DISTRIBUTD COMPUTING MODULE 4

DISTRIBUTED FILE SYSTEM

THE SHARING OF STORED INFORMATION

- The most important aspect of distributed resource sharing
- Mechanisms for data sharing take many forms
- **Web servers** provide a **restricted form of data sharing**
  - files stored locally in file s/ms at the server or in servers on a local n/w
  - made available to clients throughout the Internet
- Requirements for sharing within local networks and intranets
- **Type of service** –
  - **persistent storage of data & programs** of all types on behalf of clients
  - **consistent distribution of up-to-date data.**

- File systems
  - originally developed for centralized computer systems and desktop computers
  - as an operating system facility
  - provides a convenient programming interface to disk storage

- acquired features such as access-control and file-locking mechanisms

- 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
  - persistent storage throughout an intranet

- A well designed file service provides
  - access to files stored at a server with performance
  - reliability similar to & sometimes better than files stored on local disks

- programs store and access remote files exactly like local ones

- allow users to access files from any computer in an intranet

File system modules

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

- The Web uses caching extensively both
  - at client computers
  - at proxy servers maintained by user organizations

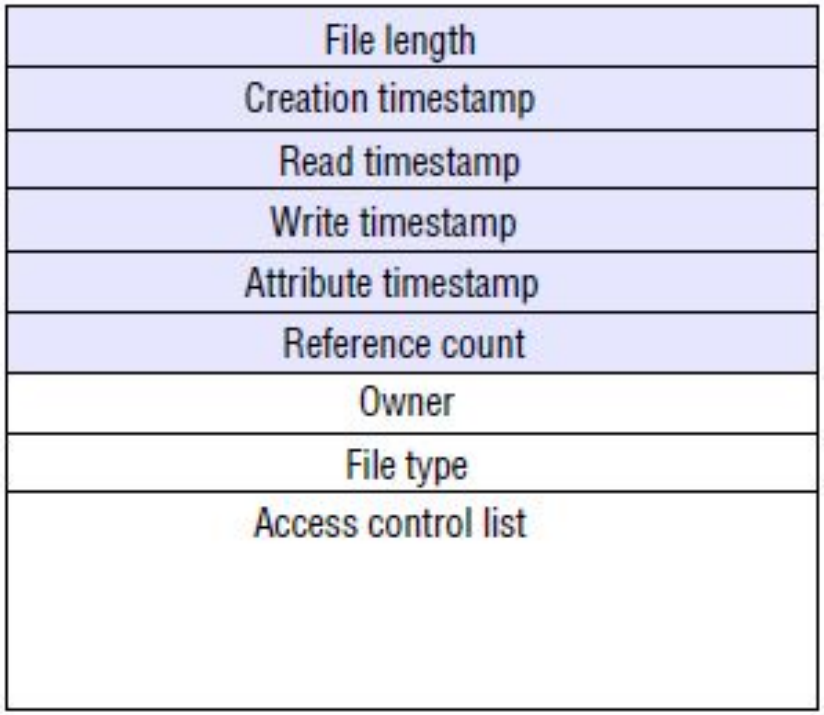
- The consistency between the copies stored is maintained by explicit user actions

Characteristics of file systems

- Responsible for the organization, storage, retrieval, naming, sharing and protection of files
- Provide a programming interface that characterizes file abstraction
- Free 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 as shown below:

File attribute record structure



- A *directory is a file that provides a mapping* from text names to internal file identifiers.

- File systems also take responsibility for the
  - control of access to files,
  - restricting access to files according to users' authorizations
  - type of access requested (reading, updating, executing etc)

- *metadata is used to refer to all the extra information stored*

- *It* is needed for the management of files

- It includes file attributes, directories etc.

File system operations

ure 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

Transparency

Access transparency

Location transparency (uniform file name space)

Mobility transparency

Performance transparency

Scaling transparency

Concurrent file updates

- Changes to a file by one client should not interfere with the operation of other clients
- Need for concurrency control for access to shared data in many applications
- Techniques are costly
- Most current file services follow modern UNIX standards

File replication

- a file may have several copies at different locations
  - multiple servers to share the load of providing a service to clients
  - enhances fault tolerance
- Few file services support replication fully,

- Most support caching of files or portions of files locally

Hardware and operating system heterogeneity

- Service interfaces should be defined

- Client and server software can be implemented for different operating systems and computers

- This is an important aspect of openness

Fault tolerance

- service should continue to operate in the face of client and server failures

- Moderately fault-tolerant design is straightforward for simple servers

- 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

- The servers can be *stateless*
  - *so that they can be restarted*
  - *the service restored after a failure without any* need to recover previous state

- Tolerance of disconnection or server failures requires file replication (which is more difficult to achieve)

Consistency

- Conventional file systems such as that provided in UNIX offer *one-copy update semantics.*

- ***Concurrent access to files*** *in which the file* contents seen by all

- As if only a single copy of the file contents existed

- Inevitable **delay in the propagation** of modifications made at one site to all of the other sites that hold copies

- This may result in some deviation from one-copy semantics

Security

- Virtually all file systems provide access-control mechanisms
- 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
  - to protect the contents of request and reply messages with
    - digital signatures
    - encryption of secret data

Efficiency

- Compared to a conventional file system distributed file system should:
  - offer facilities of at least the same power and generality
  - achieve a comparable level of performance.

- **Modular architecture:**
  - range of services addressing **requirements of clients** with different goals

- **Techniques** for **implementation of file services** are an **important** part of the design of distributed systems

- Distributed file system:
  - comparable with or better than, local file systems in performance and reliability
  - It must be convenient to administer
  - operations and tools to install and operate the system conveniently

FILE SERVICE ARCHITECTURE

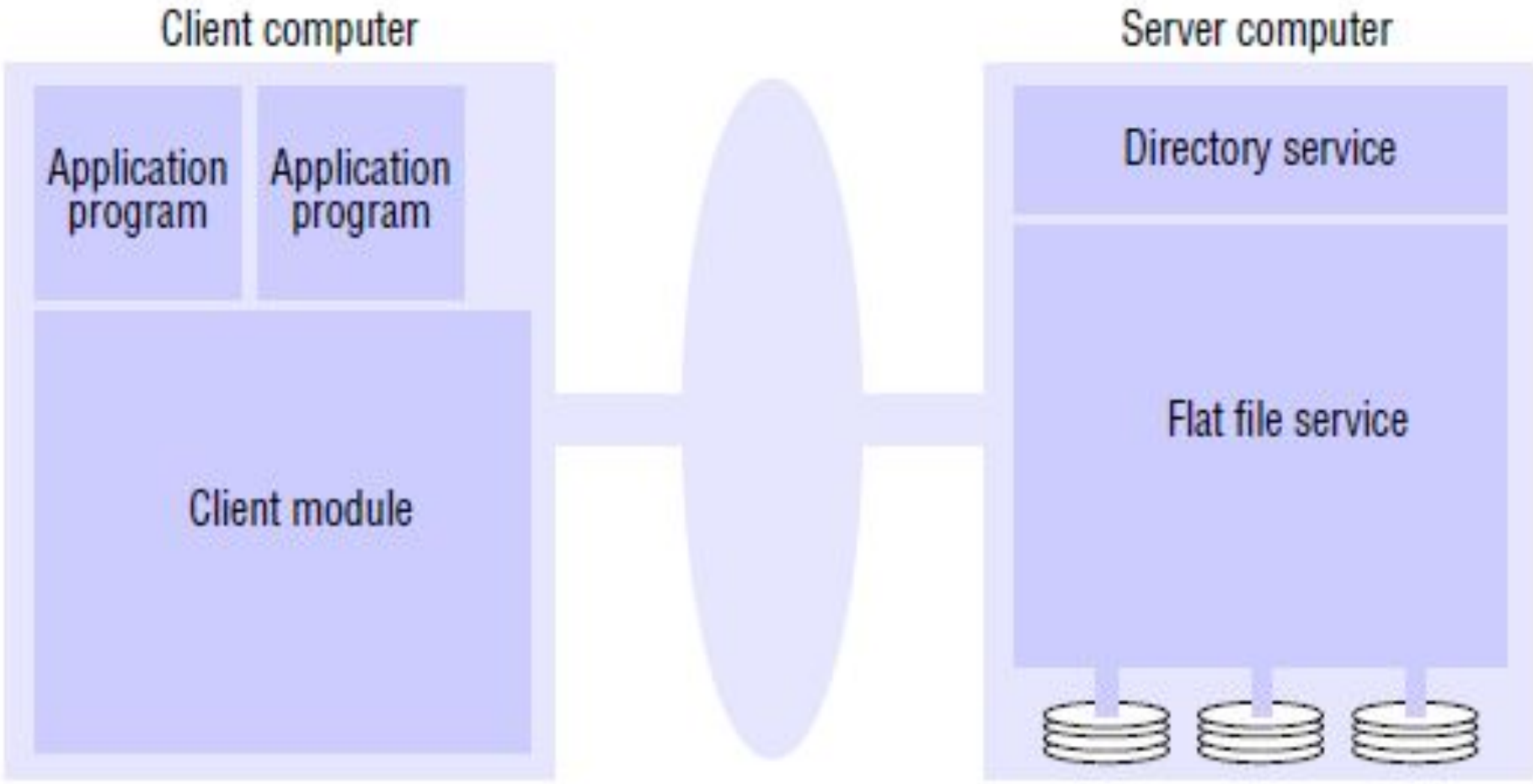
- an abstract architectural model (underpins NFS & AFS)

