MODULE - V

Distributed file Systems

- Distributed file system:
- File service architecture
- Network file system
- Andrew file system
- Google file system

Distributed file Systems

A file system is a subsystem of the operating system that performs file management activities such as

- organization
- Storing
- retrieval
- Naming
- Sharing
- protection

- A distributed file system(DFS) is a method of storing and accessing files based in a client/server architecture.

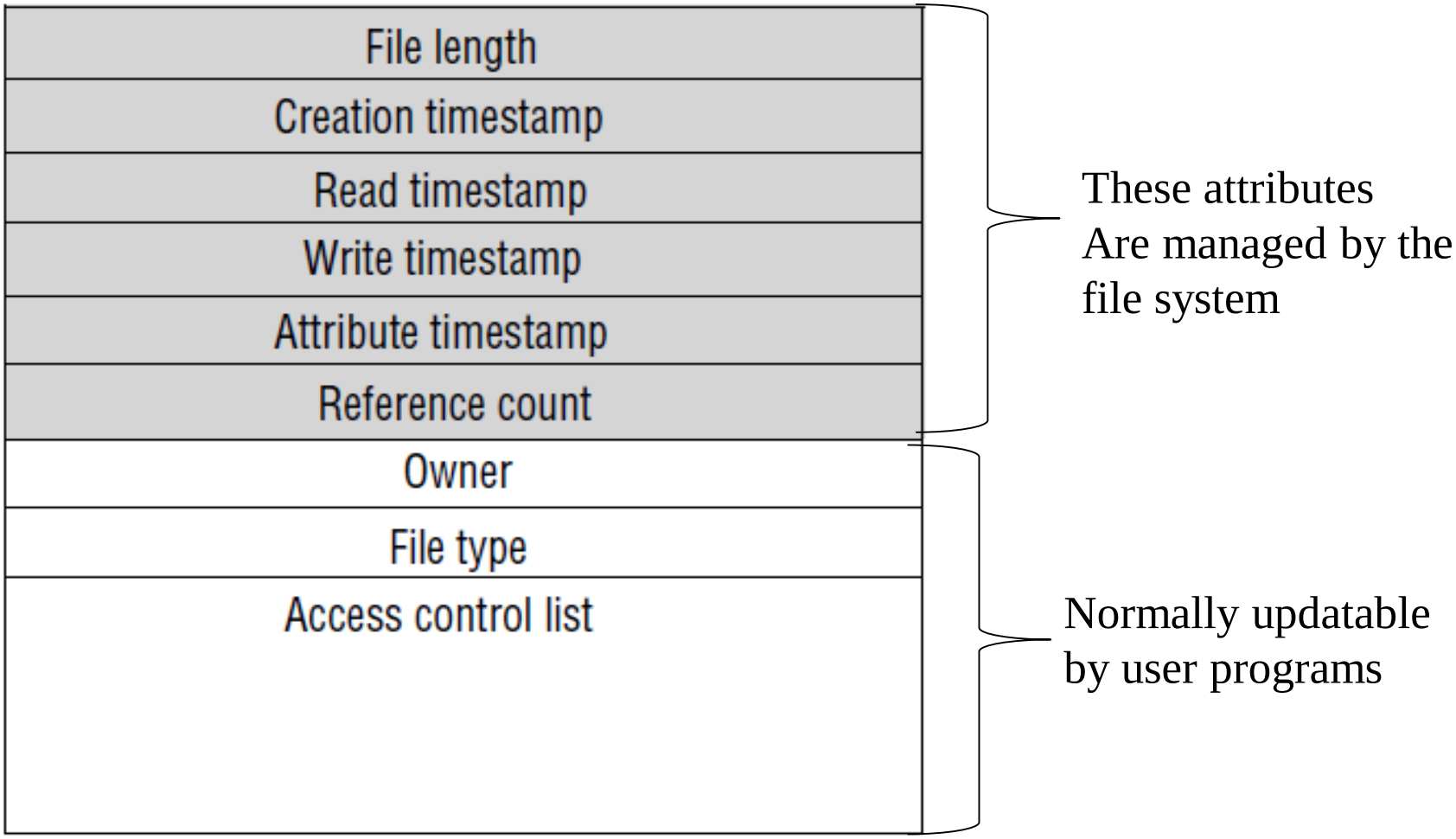
- In a distributed file system, one or more central servers store files that can be accessed, with proper authorization rights, by any number of remote clients in the network.

