KTU LECTURE NOTES

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

DOWNLOAD KTU ASSIST APP FROM PLAYSTORE

Distributed Computing

Module -4

Prepared by Amal Ganesh, Assistant Professor, Vidya Academy of Science and Technology, Thrissur

Distributed Computing – Text Book

From **Coulouris, Dollimore, Kindberg and Blair**

Distributed Systems:

Concepts and Design

Edition 5, © Addison-Wesley 2012

Course Plan			
Module	**Contents**	**Hours**	**End Sem. Exam Marks**
I	Evolution of Distributed Computing -Issues in designing a distributed system- Challenges- Minicomputer model – Workstation model - Workstation-Server model– Processor - pool model - Trends in distributed systems	7	15%
II	System models: Physical models - Architectural models - Fundamental models	6	15%
FIRST INTERNAL EXAM			
III	Interprocess communication: characteristics – group communication - Multicast Communication –Remote Procedure call - Network virtualization. Case study : Skype	7	15%
IV	Distributed file system: File service architecture - Network file system- Andrew file system- Name Service	7	15%
SECOND INTERNAL EXAM			
V	Transactional concurrency control:- Transactions, Nested transactions-Locks-Optimistic concurrency control	7	20%
VI	Distributed mutual exclusion – central server algorithm – ring based algorithm- Maekawa's voting algorithm – Election: Ring -based election algorithm – Bully algorithm	7	20%
END SEMESTER EXAM			

Prepared by Binu G S, Assistant Professor, Vidya Academy of Science and Technology, Thrissur

Distributed File System:

- A distributed file system enables programs to store and access remote files exactly as they do local ones, allowing users to access files from any computer on a network.
- The performance and reliability experienced for access to files stored at a server should be comparable to that for files stored on local disks.
- File systems were originally developed for centralized computer systems and desktop computers as an operating system facility providing a convenient programming interface to disk storage.
- They subsequently acquired features such as access-control and file-locking mechanisms that made them useful for the sharing of data and programs.
- Distributed file systems support the sharing of information in the form of files and hardware resources in the form of persistent storage throughout an intranet.
- Other services, such as the name service, the user authentication service and the print service, can be more easily implemented when they can call upon the file service to meet their needs for persistent storage.

Distributed File System:

- Web servers are reliant on filing systems for the storage of the web pages that they serve.
- In organizations that operate web servers for external and internal access via an intranet, the web servers often store and access the material from a local distributed file system.
- With the advent of distributed object-oriented programming, a need arose for the persistent storage and distribution of shared objects.
- Figure provides an overview of types of storage system.
- In addition to those already mentioned, the table includes distributed shared memory (DSM) systems and persistent object stores

Distributed File System:

Figure 12.1 Storage systems and their properties

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

Types of consistency:

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

Distributed File System:

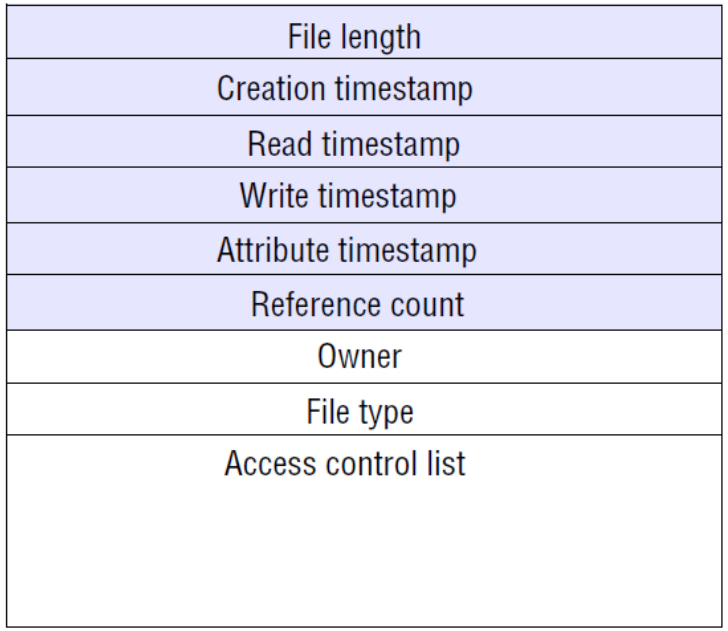
- The *consistency* column indicates whether mechanisms exist for the maintenance of consistency between multiple copies of data when updates occur.
- Virtually all storage systems rely on the use of caching to optimize the performance of programs.
- Caching was first applied to main memory and non-distributed file systems
- and for those the consistency is strict (denoted by a ‘1’, for one-copy consistency).
- Distributed file systems such as Sun NFS and the Andrew File System cache copies of portions of files at client computers, and they adopt specific consistency mechanisms to maintain an approximation to strict consistency – this is indicated by a tick (✓)

Distributed File System:

- **Characteristics of file systems**
  - File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files.
  - They provide a programming interface that characterizes the file abstraction, freeing programmers from concern with the details of storage allocation and layout.
  - Files are stored on disks or other non-volatile storage media.
  - Files contain both *data* and *attributes*.
  - The data consist of a sequence of data items (typically 8-bit bytes), accessible by operations to read and write any portion of the sequence.
  - The attributes are held as a single record containing information such as the length of the file, timestamps, file type, owner's identity and access control lists.
  - A typical attribute record structure is illustrated in Figure.
  - The shaded attributes are managed by the file system and are not normally updatable by user programs.

Distributed File System:

Figure 12.3 File attribute record structure



- File systems are designed to store and manage large numbers of files, with facilities for creating, naming and deleting files.
- The naming of files is supported by the use of directories.
- A *directory* is a file, often of a special type, that provides a mapping from text names to internal file identifiers.

Distributed File System:

- Directories may include the names of other directories, leading to the familiar hierarchic file-naming scheme and the multi-part *pathnames* for files used in UNIX and other operating systems.
- File systems also take responsibility for the control of access to files, restricting access to files according to users' authorizations and the type of access requested (reading, updating, executing and so on).
- The term *metadata* is often used to refer to all of the extra information stored by a file system that is needed for the management of files.
- It includes file attributes, directories and all the other persistent information used by the file system.

Distributed File System:

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

Figure 12.2 File system modules

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

Distributed File System:

- **File system operations**
  - Figure summarizes the main operations on files that are available to applications in UNIX systems.
  - These are the system calls implemented by the kernel; application programmers usually access them through procedure libraries such as the C Standard Input/Output Library or the Java file classes.

Figure 12.4 UNIX file system operations

filedes = open(name, mode)	Opens an existing file with the given *name*.
filedes = creat(name, mode)	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)	Puts the file attributes for file *name* into *buffer*.

Distributed File System: Requirements

- Distributed file system requirements
  - Many of the requirements and potential pitfalls in the design of distributed services were first observed in the early development of distributed file systems.
  - Initially, they offered access transparency and location transparency; performance, scalability, concurrency control, fault tolerance and security requirements emerged and were met in subsequent phases of development.
- **Transparency**
  - The file service is usually the most heavily loaded service in an intranet, so its functionality and performance are critical.
  - The design of transparency must balance the flexibility and scalability that derive from transparency against software complexity and performance.
  - The following forms of transparency are partially or wholly addressed by current file services:
  - ***Access transparency:*** Client programs should be unaware of the distribution of files. A single set of operations is provided for access to local and remote files. Programs written to operate on local files are able to access remote files without modification.

Distributed File System: Requirements

- ***Location transparency*:** Client programs should see a uniform file name space. Files or groups of files may be relocated without changing their pathnames, and user programs see the same name space wherever they are executed.
- ***Mobility transparency*:** Neither client programs nor system administration tables in client nodes need to be changed when files are moved. This allows file mobility – files or, more commonly, sets or volumes of files may be moved, either by system administrators or automatically.
- ***Performance transparency*:** Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.
- ***Scaling transparency*:** The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.

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

Distributed File System: Requirements

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

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

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

Distributed File System: Requirements

- **Fault tolerance**
  - The central role of the file service in distributed systems makes it essential that the service continue to operate in the face of client and server failures.
  - Fortunately, a moderately fault-tolerant design is straightforward for simple servers.
  - To cope with transient communication failures, the design can be based on *at-most-once* invocation semantics;
  - or it can use the simpler *at-least-once* semantics with a server protocol designed in terms of *idempotent* operations, ensuring that duplicated requests do not result in invalid updates to files.
  - Tolerance of disconnection or server failures requires file replication.
- **Consistency**
  - Conventional file systems such as that provided in UNIX offer *one-copy update semantics*.
  - This refers to a model for concurrent access to files in which the file contents seen by all of the processes accessing or updating a given file are those that they would see if only a single copy of the file contents existed.

Distributed File System: Requirements

- When files are replicated or cached at different sites, there is an inevitable delay in the propagation of modifications made at one site to all of the other sites that hold copies, and this may result in some deviation from one-copy semantics.

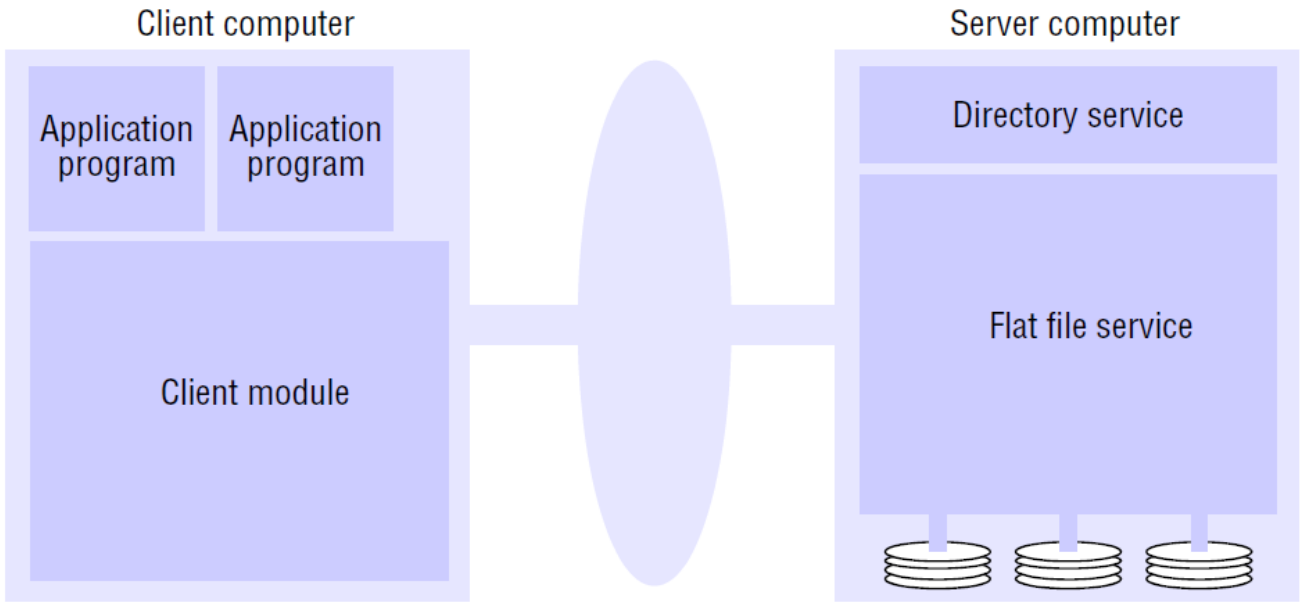
- **Security**
  - Virtually all file systems provide access-control mechanisms based on the use of access control lists.
  - In distributed file systems, there is a need to authenticate client requests so that access control at the server is based on correct user identities and to protect the contents of request and reply messages with digital signatures and (optionally) encryption of secret data.

- **Efficiency**
  - A distributed file service should offer facilities that are of at least the same power and generality as those found in conventional file systems and should achieve a comparable level of performance.

File service architecture

- **File service architecture:**
  - An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components
    - – a *flat file service*,
    - a *directory service*
    - and a *client module*.

Figure 12.5 File service architecture



File service architecture

- **Flat file service**
  - The flat file service is concerned with implementing operations on the contents of files.
  - *Unique file identifiers* (UFIDs) are used to refer to files in all requests for flat file service operations.
  - The division of responsibilities between the file service and the directory service is based upon the use of UFIDs.
  - UFIDs are long sequences of bits chosen so that each file has a UFID that is unique among all of the files in a distributed system.
  - When the flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.
- **Directory service**
  - The directory service provides a mapping between *text names* for files and their UFIDs.
  - Clients may obtain the UFID of a file by quoting its text name to the directory service.
  - The directory service provides the functions needed to generate directories, to add new file names to directories and to obtain UFIDs from directories.
  - It is a client of the flat file service; its directory files are stored in files of the flat file service.

File service architecture

- **Client module**
  - A client module runs in each client computer, integrating and extending the operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers.
  - The client module also holds information about the network locations of the flat file server and directory server processes.
  - Finally, the client module can play an important role in achieving satisfactory performance through the implementation of a cache of recently used file blocks at the client.
- **Flat file service interface**
  - Figure contains a definition of the interface to a flat file service

Figure 12.6 Flat file service operations

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

File service architecture

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

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

File service architecture

- *Repeatable operations*: With the exception of *Create*, the operations are *idempotent*, allowing the use of *at-least-once* RPC semantics – clients may repeat calls to which they receive no reply.
- Repeated execution of *Create* produces a different new file for each call.
- *Stateless servers*: The interface is suitable for implementation by *stateless* servers.
- Stateless servers can be restarted after a failure and resume operation without any need for clients or the server to restore any state.
- The UNIX file operations are neither idempotent nor consistent with the requirement for a stateless implementation.