- Clear separation of the main concerns in providing file access

- File service is structured as three components
  - a *flat file service*
  - a *directory service*
  - *a client module*

- The **design is *open***
  - implement different programming interfaces
  - simulates the file operations of a variety of different operating systems
  - optimizes the performance for different client and server hardware configurations

File service architecture



FLAT FILE SERVICE

- 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
- UFIDs are long sequences of bits- unique throughout the DS
- When the flat file service receives a request to create a file, it:
  - generates a new UFID for it
  - returns the UFID to the requester

DIRECTORY SERVICE

- mapping between *text names for* files and their UFIDs
- Clients obtain the UFID by quoting file's text name to the directory service
- The directory service provides the functions needed to
  - generate directories
  - add new file names to directories
  - 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
- Hierarchic file-naming scheme: directories hold references to other directories

CLIENT MODULE

- A client module runs in each client computer
- Integrates and extends the operations of the flat file service and the directory service
- A single application programming interface
  - For example, in UNIX hosts, a client module would be provided that emulates the full set of UNIX file operations, interpreting UNIX multi-part file names by iterative requests to the directory service.
- Holds information about the network locations of the flat file server and directory server processes
- Implements of a cache of recently used file blocks at the client
- Play an important role in achieving satisfactory performance

FLAT FILE SERVICE INTERFACE

- This is the RPC interface used by client modules.
- It is not normally used directly by user-level programs.

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).

- A ***FileId*** *is invalid if*
  - *the file that it refers to is not present in the* server processing the request
  - its access permissions are inappropriate for the operation requested

- All of the procedures in the interface except '*Create' throw* exceptions

- The most important operations are those for reading and writing

- Both the ***Read*** **and the** ***Write*** *operation require a parameter* ***i*** *specifying a* ***position in the file****.*
  - *The Read* operation copies the sequence of *n data items beginning at item i from the specified file* into *Data, which is then returned to the client.*

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

- *Create: creates a new, empty file and returns the UFID that is generated*

- *Delete:* removes the specified file

- *GetAttributes and SetAttributes: enable clients to access the attribute record*

- *GetAttributes is normally available to any client that is allowed to read the file*

- *Access* to the *SetAttributes operation:*
  - *restricted to the directory service that* provides access to the file

- Values of length and timestamp portions of attribute record
  - Not affected by *SetAttributes*
  - *Maintained separately by the* flat file service itself.

Comparison with UNIX:

- This interface and the UNIX file system primitives are functionally equivalent.
- For this interface:
  - **no *open and close*** operations – files accessed immediately by quoting the appropriate UFID.
  - ***Read and Write*** *requests in this interface* ***include a parameter*** *specifying a starting point* within the file for each transfer, whereas the equivalent UNIX operations do not
  - ***seek operation*** *is provided to enable the read-write pointer to be* explicitly repositioned

The interface differs mainly for reasons of **fault tolerance**:

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

- *Stateless servers:*
  - *Suitable for implementation by stateless servers*
  - Stateless servers can be restarted after a failure easily
  - No need for clients or the server to restore any state

UNIX FILE OPERATIONS

- Neither idempotent nor consistent with the requirement for a stateless implementation.

- The UNIX *read and write operations are not idempotent*

- File poniter is created and retained till file is closed.

- If an operation is accidentally repeated, the automatic advance of the read-write pointer

- The read-write pointer is a hidden, client-related state variable

Access Control

- **In the UNIX file system**
  - the user's access rights are checked against the access *mode (read or write) requested*
  - The user identity (UID) used in the access rights check
  - UID retrieved during the user's earlier authenticated login
  - The resulting access rights are retained until the file is closed
  - no further checks are required for subsequent operations on the same file

- **In distributed implementations**
  - server RPC interface is an otherwise unprotected point of access to files
  - access rights checks have to be performed at the server
  - A user identity passed with requests & server is vulnerable to forged identities
  - Stateless server should not store access rights check results for future accesses

Alternative approaches:

1. **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**

2. **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
- Both have been used in distributed file systems

- The second is more common; it is used in both NFS and AFS

- Neither overcomes the security problem concerning forged user identities

- This can be addressed by the use of digital signatures

- Kerberos is an effective authentication scheme that has been applied to both NFS and AFS.

- The user identity is passed as an implicit parameter and can be used whenever it is needed

DIRECTORY SERVICE INTERFACE

- The primary purpose : a service for translating text names to UFIDs
- 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

DIRECTORY SERVICE OPERTIONS

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*.

- Operations are performed on individual directories
- For each operation, a UFID for the file containing the directory is required (in the *Dir parameter)*
- ***The Lookup*** operation in the basic directory service performs a single *Name -> UFID translation.*
- Exceptions caused by inadequate access rights are omitted from the definitions.

There are two operations for altering directories: *AddName and UnName.*

- ***AddName***
  - *adds an entry to a directory and increments the reference count field in the* file's attribute record.
- ***UnName***
  - *removes an entry from a directory and decrements the reference count.*
  - If this causes the reference count to reach zero, the file is removed.
- ***GetNames*** *is* provided to enable clients to
  - examine the contents of directories
  - **implement pattern-matching** operations on file names (such as in the UNIX)

Hierarchic file system

- **A hierarchic file system such as the one that UNIX**
- A number of directories arranged in a tree structure
- Each directory holds the names of the files and other directories
- Any file or directory can be referenced using a ***pathname***
  - *a multi-part name that represents a path through* the tree
- The root has a distinguished name
- 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
  - implemented by a ***link operation****, which adds a new name for a file to a specified directory*

- A UNIX-like file-naming system can be implemented by the client module

- **The file attributes associated with files should include a type field**
  - **that distinguishes between ordinary files and directories**

- Used when following a path to ensure that each part of the name, except the last, refers to a directory

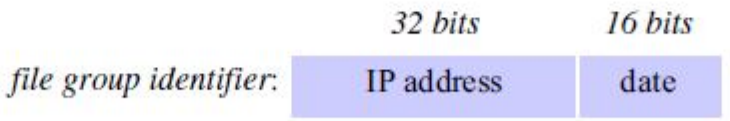
File groups

- **A *file group is a collection of files located on a given server.***
  - ***A server may*** hold several file groups
  - 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** & other OSs
  - ***filesystem*** *refers to the set of files held in a storage device or partition*
  - ***file system*** *refer to a software component that provides access to files*

