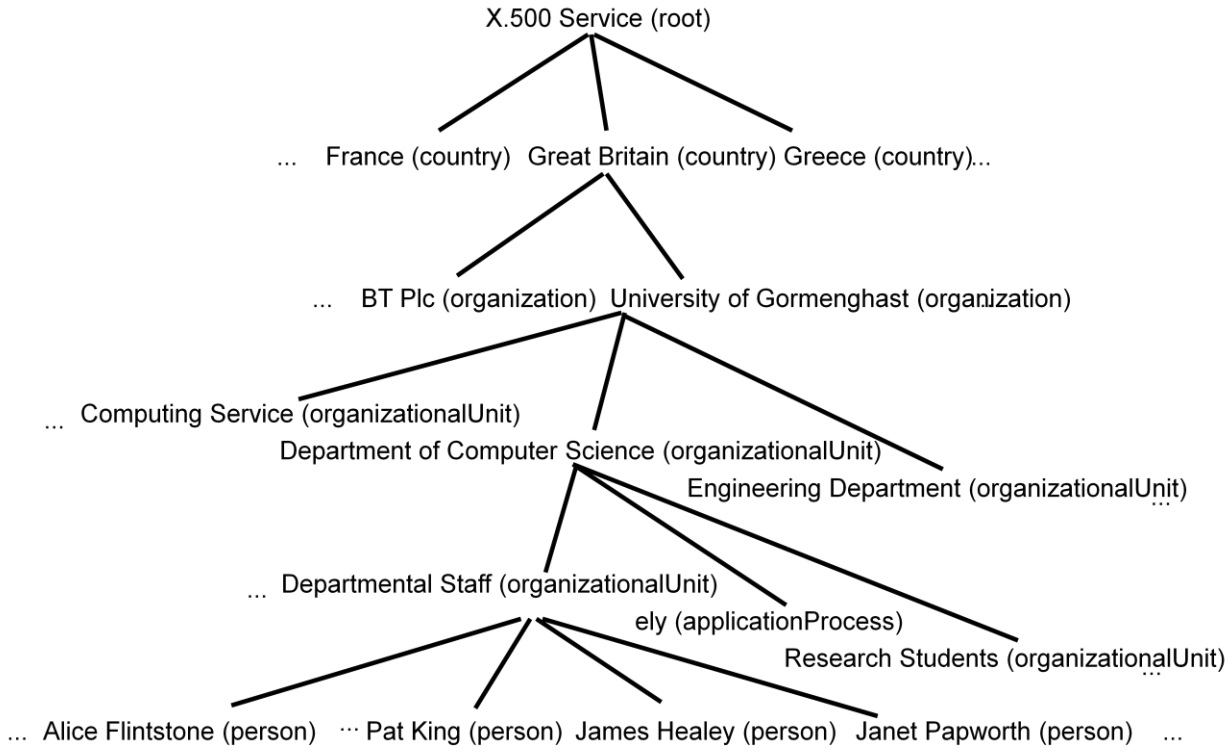
info Alice Flintstone, Departmental Staff, Department of Computer Science, University of Gormenghast, GB	
commonName Alice.L.Flintstone Alice.Flintstone Alice Flintstone A. Flintstone	*uid* alf *mail* alf@dcs.gormenghast.ac.uk Alice.Flintstone@dcs.gormenghast.ac.uk
surname Flintstone	*roomNumber* Z42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

Part of the X.500 Directory Information Tree

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 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.



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

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. 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.