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

File service architecture

- Two approaches mainly used:
  - An access check is made whenever a file name is converted to a UFID, and the results are encoded in the form of a capability, which is returned to the client for submission with subsequent requests.
  - A user identity is submitted with every client request, and access checks are performed by the server for every file operation.
- Both methods enable stateless server implementation, and both have been used in distributed file systems.
- The second is more common; it is used in both NFS and AFS.
- **Directory service interface:**
  - The primary purpose of the directory service is to provide a service for translating text names to UFIDs.
  - In order to do so, it maintains directory files containing the mappings between text names for files and UFIDs.
  - Each directory is stored as a conventional file with a UFID, so the directory service is a client of the file service.

File service architecture

Figure 12.7 Directory service operations

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

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

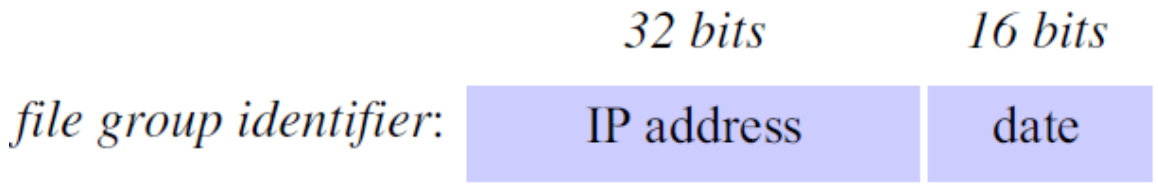
File service architecture

- The root has a distinguished name, and each file or directory has a name in a directory.
- The UNIX file-naming scheme is not a strict hierarchy – files can have several names, and they can be in the same or different directories.

- **File groups**
  - A *file group* is a collection of files located on a given server.
  - A server may hold several file groups, and groups can be moved between servers, but a file cannot change the group to which it belongs.
  - A similar construct called a *filesystem* is used in UNIX and in most other operating systems.
  - In a distributed file service, file groups support the allocation of files to file servers in larger logical units and enable the service to be implemented with files stored on several servers.
  - In a distributed file system that supports file groups, the representation of UFIDs includes a file group identifier component, enabling the client module in each client computer to take responsibility for dispatching requests to the server that holds the relevant file group.
  - File group identifiers must be unique throughout a distributed system

File service architecture

- For example, whenever a new file group is created, a unique identifier can be generated by concatenating the 32-bit IP address of the host creating the new group with a 16-bit integer derived from the date, producing a unique 48-bit integer:



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

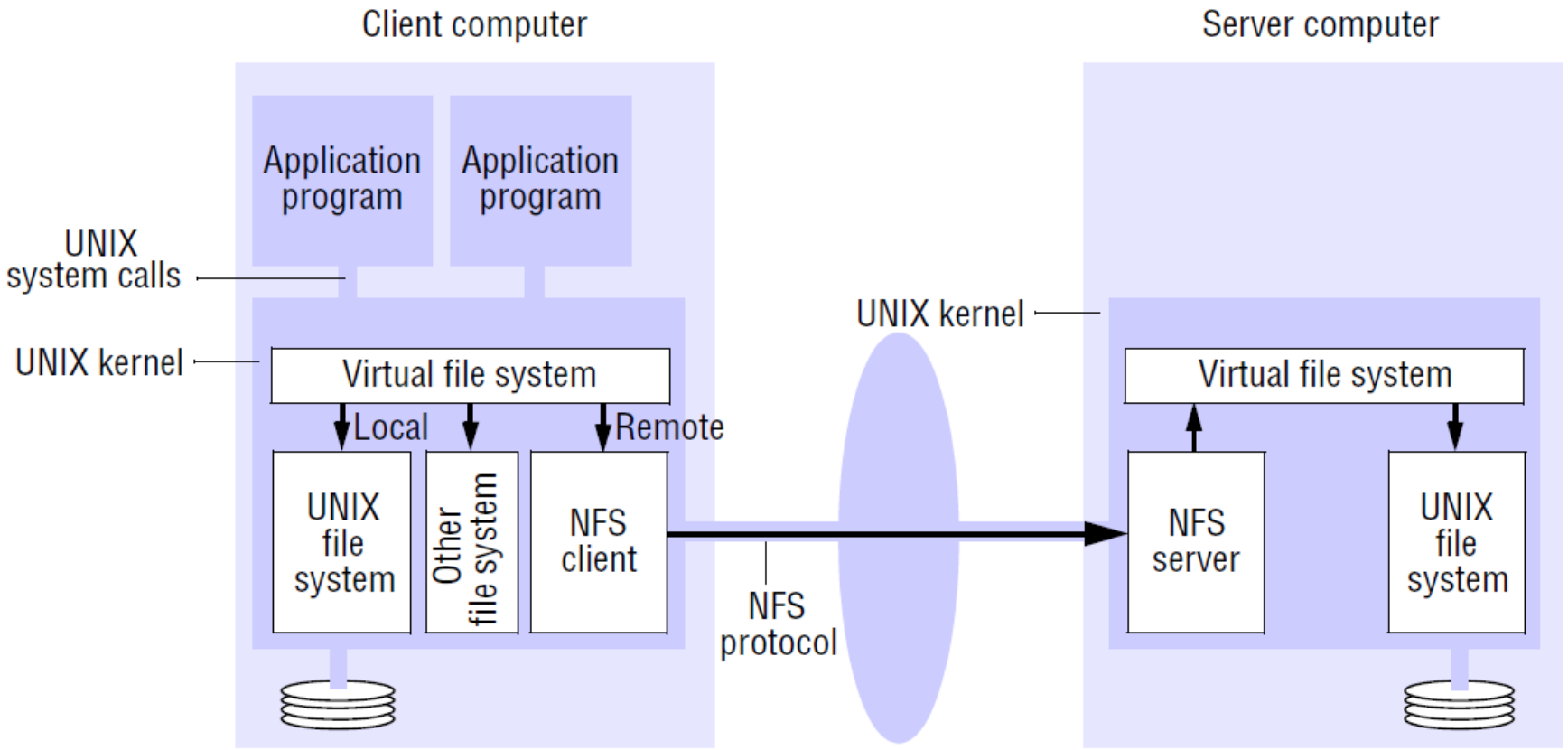
Case Study: Sun Network File System (NFS)

- Sun Microsystems's *Network File System* (NFS) has been widely adopted in industry and in academic environments since its introduction in 1985.
- 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.
- NFS provides transparent access to remote files for client programs running on UNIX and other systems.
- The client-server relationship is symmetrical: each computer in an NFS network can act as both a client and a server, and the files at every machine can be made available for remote access by other machines.
- Any computer can be a server, exporting some of its files, and a client, accessing files on other machines.

Case Study: Sun Network File System (NFS)

- NFS Architecture:
  - Figure shows the architecture of Sun NFS.

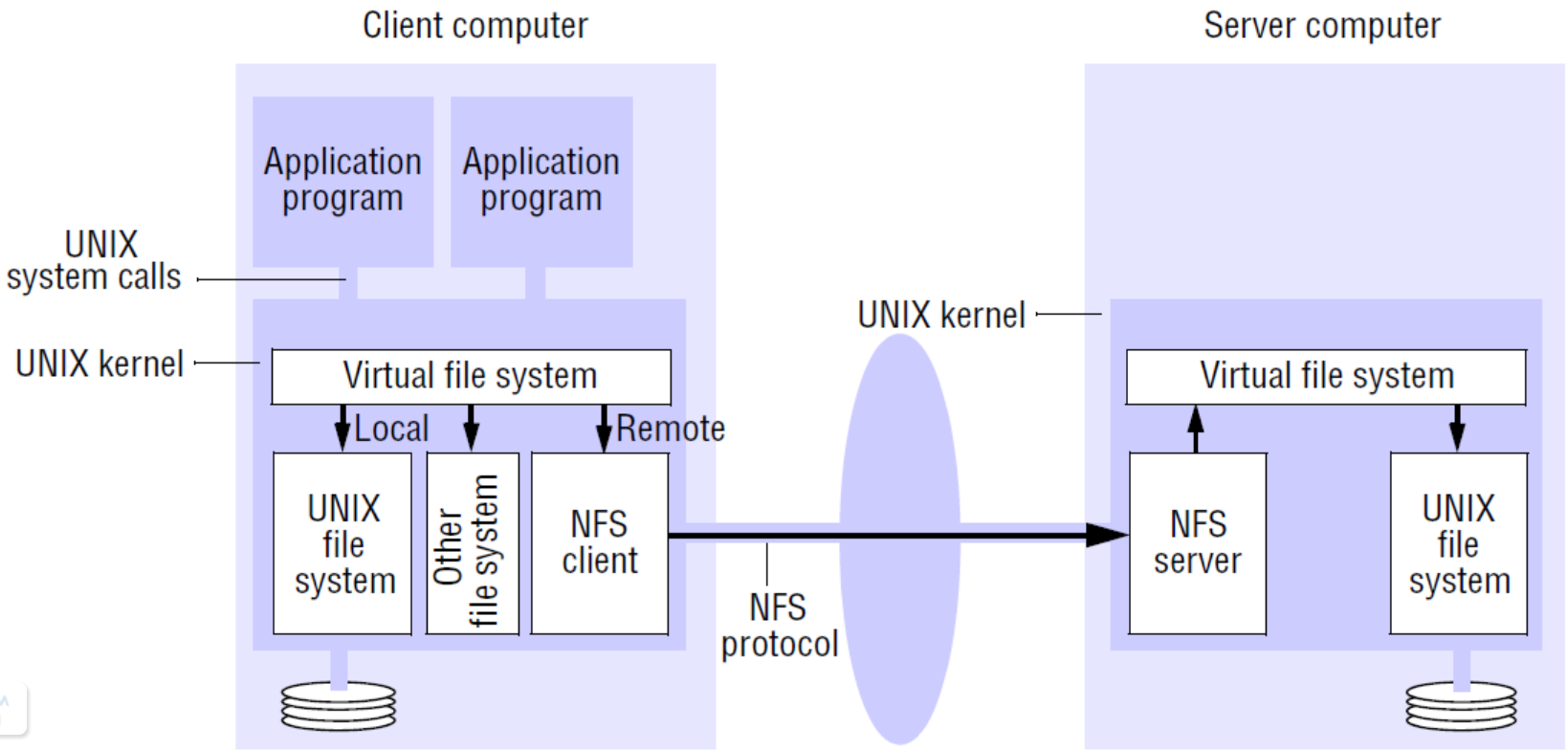
Figure 12.8 NFS architecture



Case Study: Sun Network File System (NFS)

- NFS Architecture:
  - 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.

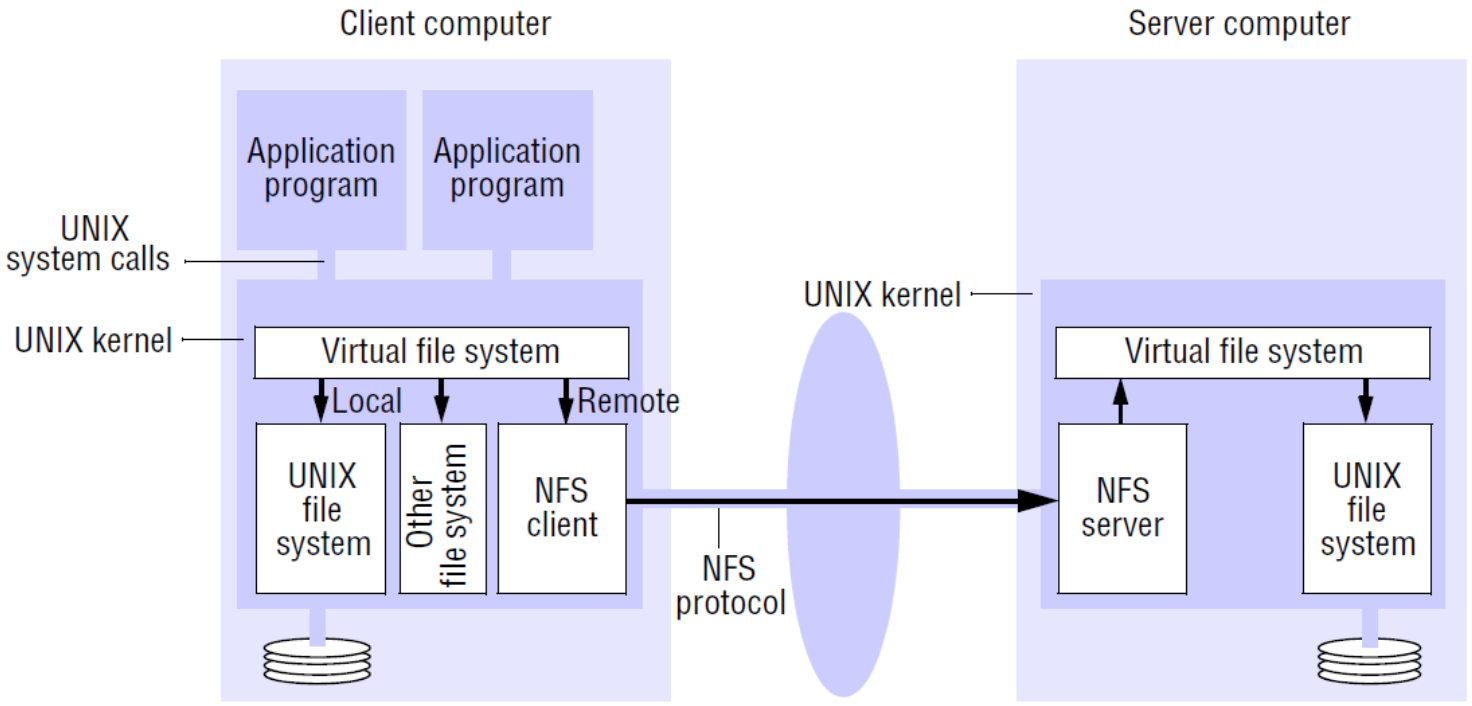
Figure 12.8 NFS architecture



Case Study: Sun Network File System (NFS)

- NFS Architecture:
  - 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.