- Files contain both data and attributes.

- The data consist of a sequence of data items , 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**.

File attribute record structure



Characteristics of file systems

- 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.
- 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.

Fig: 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

- Figure above 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.

Distributed file system requirements

- Transparency
- Concurrent file updates
- Fault tolerance
- File replication
- Hardware and operating system heterogeneity
- Consistency
- Security
- Efficiency

1. Transparency :

Its the concealment from the user and the application programmer of the separation of component in a DS.

- 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.
- Location transparency: enable files to be accessed without knowledge of location
- Mobility transparency: allow the movement of file without affecting the operation of user
- Performance : 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.

2. **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.

3. File replication :

In a file service that supports replication, a file may be represented by several copies of its contents at different locations.

It enhances fault tolerance by enabling clients to locate another server that holds a copy of the file when one has failed.

4.Heterogeneity :

The service interfaces should be defined so that client and server software can be implemented for different operating systems and computers.

5.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.

6. Security:

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.

7.Efficiency:

Provide good level of performance

8. Consistency: If any changes made to one file, that changes must do in other replicated copies.

File Service Architecture

- An architecture that offers a clear separation of the main concerns in providing access to files is obtained by **structuring the file service as three components**:

  - A flat file service
  - A directory service
  - A client module.

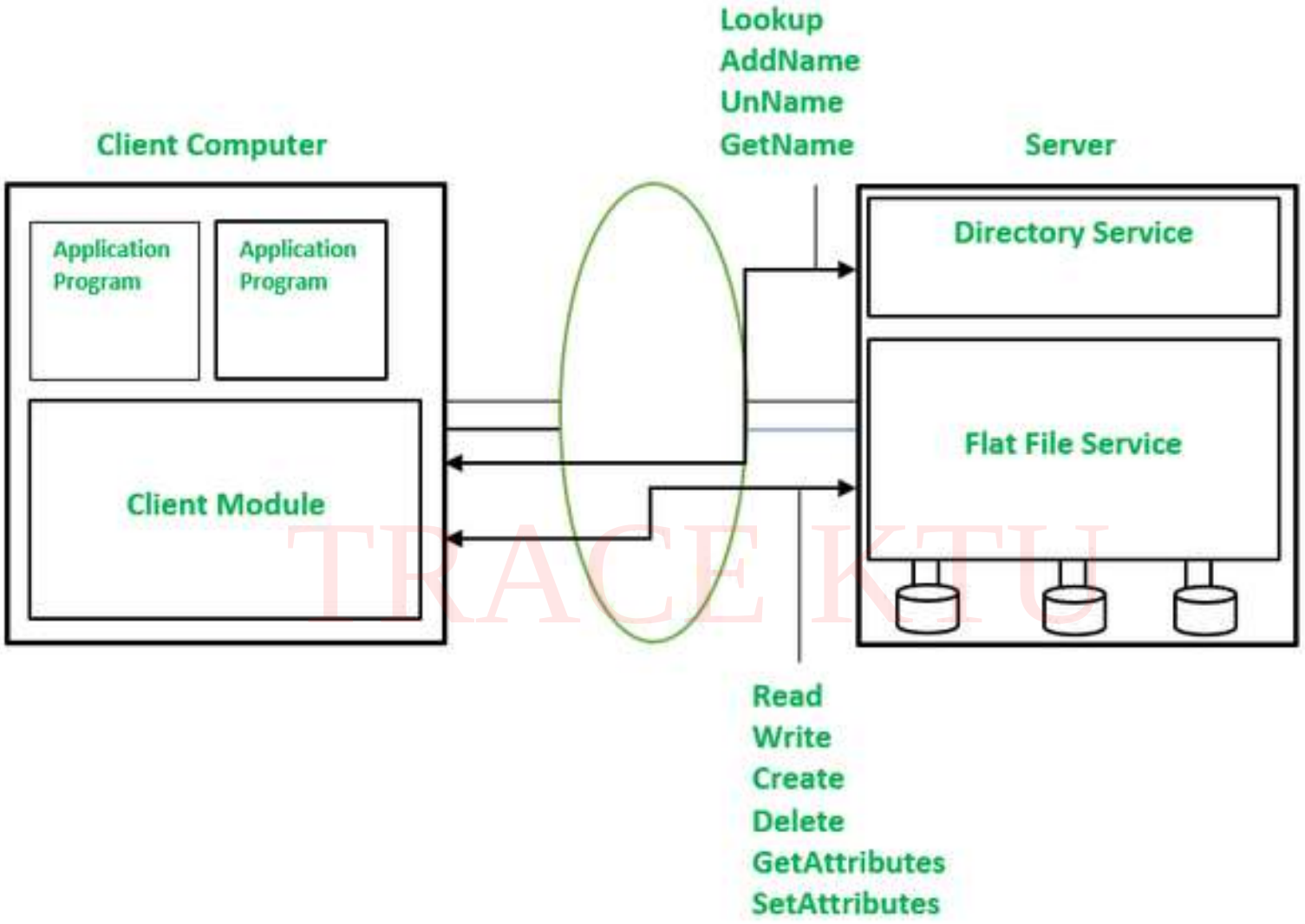


Figure 5. File service architecture

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

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).

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.

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.

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*.

DFS: Case Studies

- NFS (Network File System)
  - Developed by Sun Microsystems (in 1985)
  - Most popular, open, and widely used.

  Client and server communicate using RPC (tcp or udp, support both)
  - Os independent but originally developed for unix
- AFS (Andrew File System)
  - Developed by Carnegie Mellon University as part of Andrew distributed computing environments (in 1986)
  - A research project to create campus wide file system.

NFS (Network File System)