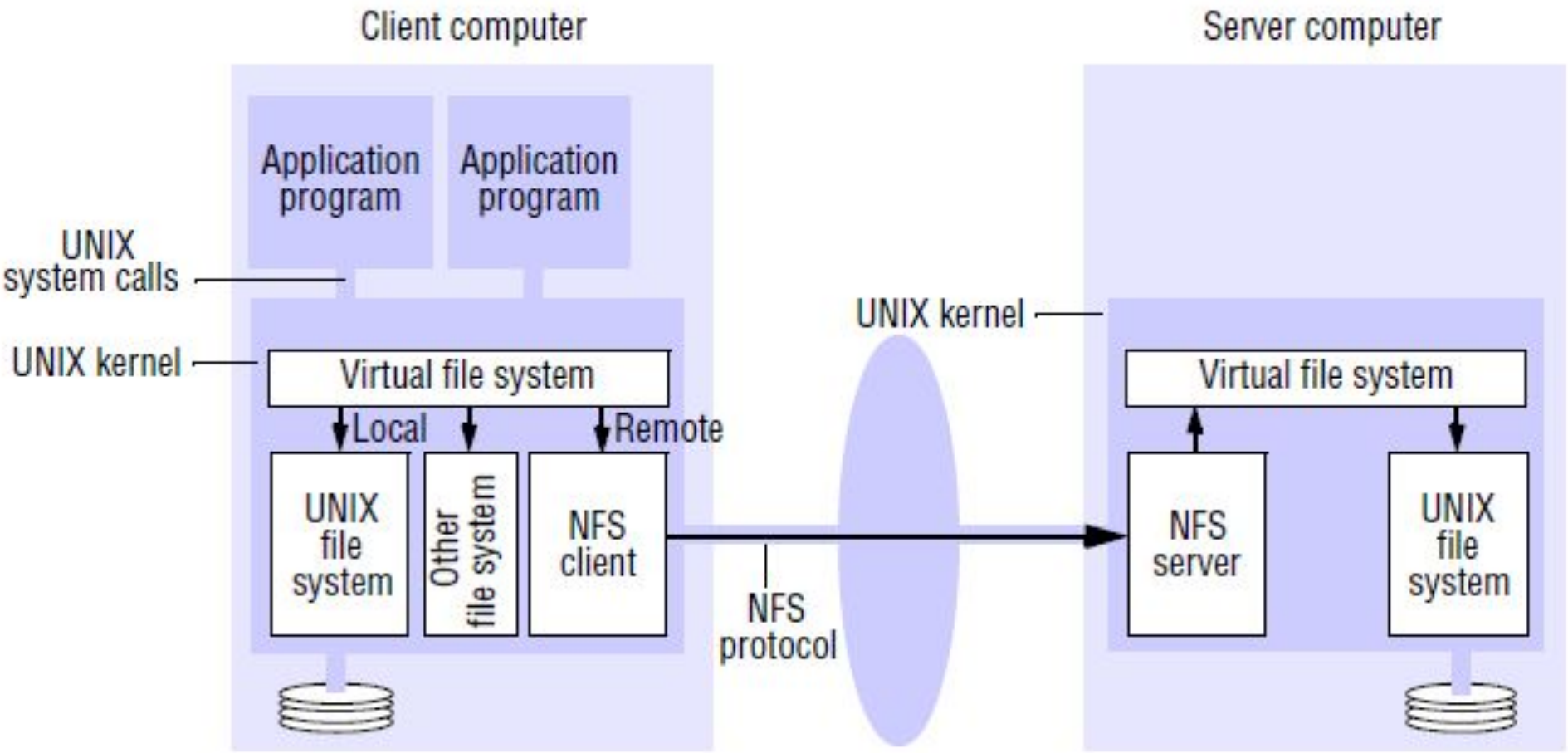
- *File groups*
  - originally introduced for moving collections of files stored on removable media between computers
  - support the allocation of files to file servers in larger logical units
  - enable the service to be implemented with files stored on several servers

- UFIDs includes a file group identifier component for DS with file groups
- File group identifiers must be unique throughout a distributed system
- generate them with an algorithm that ensures global uniqueness
- For example, whenever a new file group is created, a unique identifier can be generated as shown:



- a mapping between group identifiers and servers should be maintained by the file service.

NETWORK FILE SYSTEM



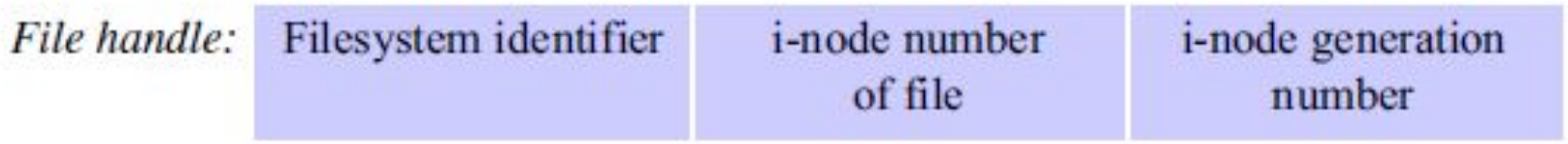
- It follows the abstract model defined.
- All implementations of NFS support the NFS protocol
- **NFS Protocol: a set of remote procedure calls** that provide the means for clients to perform operations on a remote file store
- The NFS protocol is operating system–independent
- The *NFS server module 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
  - passed to the NFS server module
- NFS client and server modules communicate using RPCs

- NFS is compatible with both TCP and UDP
- port mapper 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
  - valid requests with valid user credentials will be acted upon
- Optional Security Feature:
  - The submission of signed user credentials
  - encryption of data for privacy and integrity

Virtual file system (VFS)

- VFS module provides access transparency
- It distinguishes between local and remote files
- Translates the
  - UNIX-independent file identifiers used by NFS
  - internal file identifiers normally used in UNIX and other file systems
- Keeps track of the filesystems that are currently available both locally and remotely
- Passes each request to the appropriate local system module
  - the UNIX file system
  - the NFS client module
  - service module for another file system

- The file identifiers used in NFS are called *file handles*
- *A file handle* contains information to distinguish 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*



  - *i-node number of a UNIX file is a* number that serves to identify and locate the file within the file system

- NFS adopts the UNIX mountable filesystem as the unit of file grouping

- The *filesystem identifier field is*
  - *a unique number*
  - *allocated to* each filesystem when it is created

- The *i-node generation number*
  - i-node numbers are reused after a file is removed
  - a generation number is stored with each file and is
  - incremented each time the i-node number is reused

- The client obtains the first file handle for a remote file system when it mounts it.

- File handles are passed
  - from server to client in the results of *lookup, create* and *mkdir operations*
  - *from client to server in the argument lists of* all server operations

- The virtual file system layer has
  - one VFS structure for each mounted file system
  - one *v-node per open file*

- *A VFS structure*

  relates a remote file system to the local directory on which it is mounted

- v-node contains an indicator to show if 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

Client integration

- NFS client module provides an interface suitable for use by conventional application programs

- But unlike the model client module
  - it emulates the semantics of the standard UNIX file system primitives precisely
  - is integrated with the UNIX kernel

- It is integrated with the kernel and not supplied as a library for loading into client processes so that:
  - user programs can access files via UNIX system calls without recompilation or reloading
  - a single client module serves all of the user-level processes
  - the encryption key used to authenticate user IDs can be retained in the kernel, preventing impersonation by user-level clients.

- NFS client module cooperates with VFS in each client machine
- Operates similar to the conventional UNIX file system
  - transferring blocks of files to and from the server
  - caching the blocks in the local memory whenever possible
- Shares the same buffer cache used by the local I/O system
- a new and significant cache consistency problem arises
  - Since several clients in different host machines may simultaneously access the same remote file

Access control and authentication

- **NFS** server is stateless: doesn't keep files open on behalf of its clients

- Checks the user's identity against the file's access permission attributes afresh on each request

- Eg. The Sun RPC protocol requires clients to send user authentication information with each request

- These additional parameters are supplied automatically by the RPC system.