Figure 12.8 NFS architecture



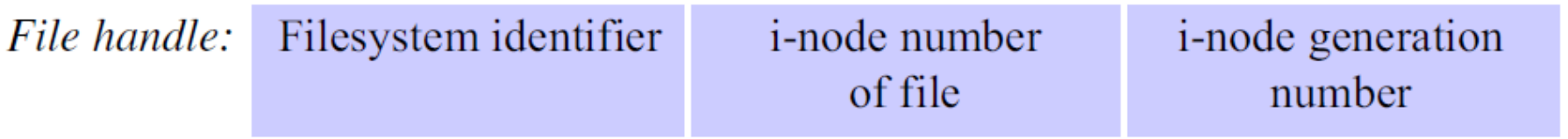
Case Study: Sun Network File System (NFS)

- 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 12.8 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.

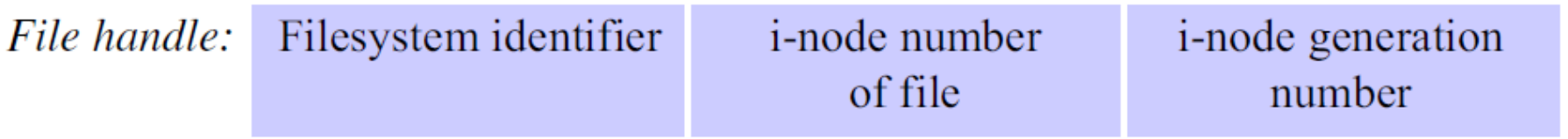
Case Study: Sun Network File System (NFS)

- 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 filesystems 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).
- The file identifiers used in NFS are called *file handles*.
- A file handle is opaque to clients and contains whatever information the server needs to distinguish an individual file.
- In UNIX implementations of NFS, the file handle is derived from the file's *i-node number* by adding two extra fields as follows (the i-node number of a UNIX file is a number that serves to identify and locate the file within the file system in which the file is stored):



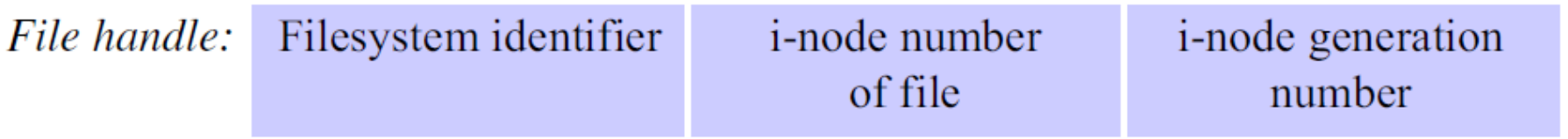
Case Study: Sun Network File System (NFS)

- The *filesystem identifier* field is a unique number that is allocated to each filesystem when it is created (and in the UNIX implementation is stored in the superblock of the file system).
- The *i-node generation number* is needed because in the conventional UNIX file system i-node numbers are reused after a file is removed.
- In the VFS extensions to the UNIX file system, a generation number is stored with each file and is incremented each time the i-node number is reused
- The virtual file system layer has one VFS structure for each mounted file system and one *v-node* per open file.
- A VFS structure relates a remote file system to the local directory on which it is mounted.
- The v-node contains an indicator to show whether a file is local or remote.
- If the file is local, the v-node contains a reference to the index of the local file (an i-node in a UNIX implementation). If the file is remote, it contains the file handle of the remote file.



Case Study: Sun Network File System (NFS)

- The *filesystem identifier* field is a unique number that is allocated to each filesystem when it is created (and in the UNIX implementation is stored in the superblock of the file system).
- The *i-node generation number* is needed because in the conventional UNIX file system i-node numbers are reused after a file is removed.
- In the VFS extensions to the UNIX file system, a generation number is stored with each file and is incremented each time the i-node number is reused
- The virtual file system layer has one VFS structure for each mounted file system and one *v-node* per open file.
- A VFS structure relates a remote file system to the local directory on which it is mounted.
- The v-node contains an indicator to show whether a file is local or remote.
- If the file is local, the v-node contains a reference to the index of the local file (an i-node in a UNIX implementation). If the file is remote, it contains the file handle of the remote file.



Case Study: Sun Network File System (NFS)

- **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.
  - 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;
    - the encryption key used to authenticate user IDs passed to the server can be retained in the kernel, preventing impersonation by user-level clients.
- **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

Case Study: Sun Network File System (NFS)

- **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 12.9.

Figure 12.9 NFS server operations (NFS version 3 protocol, simplified)

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*.

Case Study: Sun Network File System (NFS)

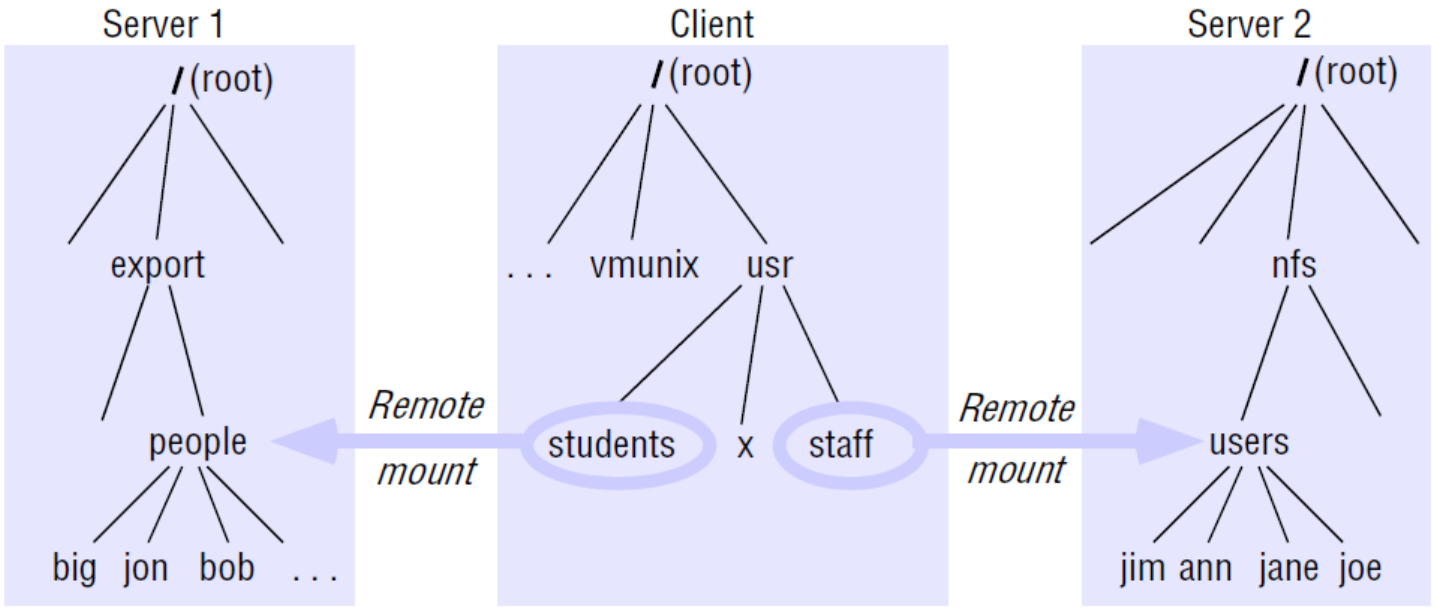
- The NFS file access operations *read*, *write*, *getattr* and *setattr* are almost identical to the *Read*, *Write*, *GetAttributes* and *SetAttributes* operations defined for our flat file service model (Figure 12.6).
- The *lookup* operation and most of the other directory operations defined in Figure 12.9 are similar to those in our directory service model (Figure 12.7).

- **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.
  - Clients use a modified version of the UNIX *mount* command to request mounting of a remote filesystem, specifying the remote host's name, the pathname of a directory in the remote filesystem and the local name with which it is to be mounted.
  - The modified *mount* command communicates with the mount service process on the remote host using a *mount protocol*. (RPC Protocol)

Case Study: Sun Network File System (NFS)

- Figure 12.10 illustrates a *Client* with two remotely mounted file stores.
- The nodes *people* and *users* in filesystems at *Server 1* and *Server 2* are mounted over nodes *students* and *staff* in *Client*'s local file store.
- The meaning of this is that programs running at *Client* can access files at *Server 1* and *Server 2* by using pathnames such as */usr/students/jon* and */usr/staff/ann*.

Figure 12.10 Local and remote filesystems accessible on an NFS client



Note: The file system mounted at */usr/students* in the client is actually the subtree located at */export/people* in Server 1; the filesystem mounted at */usr/staff* in the client is actually the subtree located at /nfs/users in Server 2.

Case Study: Sun Network File System (NFS)

- **Pathname translation**
  - UNIX file systems translate multi-part file pathnames to i-node references in a step-by-step process whenever the open, creat or 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 mayreside in filesystems 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.

Case Study: Sun Network File System (NFS)

- **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 filesystem at the first server to respond.
  - Automounter keeps the mount table small.
  - Automounter provides a simple form of replication for read-only filesystems.
  - 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.

Case Study: Sun Network File System (NFS)

- **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.

Case Study: Sun Network File System (NFS)

- **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.
  - A timestamp-based method is used to validate cached blocks before they are used.
  - Each data or metadata item in the cache is tagged with two timestamps:
    - *Tc* is the time when the cache entry was last validated.
    - *Tm* is the time when the block was last modified at the server.
  - A cache entry is valid at time *T* if *T* – *T*c is less than a freshness interval *t*, or if the value for *Tm* recorded at the client matches the value of *Tm* at the server (that is, the data has not been modified at the server since the cache entry was made).

Case Study: Sun Network File System (NFS)

- Formally, the validity condition is:

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

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

- The selection of a value for *t* involves a compromise between consistency and efficiency.
- A very short freshness interval will result in a close approximation to one-copy consistency, at the cost of a relatively heavy load of calls to the server to check the value of *Tmserver*.
- In Sun Solaris clients, *t* is set adaptively for individual files to a value in the range 3 to 30 seconds, depending on the frequency of updates to the file.
- For directories the range is 30 to 60 seconds, reflecting the lower risk of concurrent updates.

Case Study: Sun Network File System (NFS)

- There is one value of Tmserver for all the data blocks in a file and another for the file attributes.
- Since NFS clients cannot determine whether a file is being shared or not, the validation procedure must be used for all file accesses.
- A validity check is performed whenever a cache entry is used.
- The first half of the validity condition can be evaluated without access to the server. If it is true, then the second half need not be evaluated;
- if it is false, the current value of Tmserver is obtained (by means of a getattr call to the server) and compared with the local value Tmclient.
- If they are the same, then the cache entry is taken to be valid and the value of Tc for that cache entry is updated to the current time.
- If they differ, then the cached data has been updated at the server and the cache entry is invalidated, resulting in a request to the server for the relevant data.

Case Study: Sun Network File System (NFS)

- Several measures are used to reduce the traffic of *getattr* calls to the server:
  - Whenever a new value of Tm_{server} is received at a client, it is applied to all cache entries derived from the relevant file.
  - The current attribute values are sent 'piggybacked' with the results of every operation on a file, and if the value of Tm_{server} has changed the client uses it to update the cache entries relating to the file.
  - The adaptive algorithm for setting freshness interval *t* outlined above reduces the traffic considerably for most files.
  - The selection of a value for *t* involves a compromise between consistency and efficiency.

Case Study: Sun Network File System (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**
  - identified two remaining problem areas:
    - frequent use of the *getattr* call in order to fetch timestamps from servers for cache validation;
    - relatively poor performance of the *write* operation because write-through was used at the server.
  - 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.

Case Study: Sun Network File System (NFS)

- **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.
  - No modifications to existing programs are required to enable them to operate correctly with 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.