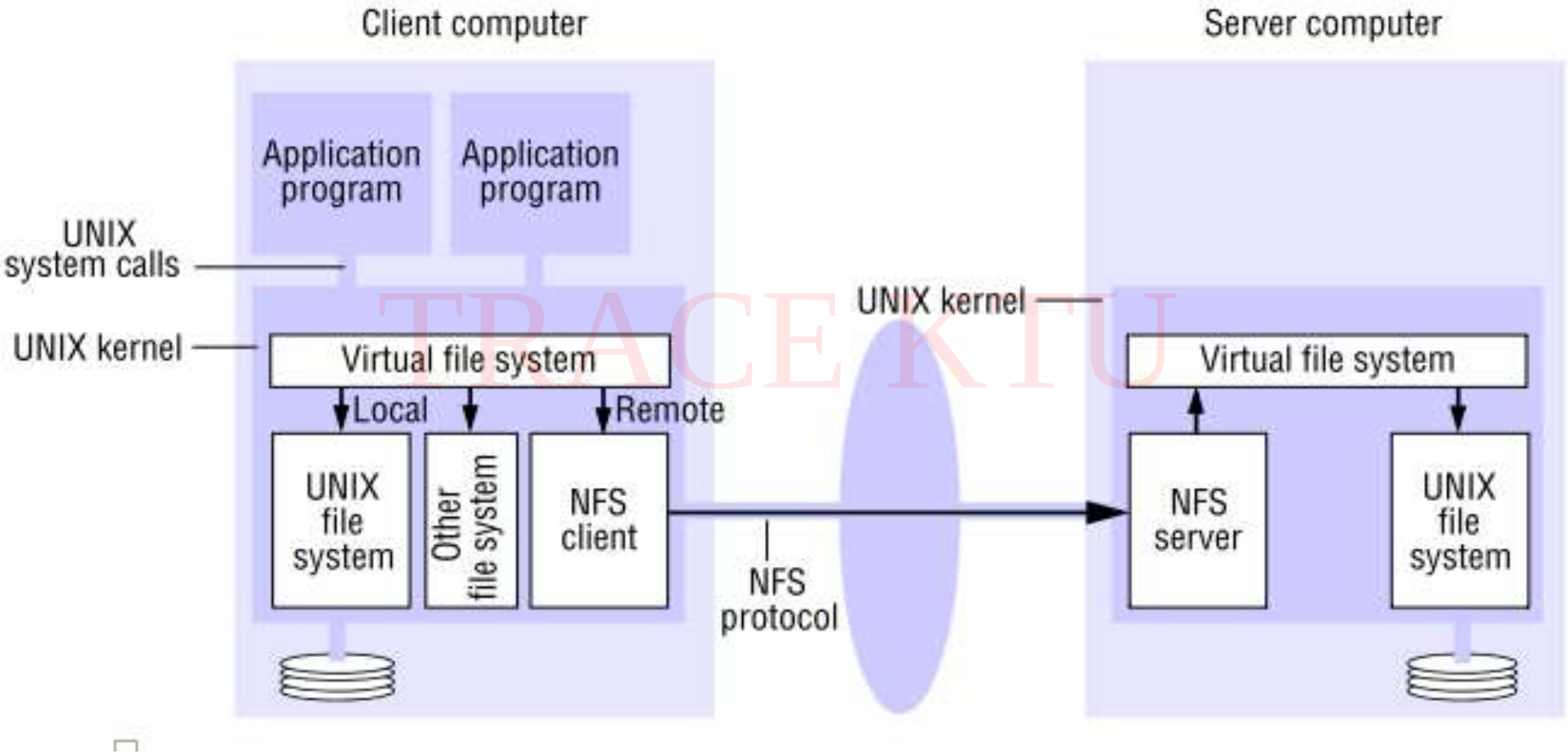
- All of the implementations of NFS support the NFS protocol – a set of remote procedure calls that provide the means for clients to perform operations on a remote file store.

- The NFS protocol is operating system–independent but was originally developed for use in networks of UNIX systems

- 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

NFS architecture



- 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.

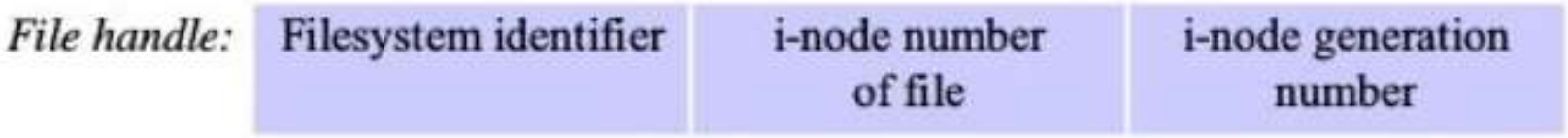
- The RPC interface to the NFS server is open: any process can send requests to an NFS server; if the requests are valid and they include valid user credentials, they will be acted upon.

Virtual File System

- The integration is achieved by a virtual file system (VFS) module, which has been added to the UNIX kernel to distinguish between local and remote files and to translate between the UNIX-independent file identifiers used by NFS and the internal file identifiers normally used in UNIX and other file systems.

- In addition, VFS keeps track of the file systems that are currently available both locally and remotely, and it passes each request to the appropriate local system module

The file identifiers used in NFS are called file handles.