- In its simplest form, there is a security loophole in this access-control mechanism.
  - An NFS server provides RPC interface at a well-known port
  - Any process can behave as a client, sending requests to the server
  - Impersonating the user without their knowledge or permission
- Loophole closed:
  - **DES encryption** of the user's authentication information
  - **Kerberos** has been integrated with Sun NFS to provide a stronger and more comprehensive solution to the problems of user authentication and security

NFS SERVER INTERFACE

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

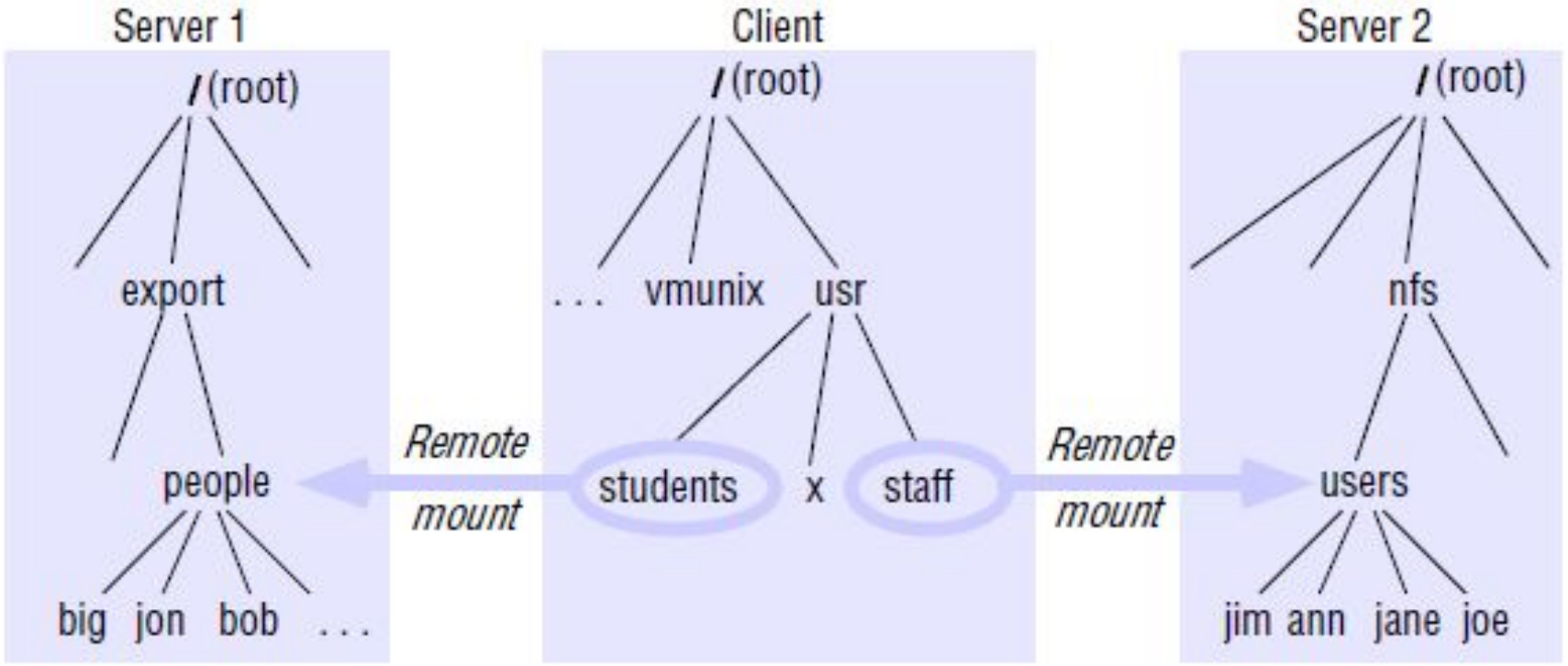
lookup(dirfh, name) → *fh, attr*	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) → *newfh, attr*	Creates a new file *name* in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) → *status*	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*.

Mount service

- The mounting of subtrees of remote filesystems by clients
- *Runs at user level on each NFS server computer*
- Each server has a file with a well-known name (*/etc/exports)*
  - *contains* names of local filesystems that are available for remote mounting
- An access list is associated with each filesystem name
- ***To request mounting*** of a remote filesystem:
  - Clients use a modified version of the UNIX ***mount command***
  - specify the
    - **remote host's name**
    - **the pathname of a directory in the remote filesystem**
    - **the local name** with which it is to be mounted
- Clients can mount any part of the remote filesystem

- The modified *mount command communicates* with the mount service process on the remote host using a *mount protocol*

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.

- Remote filesystems may be *hard-mounted or soft-mounted in a client computer*

- *Hard Mounted :*
  - The process is suspended until the request can be completed
  - if the remote host is unavailable for any reason the NFS client module continues to retry the request until it is satisfied
  - In case of server failure user-level processes are suspended until the server restarts

- Soft-mounted:
  - NFS client module returns a failure indication to user-level processes after a small number of retries
  - Properly constructed programs detect failure and take appropriate recovery or reporting actions

- But many UNIX utilities and applications do not test for the failure of file access operations

Pathname translation:

- In NFS, pathnames cannot be translated at a server
- The name may cross a ‘mount point’ at the client

  (directories holding different parts of a multi-part name may reside in filesystems at different servers)
- Pathnames are parsed & 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
- It uses a separate *lookup request* to the remote server

AUTOMOUNTER

- The automounter maintains a table of mount points (pathnames)
- It behaves like a local NFS server at the client machine
- When NFS client module resolves a pathname with mount points
  - it passes to the local automounter a *lookup() request*
  - it locates the required filesystem in its table
  - sends a 'probe' request to each server listed
- The filesystem is then mounted at the client using the normal mount service
- The mounted filesystem is linked to the mount point using a symbolic link
- File access then proceeds in the normal way
- If there are no references to the symbolic link for several minutes, automounter unmounts the remote filesystem

SERVER CACHING

Conventionally:

- ***Read-ahead*** anticipates read accesses and fetches the pages following those that have most recently been read
- ***Delayed-write optimizes writes****: altered* contents are written to disk only when the buffer is required for another page
- UNIX *sync operation* flushes altered pages to disk every 30 seconds

NFS servers:

- Server's cache holding recently read disk blocks: no consistency problem

- Server performs write operations: extra measures to ensure results of the write operations are persistent (even when server crashes occur)

- The *write operation offers two options in NFS version 3*
  - *Write through cache- write to disk before sending respone*
  - *Write to disk after commit instruction*

- ***Commit***
  - *additional operation provided in version 3 of the NFS protocol;*
  - overcomes a performance bottleneck caused by the write-through mode of operation

CLIENT CACHING:

- The NFS client module caches the results of *read, write, getattr, lookup and readdir operations*

- Different versions of files or portions of files may exist in different client nodes

- Clients are responsible for polling the server to check the currency of the cached data that they hold

- A timestamp-based method is used to validate cached blocks

Each data or metadata item in the cache is tagged with two timestamps:

- ***Tc*** *: time when the cache entry was last validated*
- ***Tm :*** *time when the block was last modified at the server*

- A cache entry is valid at time *T if*
  - *T – Tc is less than a freshness interval t,*
  - *if the value* for *Tm recorded at the client matches the value of Tm at the server*

$$(T - Tc < t) \vee (Tm_{client} = Tm_{server})$$

- There is one value of *Tmserver for all the data blocks in a file and another for the* file attributes.

- NFS clients cannot determine **whether a file is being shared or not**
- Validation procedure must **be used for all file accesses**
- A validity check is performed whenever a cache entry is used
- First half of validity condition can be evaluated without access to server
- If true, then the second half need not be valuated
- If false, current value of *Tmserver is obtained*
  (by means of a getattr call to the server)

- Measures used to reduce the traffic of *getattr calls to server:*
  - **Whenever a new value of *Tmserver 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
  - **The adaptive algorithm for setting freshness interval *t outlined above reduces the*** traffic considerably for most files
- Level of consistency not same as in conventional UNIX systems
- Recent updates are not always visible to clients sharing a file

WRITES ARE HANDLED DIFFERENTLY

- When a cached **page is modified** it is marked as **‘dirty**’
- Its scheduled to be flushed to the server asynchronously
- **Modified pages** flushed when the file is **closed** or a ***sync occurs*** *at client*
- *They are flushed more* frequently if **bio-daemons** are in use
- Does not provide the same persistence guarantee as the server cache
- But it emulates the behaviour for local writes

- B*io–daemon processes by NFS at each client*
  - *Bio stands* for block input-output
  - *Daemon is often used to refer to user-level processes that* perform system tasks
  - perform read-ahead and delayed-write operations

- Bio-daemon is notified after each **read request** , it requests **transfer of the following file** block from server to client cache

- In the case of **writing**, the bio-daemon will **send a block to the server** whenever a block has been **filled by a client operation**.

- Directory blocks are sent whenever a modification has occurred.

OTHER OPTIMIZATIONS

- In NFS **version 3**, there is **no limit on the maximum size** of file blocks that can be handled in *read and write operations*

- *Clients* and servers can **negotiate sizes larger than 8 kbytes** if both are able to handle them.