Case Study: Sun Network File System (NFS)

- Scalability transparency:
  - File systems (file groups) may be subdivided and allocated to separate servers.
  - Ultimately, the performance limit is determined by the load on the server holding the most heavily-used filesystem (file group).
- 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.
  - The Sun Network Information Service (NIS) is a separate service available for use with NFS that supports the replication of simple databases organized as key-value pairs (for example, the UNIX system files */etc/passwd* and */etc/hosts*).
  - It manages the distribution of updates and accesses to the replicated files based on a simple master–slave replication model
- 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.

Case Study: Sun Network File System (NFS)

- Fault tolerance:
  - Limited but effective; service is suspended if a server fails. Recovery from failures is aided by the simple stateless design.
  - When a server fails, the service that it provides is suspended until the server is restarted, but once it has been restarted user-level client processes proceed from the point at which the service was interrupted, unaware of the failure
- 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, 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: Andrew File System (AFS)

- **Andrew File System** 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.
- 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.

Case Study: Andrew File System (AFS)

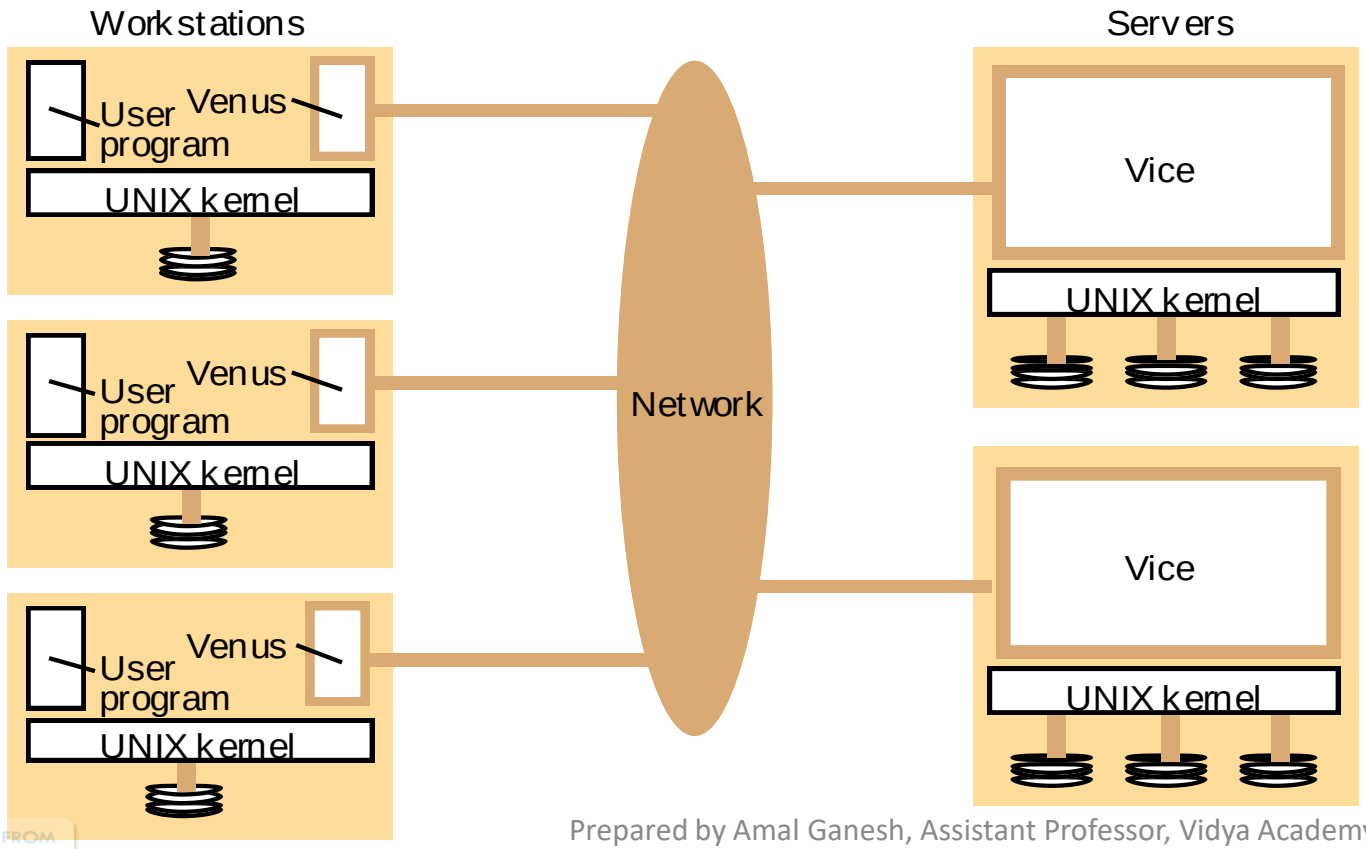
- AFS has two unusual design characteristics:
  - ***Whole-file serving*:** The entire contents of directories and files are transmitted to client computers by AFS servers (in AFS-3, files larger than 64 kbytes are transferred in 64-kbyte chunks).
  - ***Whole-file caching*:** Once a copy of a file or 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.

Case Study: Andrew File System (AFS)

- **Scenario: 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.

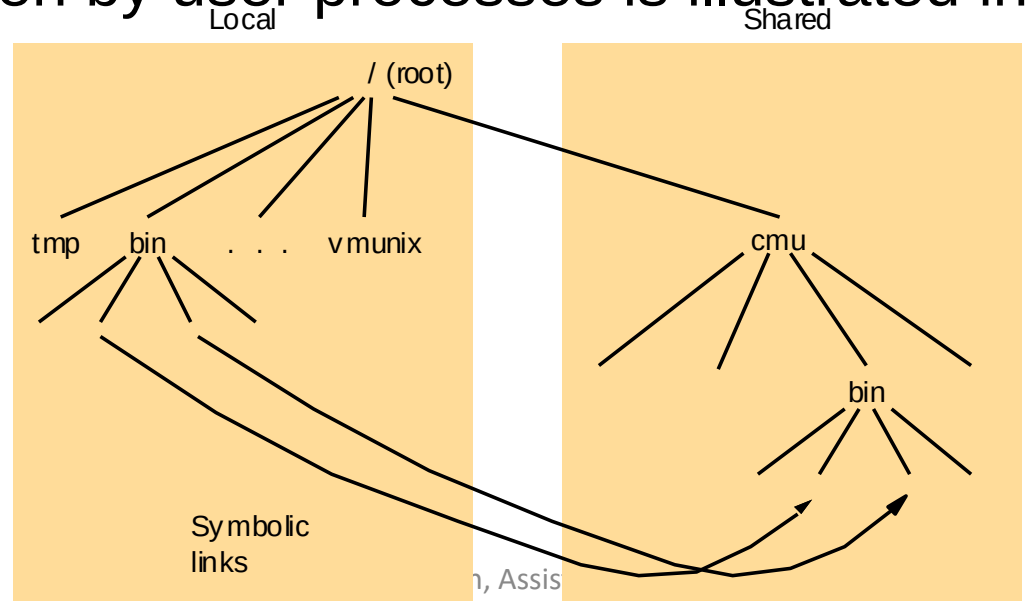
Case Study: Andrew File System (AFS)

- **AFS Implementation and Architecture**
  - 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.



Case Study: Andrew File System (AFS)

- Vice is the name given to the server software that runs as a user-level UNIX process in each server computer, and Venus is a user-level process that runs in each client computer and corresponds to the client module in our abstract model.
- 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 a 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.



Case Study: Andrew File System (AFS)

- The name space seen by user processes is illustrated in Figure 12.

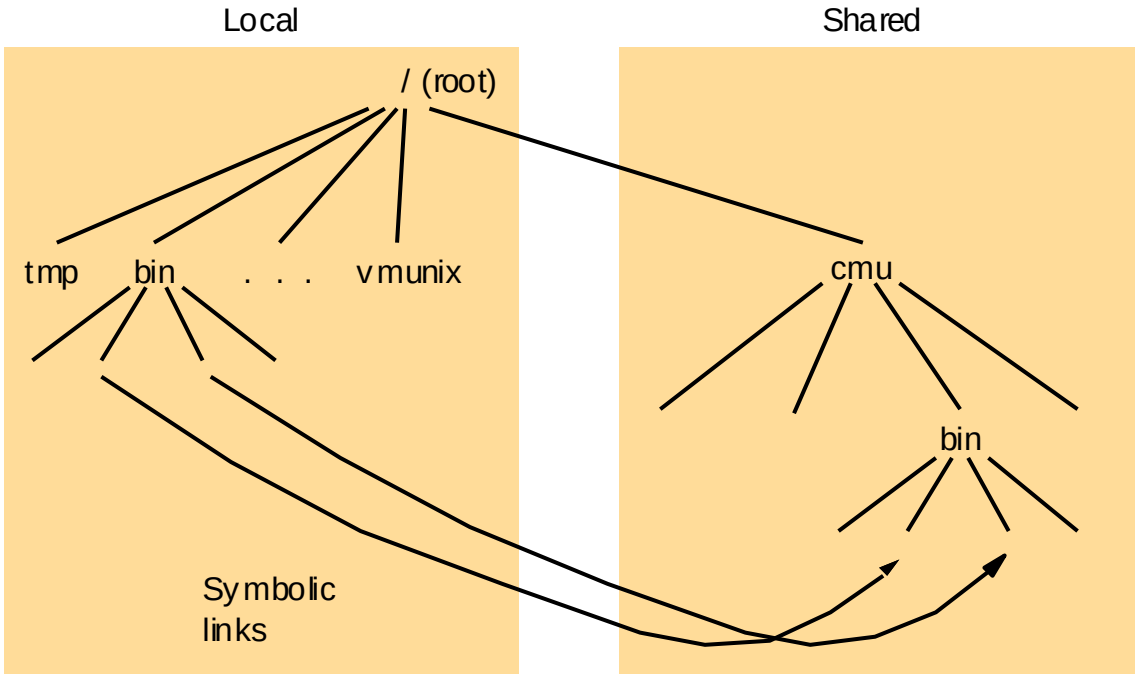


Figure 12. File name space seen by clients of AFS

- It is a conventional UNIX directory hierarchy, with a specific subtree (called *cmu*) containing all of the shared files.
- This splitting of the file name space into local and shared files leads to some loss of location transparency, but this is hardly noticeable to users other than system administrators.

Case Study: Andrew File System (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 (illustrated in Figure 12.13)

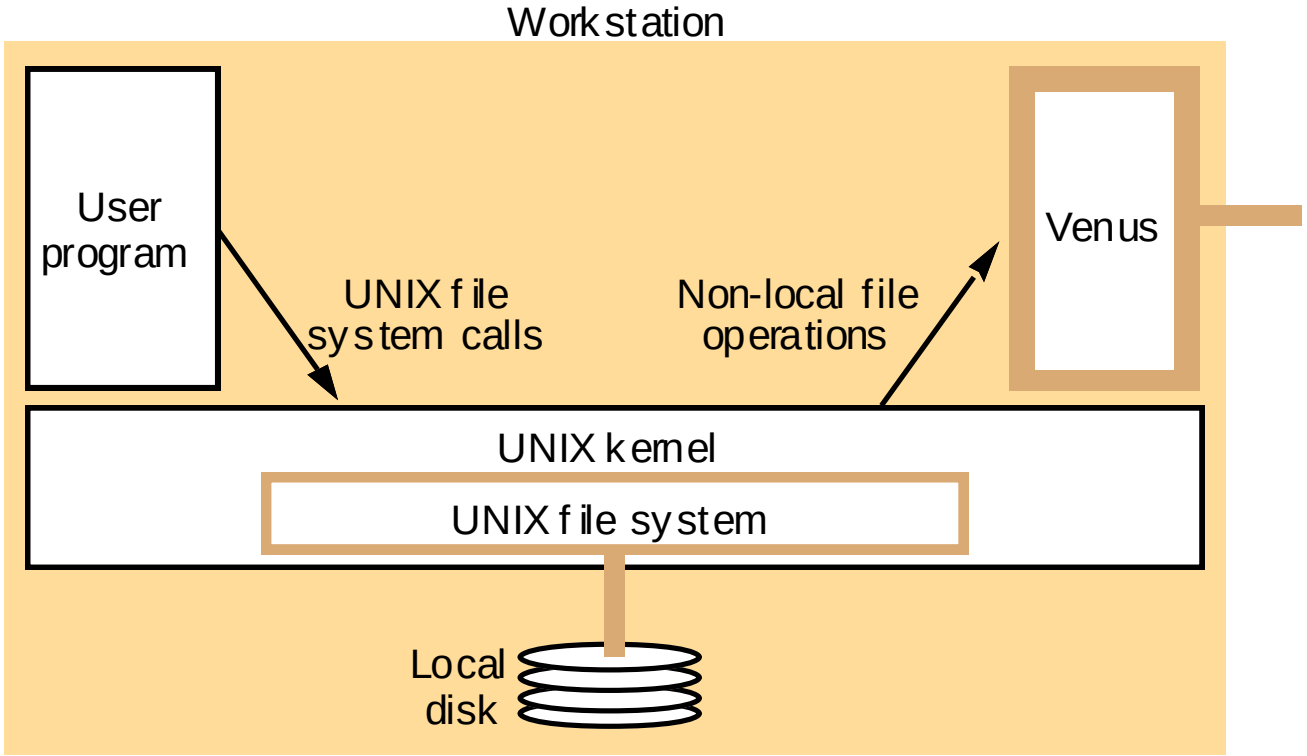


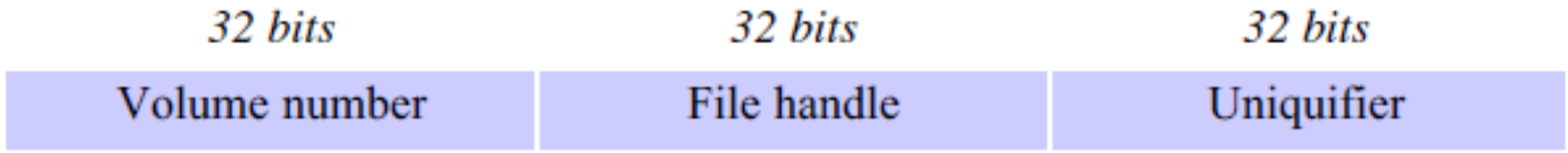
Figure 13. System call interception in AFS

Case Study: Andrew File System (AFS)

- One of the file partitions on the local disk of each workstation is used as a cache, holding the cached copies of files from the shared space.
- Venus manages the cache, removing the least recently used files when a new file is acquired from a server to make the required space if the partition is full.
- The workstation cache is usually large enough to accommodate several hundred average-sized files, rendering the workstation largely independent of the Vice servers once a working set of the current user's files and frequently used system files has been cached.
- AFS resembles the abstract file service model:
  - A flat file service is implemented by the Vice servers, and the hierarchic directory structure required by UNIX user programs is implemented by the set of Venus processes in the workstations.
  - Each file and directory in the shared file space is identified by a unique, 96-bit file identifier (fid) similar to a UFID.
  - The Venus processes translate the pathnames issued by clients to fids.