- The **file system identifier** field is a unique number that is allocated to each file system when it is created.
- The **i-node number** is needed to locate the file in file system and also used to store its attribute and i-node numbers are reused after a file is removed.
- The **i-node generation number** is needed to increment each time i-node numbers are reused after a file is removed.
- File handles are passed from server to client in the results of *lookup*, *create* and *mkdir* operations and from client to server in the argument lists of all server operations.

NFS Client :

The NFS client module cooperates with the virtual file system in each client machine.

It operates in a similar manner to the conventional UNIX file system, transferring blocks of files to and from the server and caching the blocks in the local memory whenever possible.

If the file is local, a reference to the index of the local file will be made

If the file is remote, it contains the file handle of the remote file.

NFS server operations

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*.

- NFS access control and authentication
  - The NFS server is stateless server, so the user's identity and access rights must be checked by the server on each request.
  - 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.

➢ Kerberos has been integrated with NFS to provide a stronger and more comprehensive security solution.

Mount service

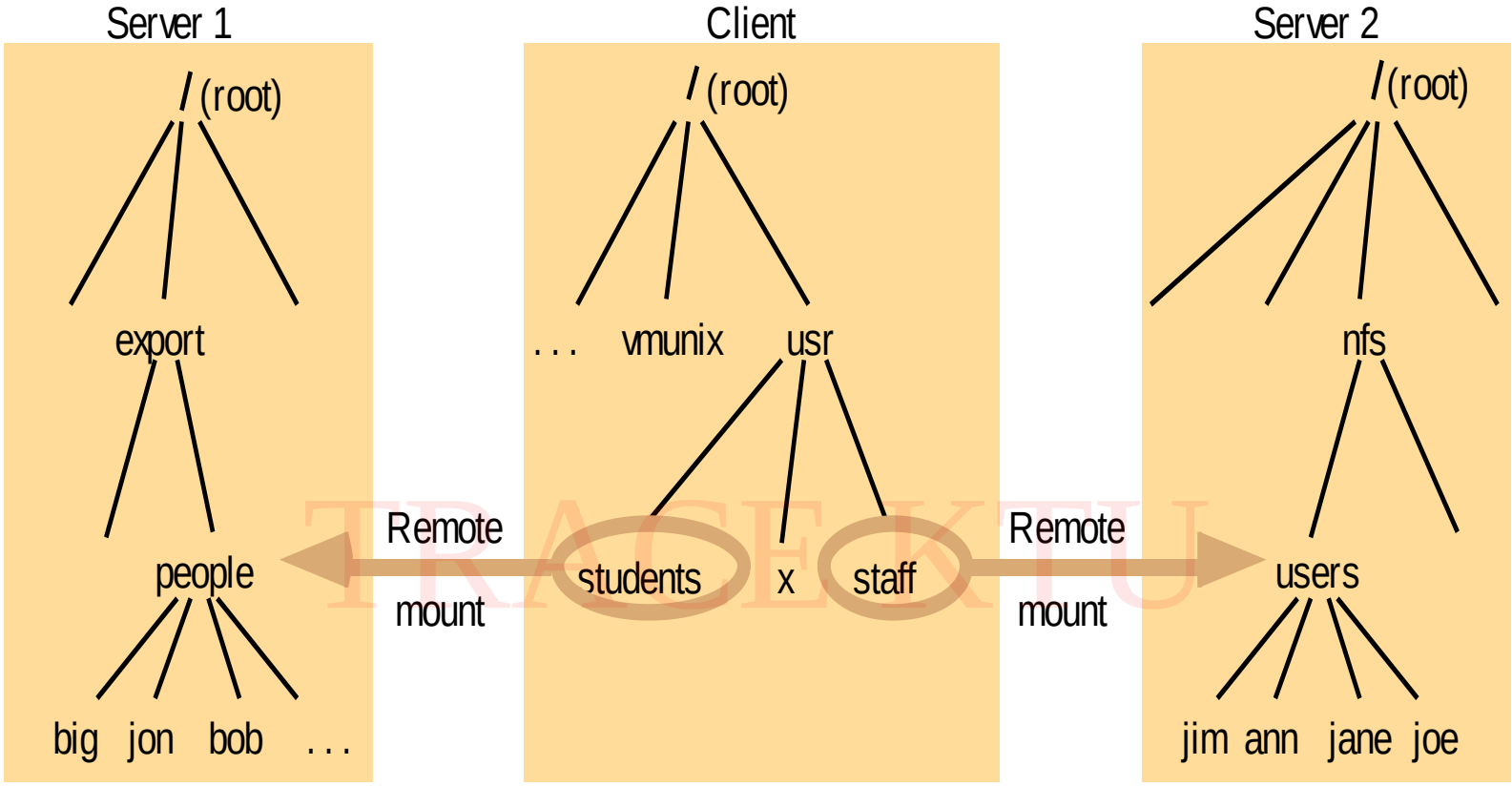
The mounting of subtrees of remote filesystems by clients is supported by a separate ***mount service*** process that runs at user level on each NFS server computer. On each server, there is a file with a well-known name (*/etc/exports*) containing the names of local filesystems that are available for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