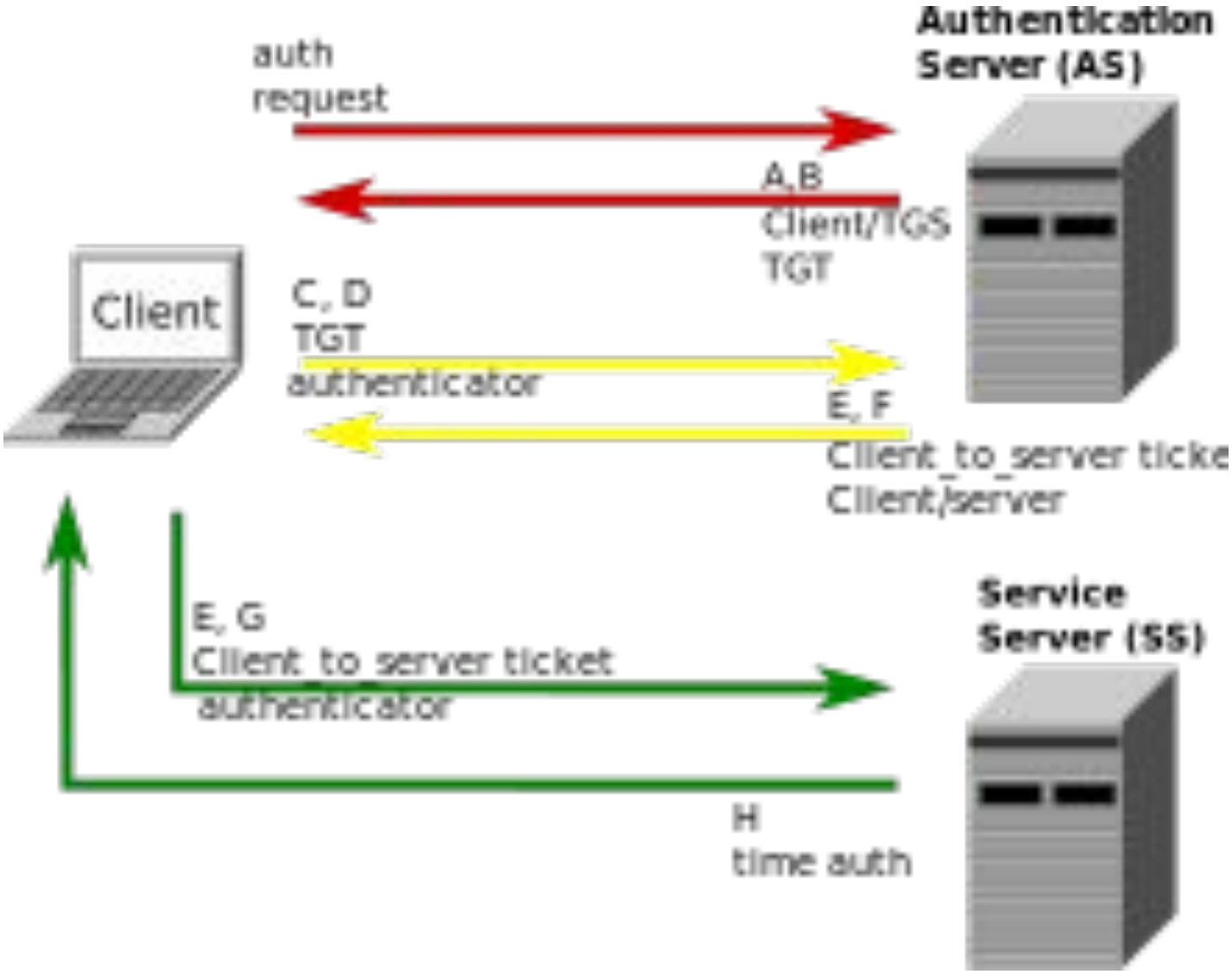
- **File status information** cached at clients must be **updated at least every three seconds for active files**.

- ***All operations*** *that refer to files or directories are* ***taken*** **as implicit** ***getattr requests***

- ***Current attribute values are 'piggybacked'*** *along with* the other results of the operation

KERBEROS

NETWORK AUTHENTICATION WITH TICKETS



PERFORMANCE

Early performance figures reported :

- No performance penalty in comparison with access to files stored on local disks.
- 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

Later:

- Commit introduced
- Frequency of getattr reduced
- NFS offers a very effective solution to distributed storage needs in intranets of most sizes and types of use

ANDREW FILE SYSTEM

- AFS provides transparent access to remote shared files
- Access to AFS files is via the normal UNIX file primitives
- enabling existing UNIX programs to access AFS files without modification or recompilation
- AFS is compatible with NFS
- AFS servers hold ‘local’ UNIX files
- Filing system in the servers is NFS-based
- Files are referenced by NFS-style file handles
- Files may be remotely accessed via NFS

- AFS differs markedly from NFS in design and implementation
- Identification of scalability as the most important design goal
- 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

    (files larger than 64 kbytes are transferred in 64-kbyte chunks)
  - ***Whole-file caching****: File/chunk* is stored in a cache on the local disk

- The cache contains several hundred of the files

- The cache is permanent, surviving reboots of the client computer

- Local copies of files are used to satisfy clients' *open requests*

Scenario

Here is a simple scenario illustrating the operation of AFS:

i. Client computer issues an ***open system call*** *for a file in*
   - **i.** **Not a current copy of the file in the local cache**
   - **ii.** **send a request to server** for a copy of the file
   - **iii.** **Copy is stored** in the local UNIX file system in the **client computer**
   - iv. Copy is *opened and the resulting UNIX file descriptor is returned*
   - **v.** **Subsequent *read, write*** *and other operations* **applied to the local copy**

ii. Client **issues a *close system call***
   - *i.* *If local copy has been* **updated** its contents are **sent back to the server**
   - ii. The server updates the file contents and the timestamps on the file
   - **iii.** **Copy on the client's local disk is retained** in case it is needed again

Observations:

Locally cached copies are likely to remain **valid for long periods**

- **For shared files that are infrequently updated**
  - (such as those containing the code of UNIX commands and libraries)
- for files that are normally **accessed by only a single user**
  - (such as most of the files in a user's home directory and its subtree),

- **The local cache** allocated a substantial proportion of the disk space
  (say, **100 megabytes**)

- Normally sufficient for the establishment of a working set of the files

- Ensures that files in regular use are normally retained in the cache until they are needed again