Case Study: Andrew File System (AFS)

- Files are grouped into *volumes* for ease of location and movement. Volumes are generally smaller than the UNIX filesystems, which are the unit of file grouping in NFS.
- For example, each user's personal files are generally located in a separate volume.
- Other volumes are allocated for system binaries, documentation and library code.
- The representation of *fid*s includes the volume number for the volume containing the file (*cf.* the *file group identifier* in UFIDs), an NFS file handle identifying the file within the volume (*cf.* the *file number* in UFIDs) and a *uniquifier* to ensure that file identifiers are not reused:



Case Study: Andrew File System (AFS)

- **Cache consistency**
  - When Vice supplies a copy of a file to a Venus process it also provides a **callback promise** – a token issued by the Vice server that is the custodian of the file, guaranteeing that it will notify the Venus process when any other client modifies the file.
  - The callback promise mentioned here is a mechanism for ensuring that cached copies of files are updated when another client closes the same file after updating it.
  - Callback promises are stored with the cached files on the workstation disks and have two states: **valid or cancelled**.
  - When a server performs a request to update a file it notifies all of the Venus processes to which it has issued callback promises by sending a callback to each – a callback is a remote procedure call from a server to a Venus process.
  - When the Venus process receives a callback, it sets the callback promise token for the relevant file to cancelled.
  - Whenever Venus handles an open on behalf of a client, it checks the cache. If the required file is found in the cache, then its token is checked. If its value is cancelled, then a fresh copy of the file must be fetched from the Vice server, but if the token is valid, then the cached copy can be opened and used without reference to Vice.

Case Study: Andrew File System (AFS)

- Figure 12.14 describes the actions taken by Vice, Venus and the UNIX kernel when a user process issues each of the system calls

User process	*UNIX kernel*	*Venus*	*Net*	*Vice*
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus. Open the local file and return the file descriptor to the application.	Check list of files in local cache. If not present or there is no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file. Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	→ ←	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.	→	Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

Figure 14. implementation of file system calls in AFS

Case Study: Andrew File System (AFS)

- Figure 12.15 shows the RPC calls provided by AFS servers for operations on files (that is, the interface provided by AFS servers to Venus processes).

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.

Figure 15. The main components of the Vice service interface

Name Services

- A **naming service** is the principal mechanism used in distributed and non-distributed systems for referring to objects from within your applications via a name identifying that object.
- The term *name* used here typically means a name that is **human readable** or at least easily converted into a human-readable String format.
- A file system uses **filenames** when doing out references associated with file media to programs requesting access to those files.
- Names
  - Identification of objects
  - Resource sharing: Internet domain names
  - Examples of human-readable names are
    - file names such as */etc/passwd*,
    - URLs such as *http://www.cdk5.net/*
    - and Internet domain names such as *www.cdk5.net.*
  - Communication: domain name part of email address

Name Services

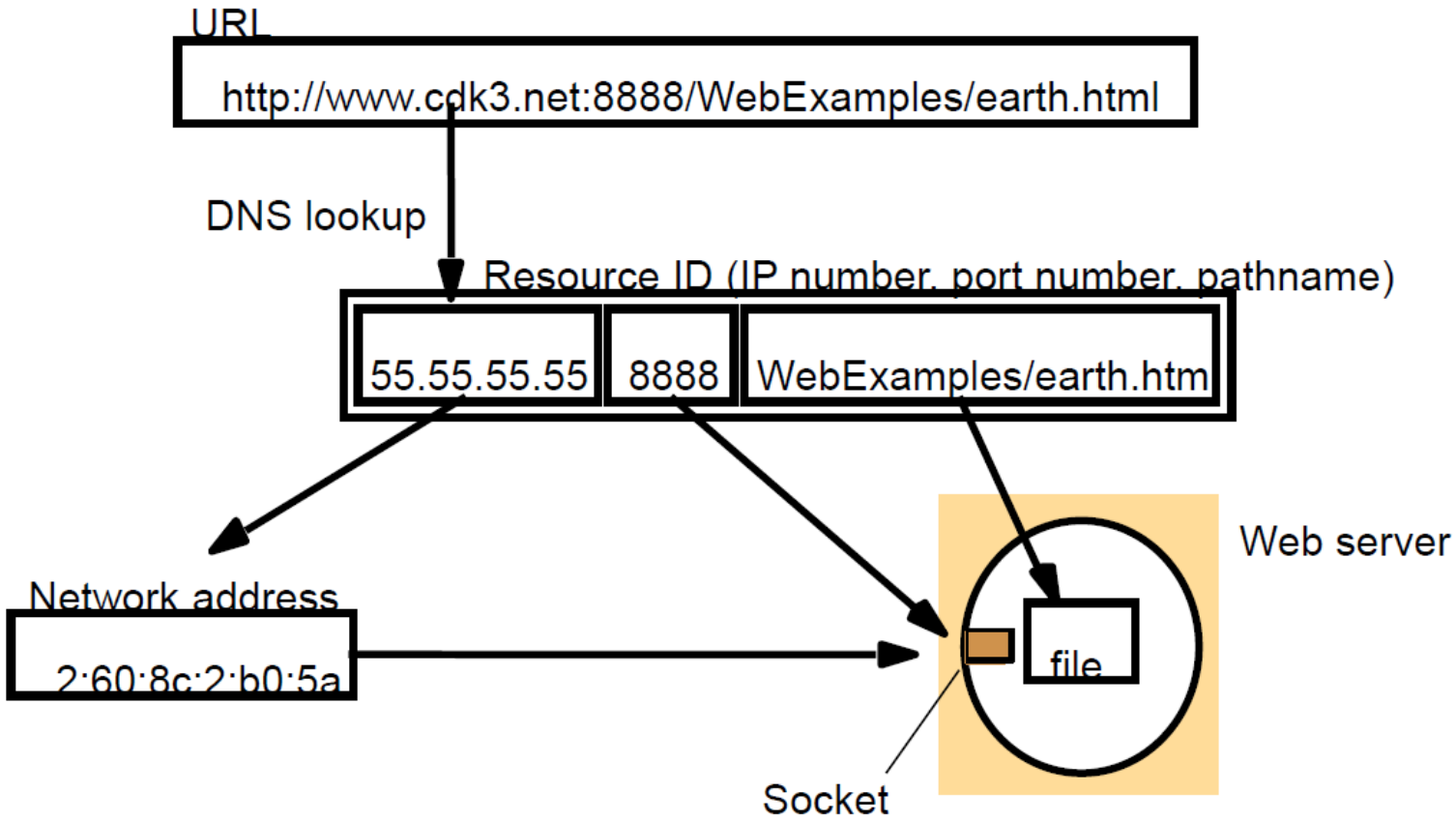
- How much information about an object is in a name?
- * Pure and Non-pure/Impure names (Needham)
- Pure names: uninterpreted bit patterns
  - - the name itself yields no information e.g. UID
  - - contains no location information
  - - commits system to nothing
  - - can only be used to compare with other similar bit-patterns e.g. in table look-up
- Non-pure names: contain information about the object, e. g., its location
  - examples: email addresses
  - foo.cl.cam.ac.uk
  - host-ID, object-ID (unless used ONLY to generate UIDs)
  - disc-pack-ID, object-ID
  - - the name yields information
  - - commits the system to maintaining the context in which the name is to be resolved
  - e.g. the directory hierarchy uk/ac/cam/cl

Name Services

- A name is ***resolved*** when it is translated into data about the named resource or object, often in order to invoke an action upon it.
- The association between a name and an object is called a ***binding***.
- 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)
- The X500 directory service can be used to map a person's name onto attributes including their email address and telephone number.
- The CORBA Naming Service and Trading Service. The Naming Service maps the name of a remote object onto its remote object reference, whereas the Trading Service maps the name of a remote object onto its remote object reference, together with an arbitrary number of attributes describing the object in terms understandable by human users.

Name Services

- Figure 13.1 shows the domain name portion of a URL resolved first via the DNS into an IP address and then, at the final hop of Internet routing, via ARP to an Ethernet address for the web server.
- The last part of the URL is resolved by the file system on the web server to locate the relevant file.



Name Services

- **Names and services**
  - Uniform Resource Identifiers
    - 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.
    - Turning to an example of incorporating existing identifiers, that has been done for telephone numbers by prefixing them with the scheme name tel and standardizing their representation, as in tel:+1-816-555-1212.
    - These tel URIs are intended for uses such as web links that cause telephone calls to be made when invoked.
  - Uniform Resource Locators:
    - Some URIs contain information that can be used to locate and access a resource; others are pure resource names.
    - The familiar term Uniform Resource Locator (URL) is often used for URIs that provide location information and specify the method for accessing the resource, including the ‘http’
    - For example, http://www.cdk5.net/ identifies a web page at the given path (‘/’) on the host www.cdk5.net, and specifies that the HTTP protocol be used to access it.
    - Another example is a ‘mailto’ URL, such as mailto:fred@flintstone.org, which identifies the mailbox at the given address.

Name Services

- **Names and services**
  - Uniform Resource Names:
    - Uniform Resource Names (URNs) are URIs that are used as pure resource names rather than locators.
    - For example, the URI: mid:0E4FC272-5C02-11D9-B115-000A95B55BC8@hpl.hp.com is a URN that identifies the email message containing it in its ‘Message-Id’ field.
    - The URI distinguishes that message from any other email message.
    - But it does not provide the message’s address in any store, so a lookup operation is needed to find it.
    - For example, urn:ISBN:0-201-62433-8 identifies books that bear the name 0-201-62433-8 in the standard ISBN naming scheme.
- **Name services and the Domain Name System**
  - A *name service* stores information about a collection of textual names, in the form of bindings between the names and the attributes of the entities they denote, such as users, computers, services and objects.
  - The collection is often subdivided into one or more naming *contexts*: individual subsets of the bindings that are managed as a unit.

Name Services

- **Requirements of Name Service**
  - Usage of a convention for unique global naming
    - Enables sharing
    - It is often not easily predictable which services will eventually share resources
  - Scalability
    - Naming directories tend to grow very fast, in particular in Internet
  - Consistency
    - Short and mid-term inconsistencies tolerable
    - In the long term, system should converge towards a consistent state
  - Performance and availability
    - peed and availability of lookup operations
    - Name services are at the heart of many distributed applications
  - Adaptability to change
    - Organizations frequently change structure during lifetime
  - Fault isolation
    - System should tolerate failure of some of its servers

Name Services

- **Name Space**
  - A name space is the collection of all valid names recognized by a particular service. The service will attempt to look up a valid name, even though that name may prove not to correspond to any object – i.e., to be unbound.
  - Name spaces require a syntactic definition to separate valid names from invalid names. For example, '...' is not acceptable as the DNS name of a computer, whereas www.cdk99.net is valid (even though it is unbound).
  - Set of all valid names to be used in a certain context, e. g., all valid URLs in WWW
  - Can be described using a generative grammar (e. g., BNF for URLs).
- Internal structure
  - Names may have an internal structure that represents their position in a hierarchic name space such as pathnames
    - in a file system,
    - or in an organizational hierarchy such as Internet domain names;
    - or they may be chosen from a flat set of numeric or symbolic identifiers.
  - One important advantage of a hierarchy is that it makes large name spaces more manageable.