- Mount operation:

  mount(remotehost, remotedirectory, localdirectory)
- Server maintains a table of clients who have mounted filesystems at that server.
- Each client maintains a table of mounted file systems holding:

  < IP address, port number, file handle>
- Figure illustrates a Client with two remotely mounted file stores.

Case Study: Sun NFS



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

- Automounter
  - The automounter was added to the UNIX implementation of NFS in order to mount a remote directory dynamically whenever a 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.

Server caching

In conventional UNIX systems, file pages, directories and file attributes that have been read from disk are retained in a main memory *buffer cache* until the buffer space is required for other pages. If a process then issues a read or a write request for a page that is already in the cache, it can be satisfied without another disk access

Client caching

The NFS client module caches the results of *read*, *write*, *getattr*, *lookup* and *readdir* operations in order to reduce the number of requests transmitted to servers.

The Andrew File System (AFS)

- 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. (exist as two process)

Vice is the name given to the server software that runs as a user-level UNIX process in each server computer

Venus is a user-level process that runs in each client computer and corresponds to the client module in our abstract model.

Architecture : The Andrew File System (AFS)

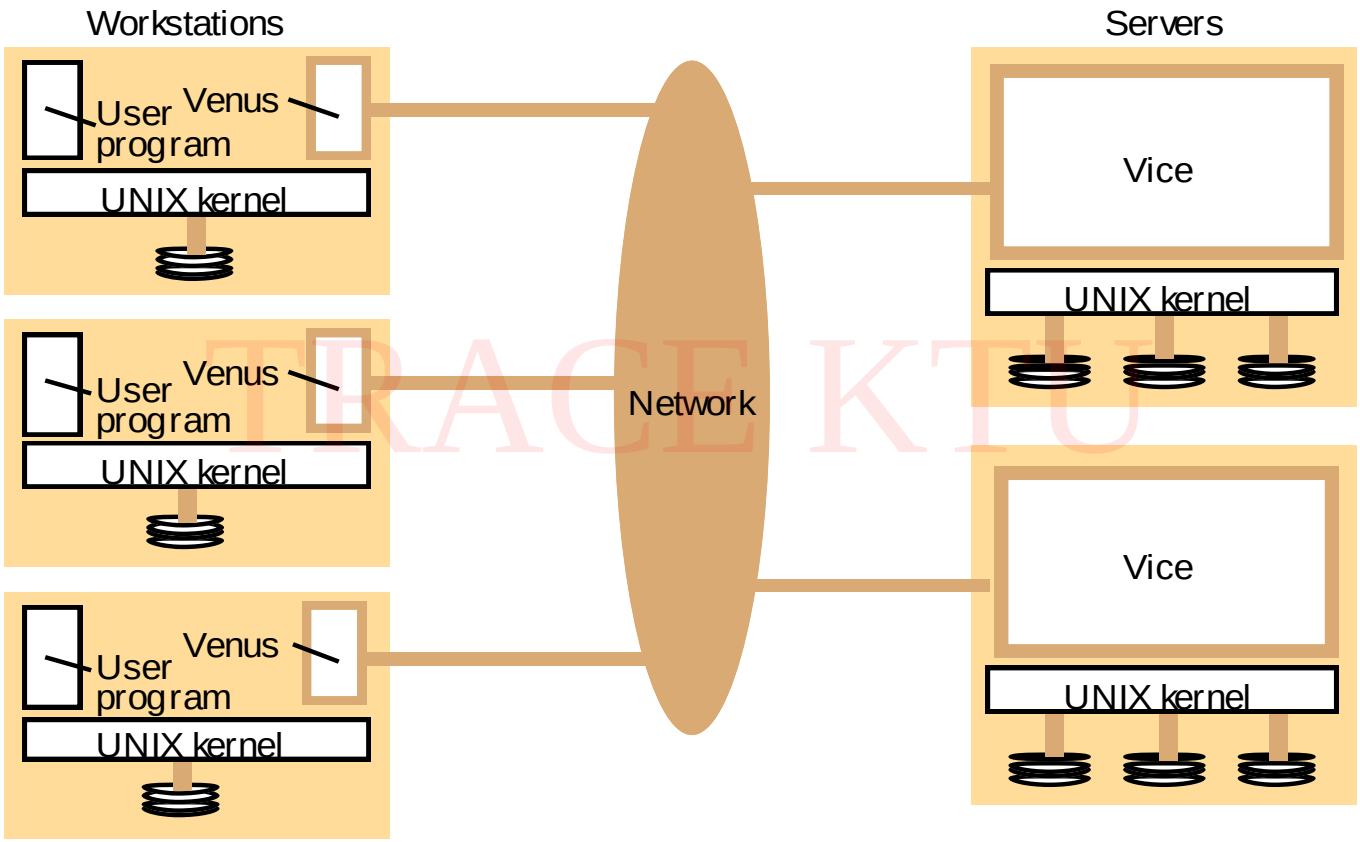


Figure 11. Distribution of processes in the Andrew File System

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

AFS is designed to perform well with ***larger numbers of active users than other distributed file systems***. The key strategy for achieving scalability is the caching of whole files in client nodes.

AFS has two 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.