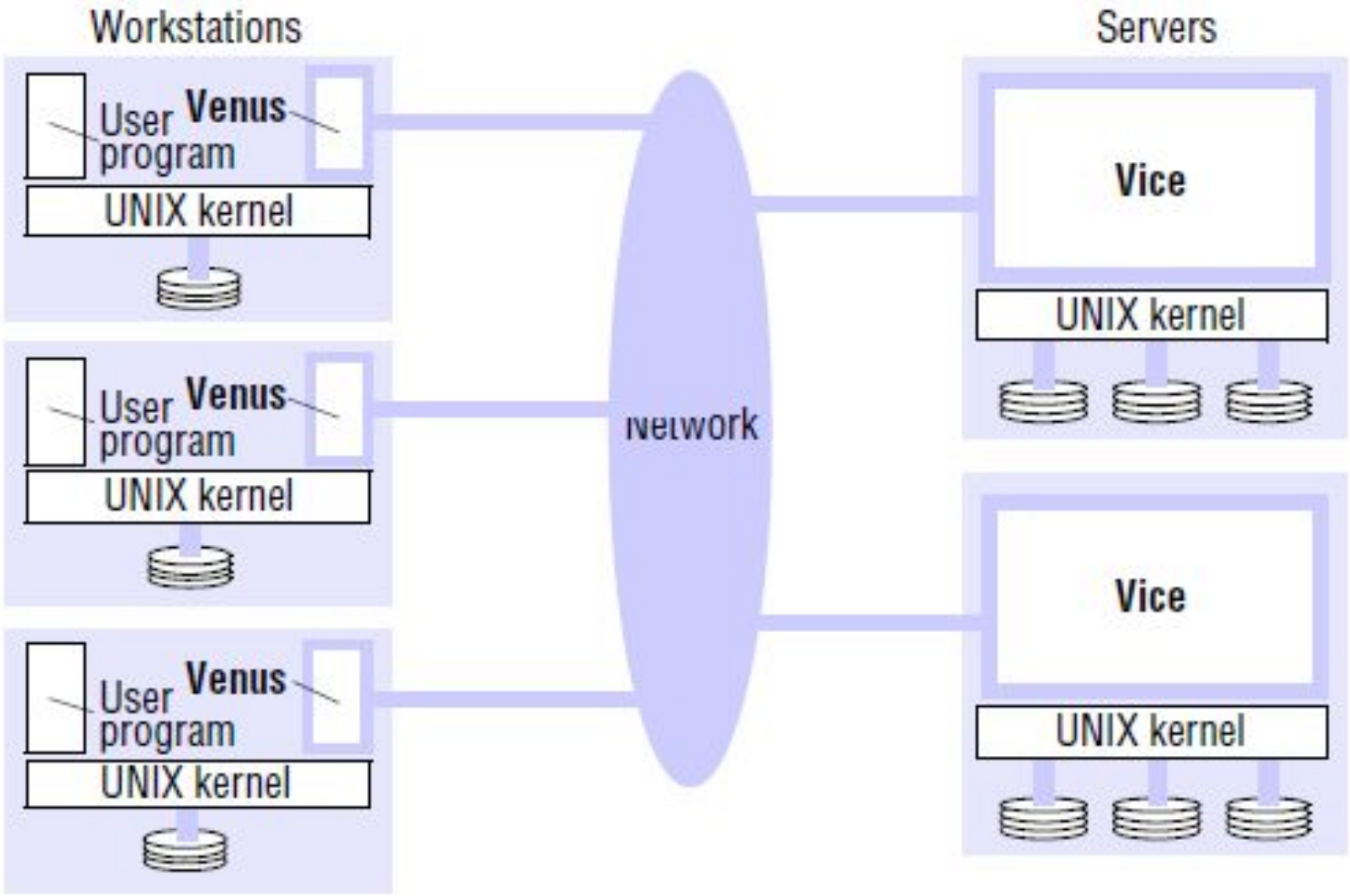
- The design strategy is based on some **assumptions:**
  - Files are small; most are less than 10 kilobytes in size.
  - **Read** operations on files are **much more common** than writes
    (about 6 times more common)
  - **Sequential access is common**, and random access is rare
  - Most files are read and written by **only one user**
  - When a file is shared, it is usually only one user who modifies it
  - Files are **referenced in bursts**

- AFS works best with small files

- **Databases** : type of file **that does not fit into any of these classes**
  - typically shared by many users
  - often updated quite frequently

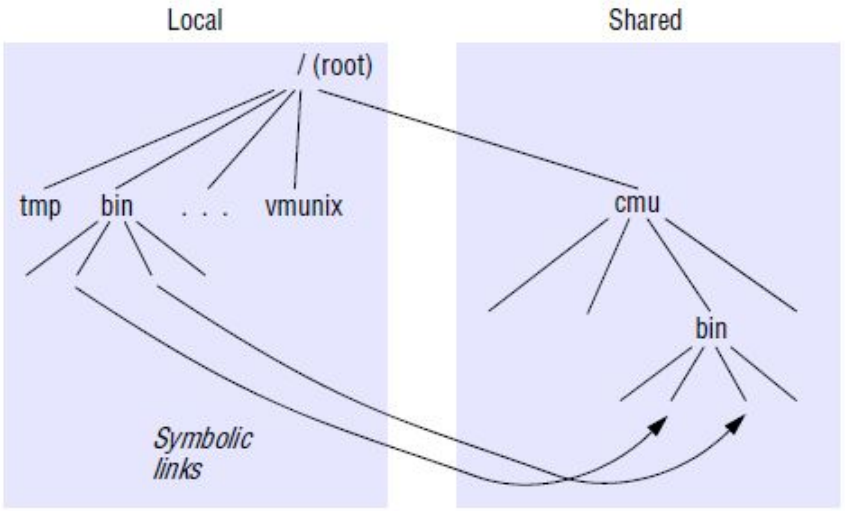
- The designers of AFS have explicitly excluded the provision of storage facilities for databases from their design goals

Distribution of processes in the Andrew File System



- AFS is implemented as two software components : *Vice and Venus*
  - **Vice** - server software that runs as a user-level UNIX process in each server computer
  - **Venus** - user-level process that runs in each client computer

File name space seen by clients of AFS



- **Local files**
  - handled as **normal UNIX files**
  - Stored on a workstation's disk
  - available only to local user processes
- **Shared files** are
  - stored on **servers**
  - Copies are cached on the local disks of workstations

- Directory hierarchy, with a specific subtree (called *cmu) contains all of the shared files*

- **Some loss of location transparency**
  - *Due to* splitting of the file name space into local and shared files

- Local files are used only for
  - temporary files (*/tmp)*
  - *processes that are essential for* workstation startup

- Other standard UNIX files (found in */bin, /lib and so on)*
  - *implemented as symbolic links from local directories to files held in* the shared space

- **Users' directories** are in the **shared space**,

- Enables users to access their files from any workstation

- The UNIX kernel in each workstation and server is a modified version of BSD UNIX.

- M**odifications** designed to **intercept *file* system calls** which refer to files in the shared name space

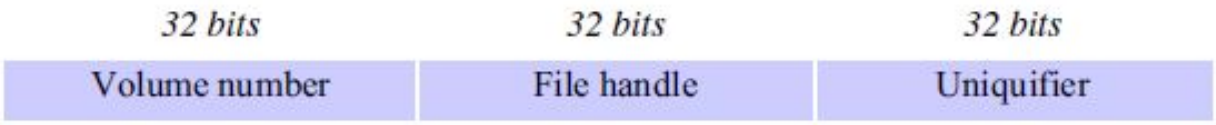
- W**orkstation cache** large enough to **accommodate several hundred average-sized files**

- AFS resembles the abstract file service model described in in these respects:
  - **Flat file service** is implemented by the **Vice** servers
  - Hierarchic directory structure 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*

- Files are grouped into *volumes for ease of location and movement*

- Volumes are generally smaller than the UNIX filesystems

- The representation of ***fids*** *includes the*
  - ***volume number*** *for the volume containing* the file
  - ***NFS file handle*** *identifying the file* within the volume
  - ***uniquifier*** *to ensure that file* identifiers are not reused



- User programs use conventional UNIX pathnames to refer to files
- AFS uses *fids* in the communication between Venus and Vice
- The Vice servers accept requests only in terms of fids
- Venus translates the pathnames supplied by clients into fids

Implementation of file system calls in AFS

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.	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.	→	Transfer a copy of the file and a *callback promise* to the workstation. Log the callback promise.
	Open the local file and return the file descriptor to the application.	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.	←	
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.

CACHE CONSISTENCY

- Vice supplies a copy of a file to a Venus process with a ***callback promise***
  - *It is a token issued by the Vice server that is the custodian of the file*
  - *Guarantee* that it will notify Venus when any other client modifies the file
  - Callback promises are stored with cached files on the workstation disks
  - They have two states: ***valid or cancelled***
- *When a server performs a request to update a file*
  - *It sends a callback*
  - *Sent to all of the* Venus processes to which it has issued callback promises
- A **callback** is a remote procedure call from a server to a Venus process

- When the Venus process **receives a callback**
  - **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
  - If the token is ***valid,*** then the **cached copy can be opened** and used

WORKSTATION IS RESTARTED AFTER A FAILURE OR A SHUTDOWN

- Cannot assume that the callback promise tokens are correct
- Venus generates a cache **validation request**
  - contains the file modification timestamp to the server that is the custodian of the file
- If the timestamp is current
  - the server responds with *valid*
  - the token is reinstated.
- If the timestamp shows that the file is out of date, then
  - the server responds with *cancelled*
  - *the token is set to cancelled*

- *If a time T has elapsed since the* file was cached without communication from the server
  - Callbacks must be renewed before an *open*
  - *T is typically on the order of a few minutes*
- Deals with loss of callback messages due to communication failures
- Callback-based mechanism was adopted as the most scalable approach
- **AFS-1 : timestamp requested for every open**
- **AFS-2 : callback based**

MAIN COMPONENTS OF VICE INTERFACE

Fetch(fid) → attr, data	Returns the attributes (status) and, optionally, the contents of the file identified by *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 the server that a Venus process has flushed a file from its cache.
BreakCallback(fid)	Call made by a Vice server to a Venus process; cancels the callback promise on the relevant file.