Name Services

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

- Hierarchic name spaces are potentially infinite,
  - So they enable a system to grow indefinitely.
  - By contrast, flat name spaces are usually finite; their size is determined by fixing a maximum permissible length for names.
  - Another potential advantage of a hierarchic name space is that different contexts can be managed by different people or organizations.
- DNS names are strings called domain names.
  - Eg:- www.cdk5.net (a computer), net, com and ac.uk (the latter three are domains).
  - The DNS name space has a hierarchic structure: a domain name consists of one or more strings called name components or labels, separated by the delimiter ‘.’.

Name Services

- There is no delimiter at the beginning or end of a domain name, although the root of the DNS name space is sometimes referred to as ‘.’ for administrative purposes.
- DNS names are not case-sensitive, so www.cdk5.net and WWW.CDK5.NET have the same meaning.
- DNS servers do not recognize relative names: all names are referred to the global root.

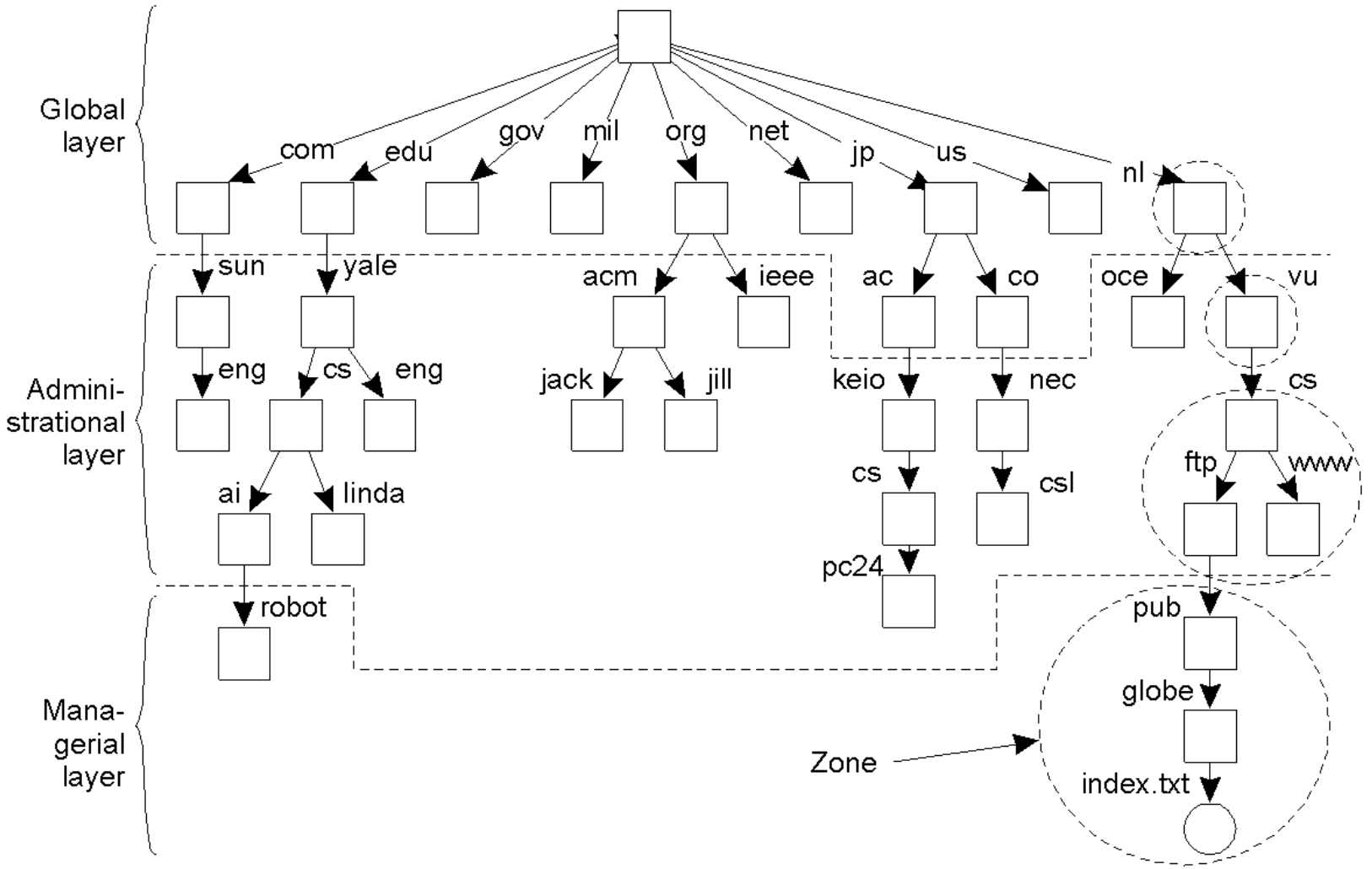
- **Aliases**
  - An *alias* is a name defined to denote the same information as another name, similar to a symbolic link between file path names.
  - Aliases allow more convenient names to be substituted for relatively complicated ones, and allow alternative names to be used by different people for the same entity.
  - An example is the common use of URL shorteners, often used in Twitter posts and other situations where space is at a premium.
  - For example, using web redirection, *http://bit.ly/ctqjvH* refers to *http://cdk5.net/additional/rmi/programCode/ShapeListClient.java*.
  - For example, the name *www.cdk5.net* is an alias for *cdk5.net*.
  - This has the advantage that clients can use either name for the web server, and if the web server is moved to another computer, only the entry for cdk5.net needs to be updated in the DNS database.

Name Services

- **Naming domains**
  - A *naming domain* is a name space for which there exists a single overall administrative authority responsible for assigning names within it.
  - This authority is in overall control of which names may be bound within the domain, but it is free to delegate this task.
  - Domains in DNS are collections of domain names; syntactically, a domain's name is the common suffix of the domain names within it, but otherwise it cannot be distinguished from, for example, a computer name.
  - For example, *net* is a domain that contains *cdk5.net*.
  - Note that the term 'domain name' is potentially confusing, since only some domain names identify domains (others identify computers).

Name Services

- **Name Space Distribution**



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

Name Space Distribution (2)

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

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

Name Services

Issues in Name & Directory Services

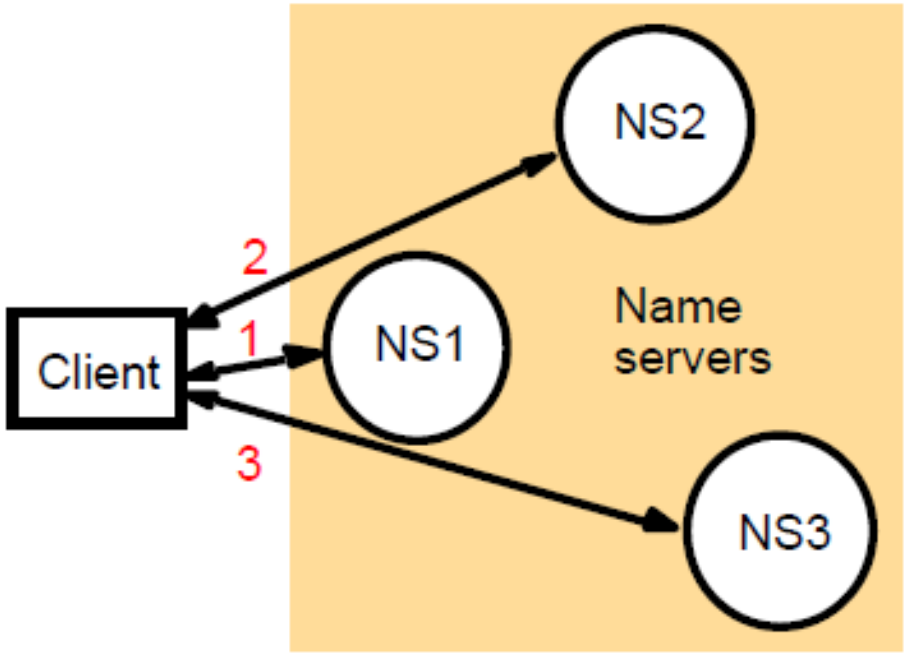
- Partitioning
  - No one name server can hold all name and attribute entries for entire network, in particular in Internet
  - Name server data partitioned according to domain
- Replication
  - A domain has usually more than one name server
  - Availability and performance are enhanced
- Caching
  - Servers may cache name resolutions performed on other servers
    - Avoids repeatedly contacting the same name server to look up identical names
  - Client lookup software may equally cache results of previous requests

Name Services

- **Name Resolution**
  - Translation of a name into the related primitive attribute
  - Often, an iterative process
    - Name service returns attributes if the resolution can be performed in it's naming context
    - Name service refers query to another context if name can't be resolved in own context
  - Deal with cyclic alias references, if present
    - Abort resolution after a predefined number of attempts, if no result obtained
- Name Servers and Navigation
  - The process of locating naming data from more than one name server in order to resolve a name is called navigation.
  - The client name resolution software carries out navigation on behalf of the client. It communicates with name servers as necessary to resolve a name.