Here is a simple scenario illustrating the operation of AFS:

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

Implementation

- Statefull Server in AFS allows, the server to inform all clients with open files about any updates made to that file by another client, through what is known as callback.
- 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.
- Callback promises are stored with the cached files on the workstation disks and have two states: valid or cancelled

Implementation

- A callback is a RPC 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

Implementation

- When a workstation is restarted after a failure or a shutdown, Venus aims to retain as many as possible of the cached files on the local disk, but it cannot assume that the callback promise tokens are correct, since some callbacks may have been missed.
- Before the first use of each cached file or directory after a restart, Venus therefore generates a cache validation request containing the file modification timestamp to the server that is the custodian of the file.
- If the timestamp is current, the server responds with valid and the token is reinstated.
- If the timestamp shows that the file is out of date, then the server responds with cancelled and the token is set to cancelled. Callbacks must be renewed

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.
		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.	←	
	Open the local file and return the file descriptor to the application.			
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.

Operations

Fetch(fid) -> attr, data	Returns the attributes (status) and, optionally, the contents of file .
Store(fid, attr, data)	Updates the attributes and (optionally) the contents of a specified file.
Create() -> fid	Creates a new file and records.
Remove(fid)	Deletes the specified file.
SetLock(fid, mode)	Sets a lock on the specified file or directory. .
ReleaseLock(fid)	Unlocks the specified file or directory.