UPDATE SEMANTICS

For AFS-1, the update semantics stated in very simple terms.

- For a **client *C*** *operating on a* ***file F*** *whose custodian is a* ***server S***

- *Following guarantees* of currency for the copies of *F are maintained:*
  - after a successful *open: latest(F, S)*
  - after a failed *open: failure(S)*
  - after a successful *close: updated(F, S)*
  - after a failed *close: failure(S)*

- *latest(F, S): the current value of F at C is the same as the* value at *S*

- *failure(S): open or close operation has not been performed at S*
  (and the failure can be detected by C)

- *updated(F, S): C's value of F* has been successfully propagated to *S*

For AFS-2, the currency guarantee for *open is slightly weaker*

- We have the following guarantee after a successful *open:*
  - *latest(F, S, 0)*

  Or
  - *lostCallback(S, T) and inCache(F) and latest(F, S, T)*

- ***T*** *(usually 10 mins): max time for which a client remains unaware of a newer version* of a file

- *latest(F, S, T) : copy of F seen by client is no more than T* seconds out of date

- *lostCallback(S, T) : a callback message from S to C has* been lost at some time during the last *T seconds*

- *inCache(F) : the file F was in the cache at C before the open operation was attempted*

- Formal statement of the guarantee is more complex since
  - client may open an old copy of a file after it has been updated by another client
  - Occurs if a callback message is lost, for example as a result of a network failure

- Cache consistency algo comes into action only on *open and close*

- When the file is closed
  - a copy is returned to the server
  - replaces the current version

- If clients in different workstations *open, write and close the same file* concurrently:
  - **all but the update resulting from the last *close will be silently lost***

    (no error report is given)
  - Clients must implement concurrency control independently if they require it

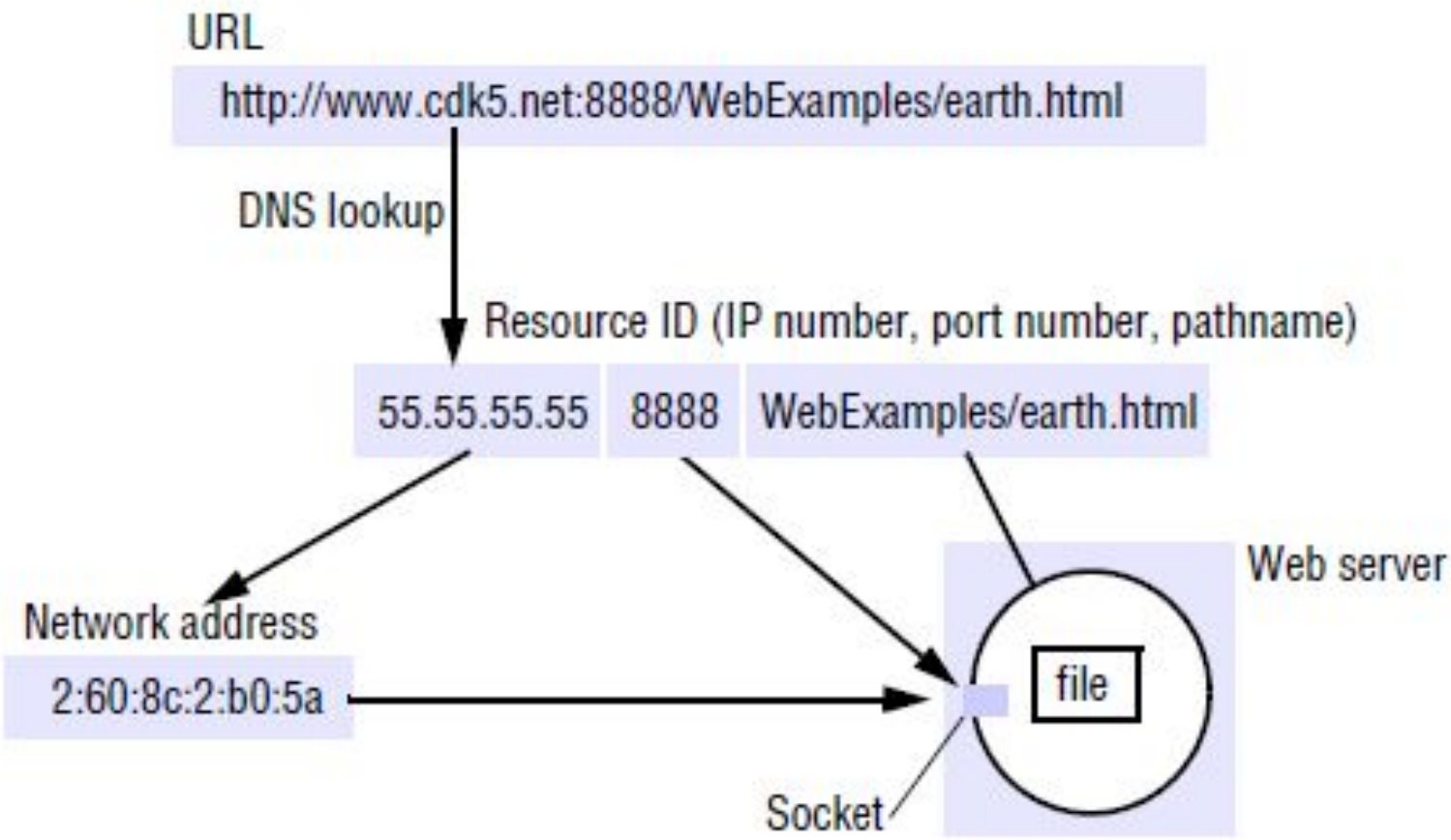
OTHER MODIFICATIONS

- **UNIX kernel modifications:** file handles instead of the conventional UNIX file descriptors
- **Location database**: Each server contains a copy of a fully replicated location database giving a mapping of volume names to servers
- **Threads** : for **concurrent** processing at both the client and the server
- **Read-only replicas** : Volumes containing files that are frequently read but rarely modified replicated as read-only volumes at several servers
- **Bulk transfers** : transfers files between clients and servers in 64-kilobyte chunks
- **Partial file caching**
- **Performance** : Scalability, performance with large numbers of users
- **Wide area support**

NAME SERVICE

- Any process that requires access to a specific resource must possess a **name** or an identifier for it. Examples of human-readable names are file names such as */etc/passwd,* URLs such as *http://www.cdk5.net/ and Internet domain names such as www.cdk5.net.*
- The term ***identifier*** *is sometimes used to refer to names that are interpreted only by* programs. Remote object references and NFS file handles are examples of identifiers.
- Identifiers are chosen for the efficiency with which they can be looked up and stored by software.
- Pure name and non-pure name
- ***Address****: a value that identifies* the location of the object rather than the object itself. Addresses are efficient for accessing objects, but objects can sometimes be relocated, so addresses are inadequate as a means of identification.
- 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.*

Composed naming domains used to access a resource from a URL



Names and services

- Client may use a service-specific name when requesting a service to perform an operation upon a named object or resource that it manages.
- Names are also sometimes needed to refer to entities in a distributed system that are beyond the scope of any single service(email address).

Uniform Resource Identifiers

- identify resources on the Web, and other Internet
- 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

Uniform Resource Locators

- Some URIs contain information that can be used to locate and access a resource; others are pure resource names.
- The familiar term *Uniform Resource Locator (URL) is often used for URIs that provide location information and* specify the method for accessing the resource, including the 'http' URLs

  For example, the URL http://example.org/wiki/Main_Page refers to a resource identified as /wiki/Main_Page whose representation, in the form of HTML and related code, is obtainable via the Hypertext Transfer Protocol (*http:*) from a network host whose domain name is example.org.

- Uniform Resource Names: *Uniform Resource Names (URNs) are URIs that are used as* pure resource names rather than locators

  For example, in the International Standard Book Number (ISBN) system, *ISBN 0-486-27557-4* identifies a specific edition of Shakespeare's play *Romeo and Juliet*. The URN for that edition would be *urn:isbn:0-486-27557-4*.

Name services and the Domain Name System

- A *name service stores information about a collection of textual names, in the form of* bindings between the names and the attributes of the entities they denote, such as users, computers, services and objects.