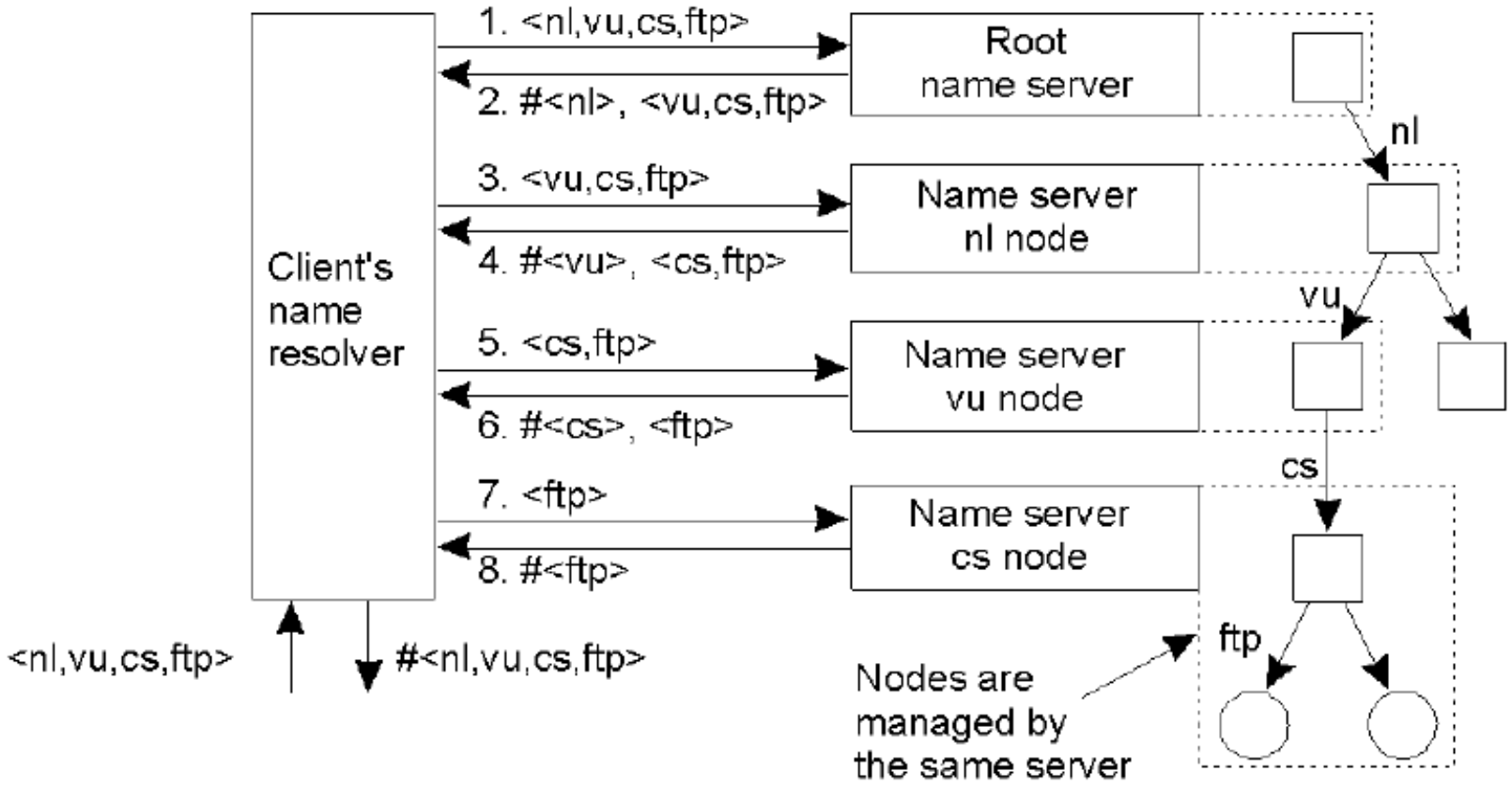
Name Services

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



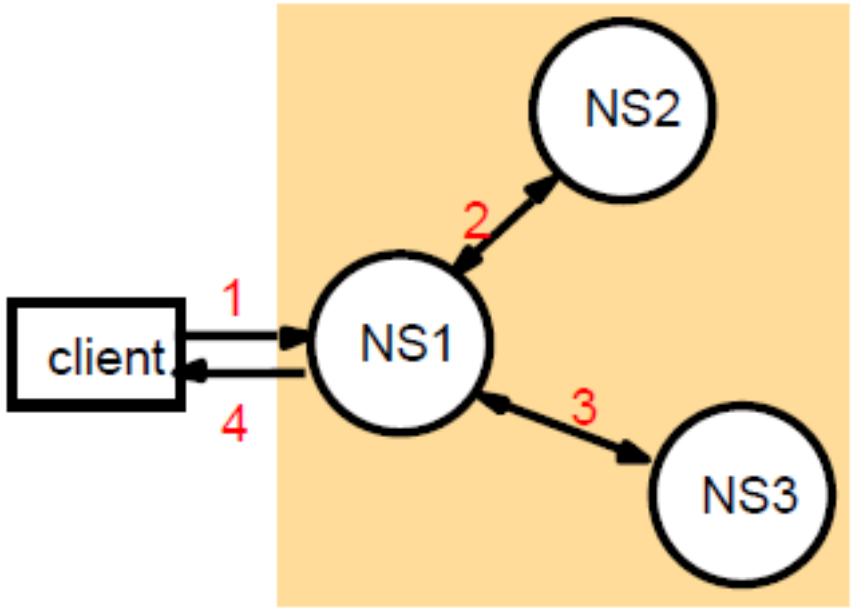
Name Services

Iterative Navigation Example



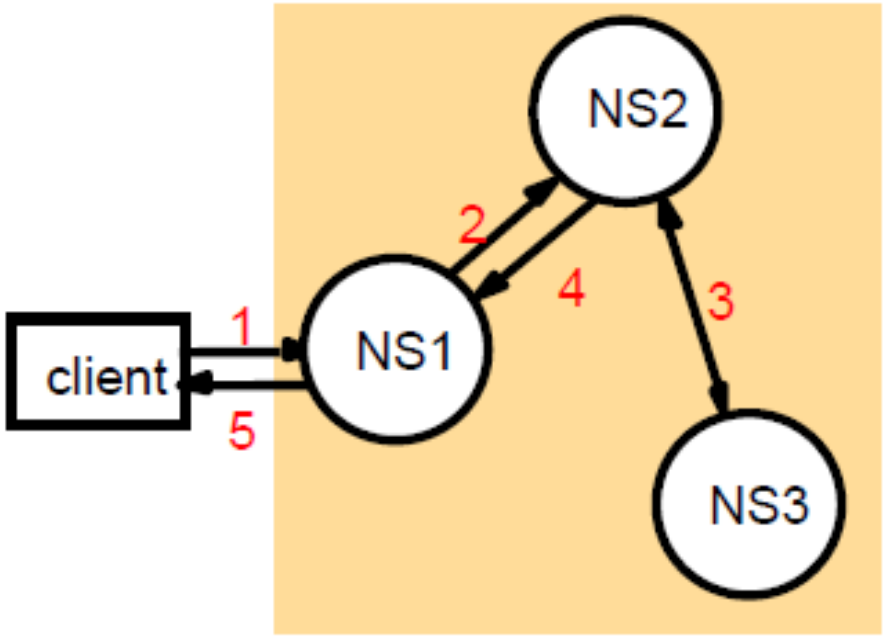
Name Services

- Multicast Navigation
  - In multicast navigation, a client multicasts the name to be resolved and the required object type to the group of name servers.
  - Only the server that holds the named attributes responds to the request.
  - Unfortunately, however, if the name proves to be unbound, the request is greeted with silence.
- Non-recursive server-controlled navigation
  - Any name server may be chosen by the client.
  - This server communicates by multicast or iteratively with its peers, as though it were a client.



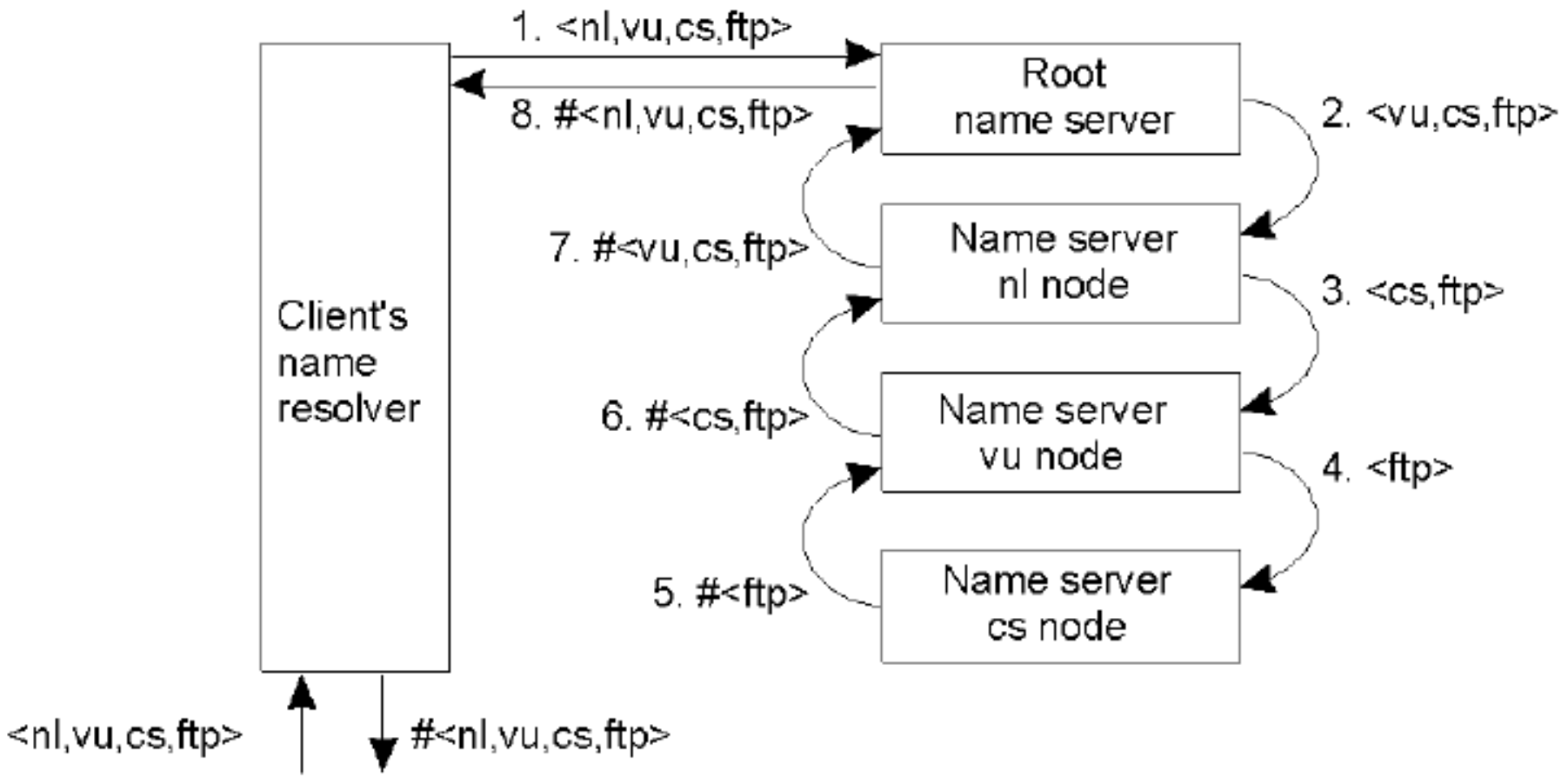
Name Services

- Recursive server-controlled navigation
  - Under recursive server-controlled navigation, the client once more contacts a single server.
  - If name cannot be resolved, server contacts superior server responsible for a larger prefix of the name space
    - Recursively applied until name resolved
    - Can be used when clients and low-level servers are not entitled to directly contact high-level servers



Name Services

Recursive Navigation Example



Name Services

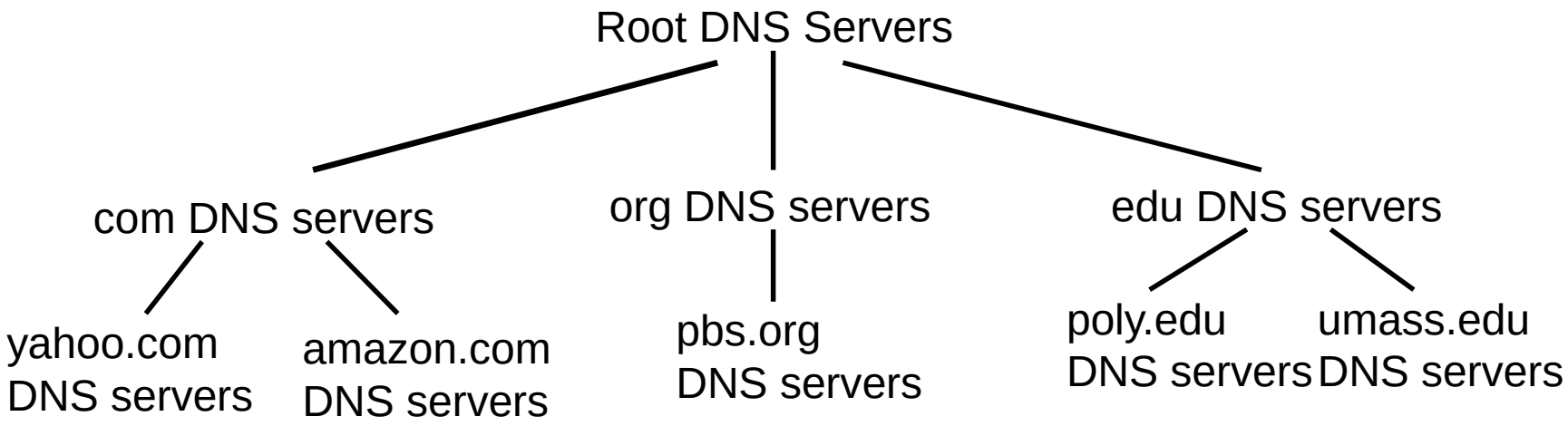
- **Caching in Name servers and DNS.**
  - In DNS and other name services, client name resolution software and servers maintain a cache of the results of previous name resolutions.
  - When a client requests a name lookup, the name resolution software consults its cache.
  - If it holds a recent result from a previous lookup for the name, it returns it to the client; otherwise, it sets about finding it from a server.
  - That server, in turn, may return data cached from other servers.
  - Caching is key to a name service's performance and assists in maintaining the availability of both the name service and other services in spite of name server crashes.
  - Enhance response times by saving communication with name servers.

Name Services

- The Domain Name System (DNS)
  - The Domain Name System is a name service design whose main naming database is used across the Internet.
  - Performs name-to-IP mapping
  - 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.
  - Distributed database
    - Implemented in hierarchy of many name servers
  - Application-layer protocol
    - Host, routers, name servers to communicate to resolve names (address/name translation)
- Why not centralize DNS?
  - Single point of failure
  - Traffic volume
  - Distant centralized database
  - Maintenance
  - Doesn't scale!

Name Services

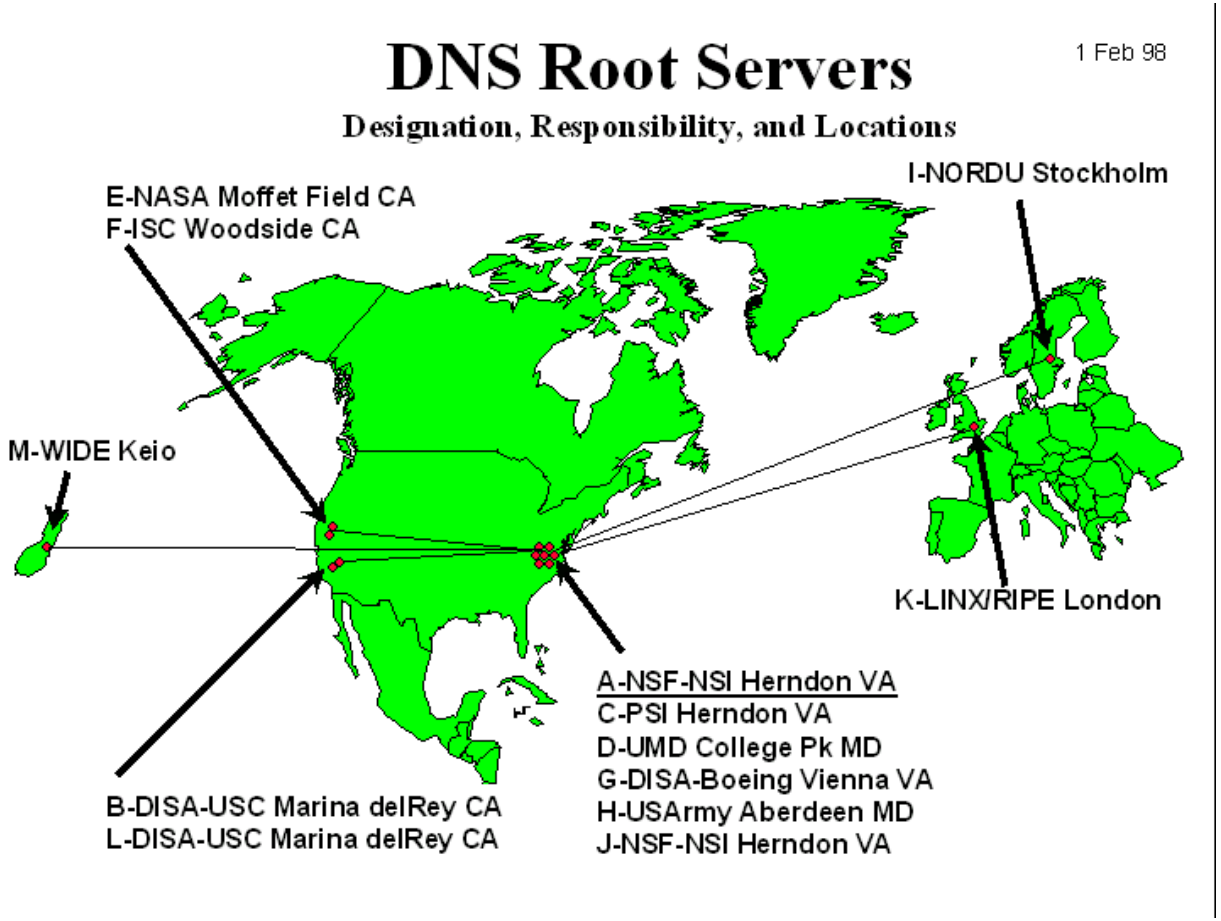
- **Distributed, Hierarchical Database**



- **DNS Types:**
  - Top-level domain (TLD) servers:
    - Responsible for com, org, net, edu, etc, and all top-level country domains uk, fr, ca, jp.
    - Network Solutions maintains servers for com TLD
  - Authoritative DNS servers:
    - organization's DNS servers, providing authoritative hostname to IP mappings for organization's servers (e.g., Web, mail).
    - can be maintained by organization or service provider
  - Local Name Server
    - does not strictly belong to hierarchy
    - each ISP (residential ISP, company, university) has one.
    - also called "default name server"
    - when host makes DNS query, query is sent to its local DNS server acts as proxy, forwards query into hierarchy

Name Services

- Root Name Servers
  - contacts authoritative name server if name mapping not known
  - gets mapping
  - returns mapping to local name server
  - Approximately dozen root name servers worldwide



Name Services

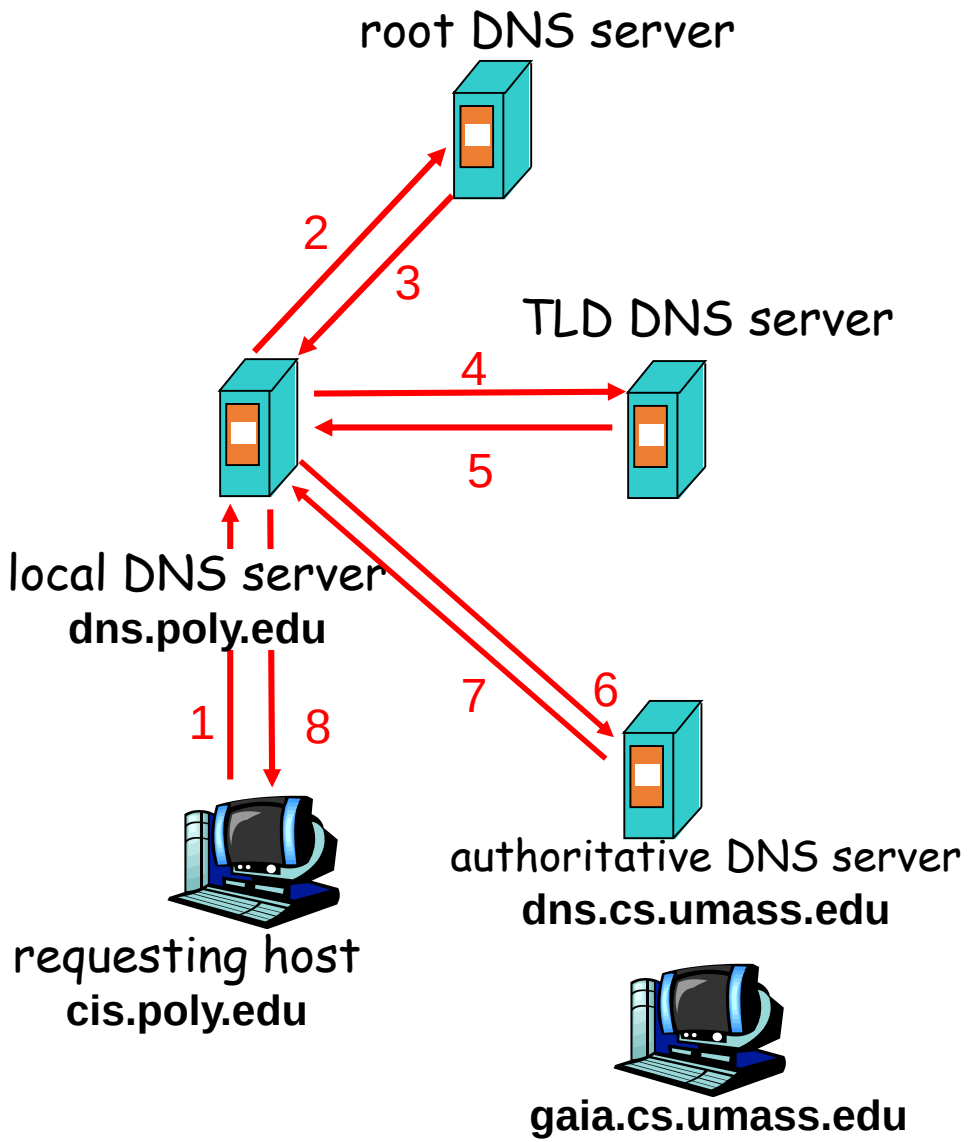
- **Domain Types**
  - The original top-level organizational domains (also called generic domains) in use across the Internet were:
    - com – Commercial organizations
    - edu – Universities and other educational institutions
    - gov – US governmental agencies
    - mil – US military organizations
    - 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

Name Services

- **DNS queries:**
  - Host name resolution:
    - In general, applications use the DNS to resolve host names into IP addresses.
    - For example, when a web browser is given a URL containing the domain name www.dcs.qmul.ac.uk, it makes a DNS enquiry and obtains the corresponding IP address.
  - Mail host location:
    - Electronic mail software uses the DNS to resolve domain names into the IP addresses of mail hosts – i.e., computers that will accept mail for those domains.
    - For example, when the address tom@dcs.rnx.ac.uk is to be resolved, the DNS is queried with the address dcs.rnx.ac.uk and the type designation ‘mail’.
    - It returns a list of domain names of hosts that can accept mail for dcs.rnx.ac.uk, if such exist (and, optionally, the corresponding IP addresses).
  - Reverse resolution:
    - Some software requires a domain name to be returned given an IP address.
    - This is just the reverse of the normal host name query, but the name server receiving the query replies only if the IP address is in its own domain.
  - Host information:
    - The DNS can store the machine architecture type and operating system with the domain names of hosts.
    - It has been suggested that this option should not be used in public, because it provides useful information for those attempting to gain unauthorized access to computers.

DNS Navigation and Query Processing

- Host at cis.poly.edu wants IP address for gaia.cs.umass.edu



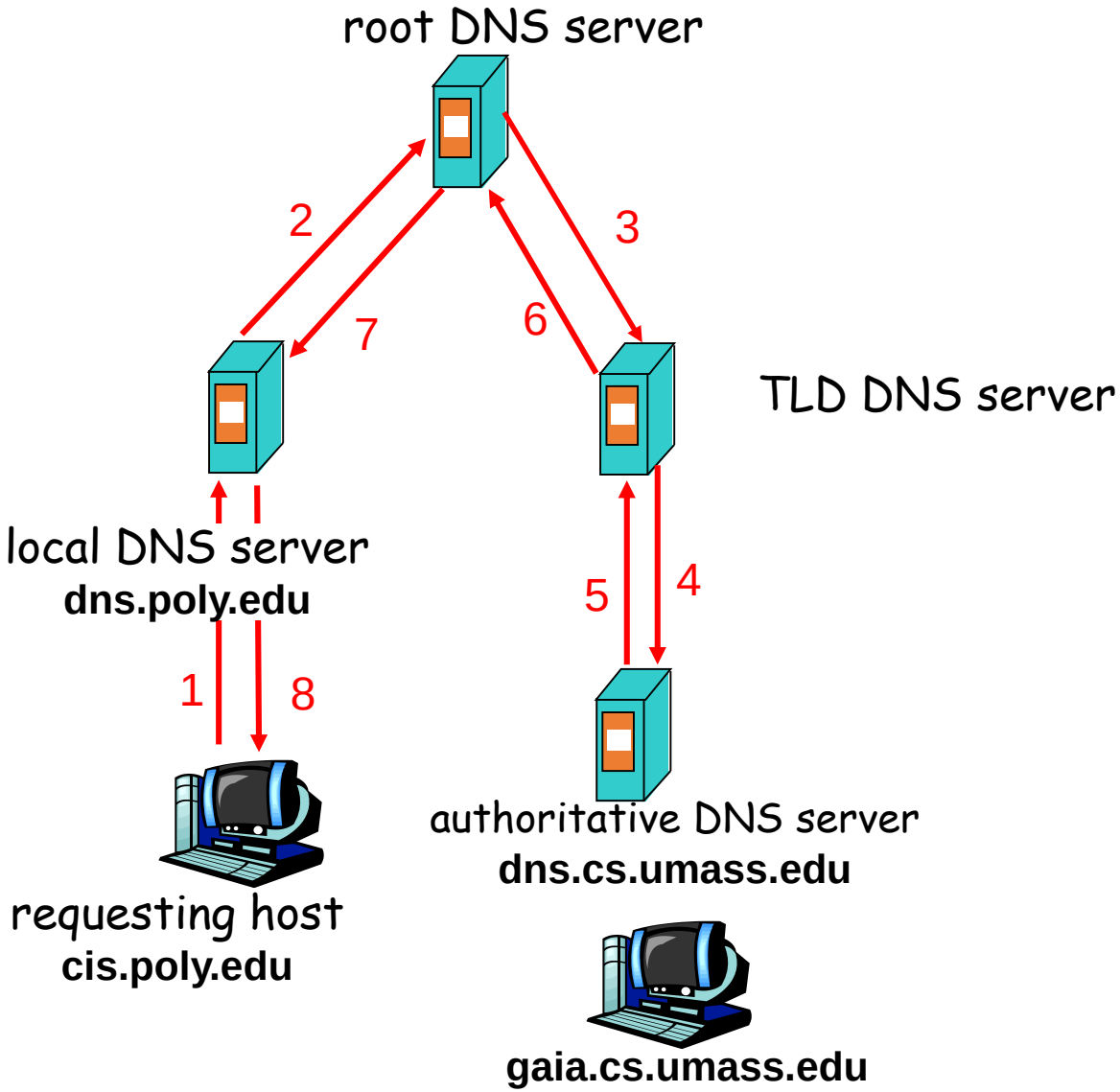
iterated query:

- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"

DNS Navigation and Query Processing

recursive query:

- puts burden of name resolution on contacted name server
- heavy load?



Name Services

- **Domain Types**
  - 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

Name Services

- **DNS Records:**
  - DNS holds resource records (RR)
    - RR format: (name, value, type,ttl)

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

Name Services

- **DNS Records:**
  - Example records:
    - Type=A
      - name is hostname
      - value is IP address
    - Type=NS
      - name is domain (e.g. foo.com)
      - value is hostname of authoritative name server for this domain
    - Type=CNAME
      - name is an alias name for some “canonical” (the real) name
      - value is canonical name
    - Type=MX
      - value is hostname of mail server associated with name
    - Type=TXT
      - TXT entries are included to allow arbitrary other information to be stored along with domain names.
    - Type =SOA
      - 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.

Name Services

- For example, part of the database for the domain dcs.qmul.ac.uk at one point is shown in Figure 13.6,
  - where the time to live 1D means 1 day.
  - Further records of type A later in the database give the IP addresses for the two name servers dns0 and dns1.
  - The IP addresses of the mail hosts and the third name server are given in the databases corresponding to their domains.

Figure 13.6 DNS zone data records

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

Name Services

- 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*

Name Services

- Load Sharing by Name Servers:
  - High overload is balanced by sharing the same domain name across several computers.
  - When a domain name is shared by several computers, there is one record for each computer in the group, giving its IP address.
  - By default, the name server responds to queries for which multiple records match the requested name by returning the IP addresses according to a round-robin schedule.
  - Successive clients are given access to different servers so that the servers can share the workload.
  - Caching has a potential for spoiling this scheme, for once a nonauthoritative name server or a client has the server's address in its cache it will continue to use it.
  - To counteract this effect, the records are given a short time to live.
- The BIND implementation of the DNS
  - The Berkeley Internet Name Domain (BIND) is an implementation of the DNS for computers running UNIX.
  - Client programs link in library software as the resolver.
  - DNS name server computers run the named daemon.
  - A typical organization has one primary server, with one or more secondary servers that provide name serving on different local area networks at the site.
  - Additionally, individual computers often run their own caching-only server, to reduce network traffic and speed up response times still further.

END