- The collection is often subdivided into one or more naming *contexts: individual subsets of the bindings that are managed as a unit.*

- *The* major operation that a name service supports is to resolve a name – that is, to look up attributes from a given name.

- Operations are also required for creating new bindings, deleting bindings and listing bound names, and adding and deleting contexts.

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

- ***Integration****: It is not always possible to predict the scope of sharing in a distributed* system. It may become necessary to share and therefore name resources that were created in different administrative domains.

- Without a common name service, the administrative domains may use entirely different naming conventions

General name service requirements

- Grapevine [*1982] was one of the earliest extensible, multi-domain* name services

- The Global Name Service, developed at the Digital Equipment Corporation Systems Research Center [1986], is a descendant of Grapevine with ambitious Goals, including:
  - *To handle an essentially arbitrary number of names and to serve an arbitrary number of administrative organizations*
  - *A long lifetime*
  - *High availability*
  - *Fault isolation*
  - *Tolerance of mistrust*

NAME SPACES

- A *name space is the collection of all valid names recognized by a particular service. The* service will attempt to look up a valid name, even though that name may prove not to correspond to any object – i.e., to be *unbound.*
- *Name spaces require a syntactic* definition to separate valid names from invalid names.
- 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

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

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***

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.*

Combining and customizing name spaces

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

Merging:

- The practice of mounting file systems in UNIX and NFS provides an example in which a part of one name space is conveniently embedded in another.

Customization:

- In the example of embedding NFS-mounted file systems above that sometimes users prefer to construct their name spaces independently rather than sharing a single name space.

- File system mounting enables users to import files that are stored on servers and shared, while the other names continue to refer to local, unshared files and can be administered autonomously.

- But the same files accessed from two different computers may be mounted at different points and thus have different names.

- Not sharing the entire name space means users must translate names between computers.

Name resolution

- For the common case of hierarchic name spaces, name resolution is an iterative or recursive process whereby a name is repeatedly presented to naming contexts in order to look up the attributes to which it refers.

- A naming context either maps a given name onto a set of primitive attributes (such as those of a user) directly, or maps it onto a further naming context and a derived name to be presented to that context.

- To resolve a name, it is first presented to some initial naming context; resolution iterates as long as further contexts and derived names are output.

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

Name servers and navigation

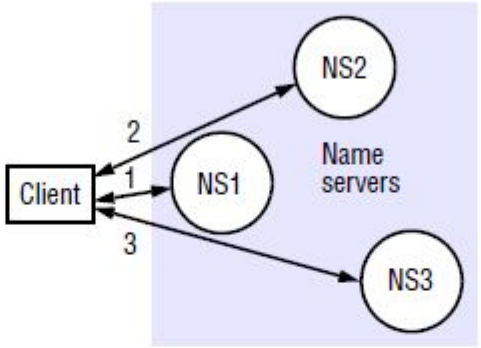
- Any name service, such as DNS, that stores a very large database and is used by a large population will not store all of its naming information on a single server computer. Such a server would be a bottleneck and a critical point of failure.
- Any heavily used name services should use replication to achieve high availability. DNS specifies that each subset of its database is replicated in at least two failure-independent servers.
- Data belonging to a naming domain is usually stored by a local name server managed by the authority responsible for that domain.
- Although, in some cases, a name server may store data for more than one domain, it is generally true to say that data is partitioned into servers according to its domain.
- In DNS, most of the entries are for local computers.
- But there are also name servers for the higher domains, such as *yahoo.com and ac.uk, and for the root.*
- The partitioning of data implies that the local name server cannot answer all enquiries without the help of other name servers. For example, a name server in the *dcs.qmul.ac.uk domain would not be able to supply the IP address of a computer in the* domain *cs.purdue.edu unless it was cached – certainly not the first time it is asked.*

- The process of locating naming data from more than one name server in order to resolve a name is called *navigation. The client name resolution software carries out* navigation on behalf of the client. It communicates with name servers as necessary to resolve a name. It may be provided as library code and linked into clients

Iterative navigation

- 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 thename is located or is discovered to be unbound

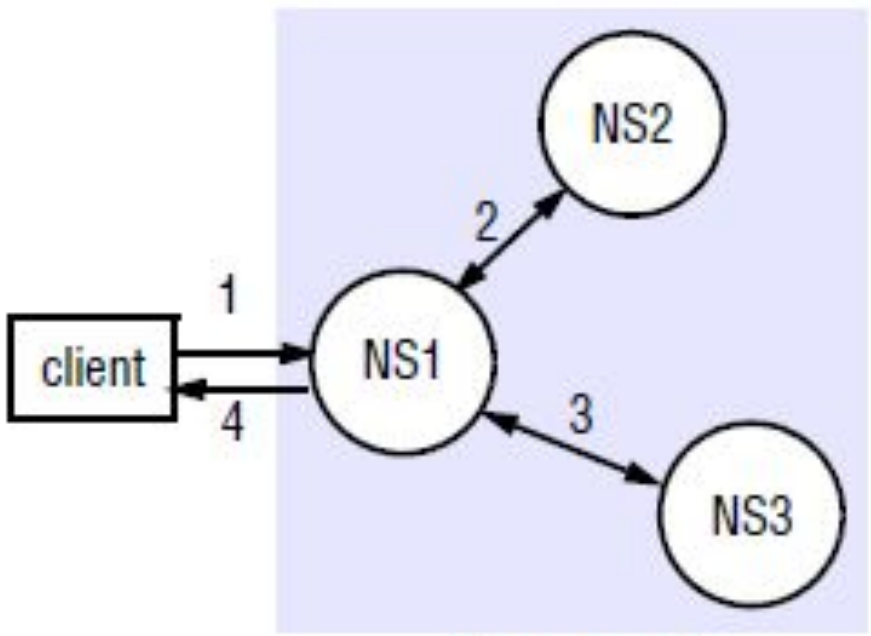
Iterative navigation



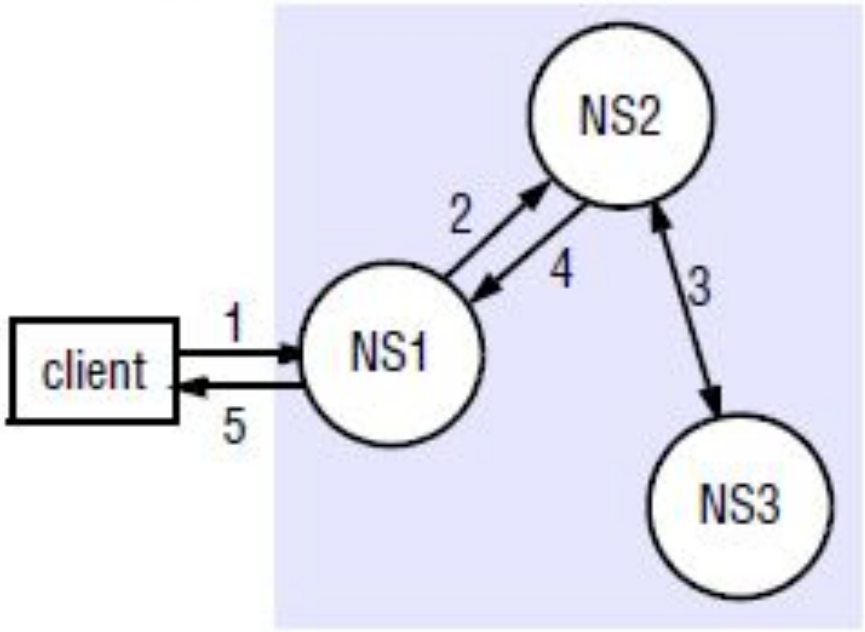
A client iteratively contacts name servers NS1–NS3 in order to resolve a name

- 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.
- a name server coordinates the resolution of the name and passes the result back to the user agent Under non-recursive server-controlled navigation, any name server may be chosen by the client.
- This server communicates by multicast or iteratively with its peers in the style described above, as though it were a client.
- Under recursive server-controlled navigation, the client once more contacts a single server. If this server does not store thename, the server contacts a peer storing a (larger) prefix of the name, which in turnattempts to resolve it. This procedure continues recursively until the name is resolved.

Non-recursive and recursive server-controlled navigation



Non-recursive
server-controlled



Recursive
server-controlled

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

Caching

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