KTU NOTES

Agreement in (message-passing) synchronous systems with failures : –

Consensus algorithm for crash failures

Vidya Academy of Science & Technology, Thrissur
Sivadasan E T
Associate Professor
Department of CSE.

Consensus algorithm for crash failures

- A set of processes need to agree on a value (decision), after one or more processes have proposed what that value (decision) should be

- Examples:

  mutual exclusion, election, transactions

Consensus algorithm for crash failures

System model

- *N* processes $\{p_1, p_2, ..., p_N\}$
- Communication is reliable but processes may fail.
- At most *f* processes out of *N* may be *faulty.*
  - Crash failure.
  - Byzantine failure (arbitrary).

- The system is logically fully connected.
- A receiver process knows the identity of the sender process.

Authenticated & Non-authenticated messages

- To reach an agreement, processes have to exchange their values and relay the received values to other processes.

Authenticated & Non-authenticated messages

Authenticated or *signed* message system – A (faulty) process cannot forge a message or change the contents of a received message (before it relays the message to other).

Because a process can verify the authenticity of a received message.

Authenticated & Non-authenticated messages

Non-authenticated or *unsigned* or *oral* message

– A (faulty) process can forge a message and claimed to have received it from another process or change the contents of a received message before it relays the message to other.

A process has no way of verifying the authenticity of a received message.

Consensus problem:

- ‘N’ processes agree on a value (e.g. synchronized action – go / abort)

- Consensus may have to be reached in the presence of failure

    - Process failure – crash/fail-stop, arbitrary failure
    - Communication failure

Consensus algorithm for crash failures

```
(global constants)
integer: $f$;          // maximum number of crash failures tolerated
(local variables)
integer: $x \longleftarrow$ local value;

(1)    Process $P_i$ $(1 \le i \le n)$ executes the consensus algorithm for up to
       $f$ crash failures:
(1a)   for round from 1 to $f+1$ do
(1b)       if the current value of $x$ has not been broadcast then
(1c)           broadcast($x$);
(1d)       $y_j \longleftarrow$ value (if any) received from process $j$ in this round;
(1e)       $x \longleftarrow min_{\forall j}(x, y_j)$;
(1f)   output $x$ as the consensus value.
```

Algorithm 14.1 Consensus with up to f fail-stop processes in a system of n processes, $n > f$ [8]. Code shown is for process P_i, $1 \le i \le n$.

Consensus algorithm for crash failures

- The above algorithm is a consensus algorithm where *“Consensus with up to **f** fail-stop processes in a system of **n** processes, **n>f**”.*

Consensus algorithm for crash failures

- Here, the consensus variable ***x*** is integer-valued.
- Each process has an initial value x_i. If up to ***f*** failures are to be tolerated, then the algorithm has ***f* +1** rounds.

Consensus algorithm for crash failures

- In each round, a process ***i*** sends the value of its variable $\boldsymbol{x_i}$ to all other processes if that value has not been sent before.
- Of all the values received within the round and its own value $\boldsymbol{x_i}$ at the start of the round, the process takes the minimum, and updates $\boldsymbol{x_i}$.

Consensus algorithm for crash failures

- After ***f* +*1*** rounds, the local value $\boldsymbol{x_i}$ is guaranteed to be the consensus value.

Consensus problem:

- ***The agreement condition*** is satisfied because in the ***f* +1** rounds, *there must be at least one round in which no process failed*.
- In this round, say round r, all the processes that have not failed so far succeed in broadcasting their values, and all these processes take the minimum of the values broadcast and received in that round.

Consensus problem:

- Thus, the local values at the end of the round are the same, say for all non-failed processes.

- In further rounds, only this value may be sent by each process at most once, and no process ***i*** will update its value x_i^r.

Consensus problem:

- The validity condition is satisfied because processes do not send fictitious values in this failure model. (Thus, a process that crashes has sent only correct values until the crash.)
- For all ***i***, if the initial value is identical, then the only value sent by any process is the value that has been agreed upon as per the agreement condition.

THANK YOU !

Distributed File System : –
File Service Architecture

Vidya Academy of Science & Technology, Thrissur
Sivadasan E T
Associate Professor
Department of CSE.

Distributed File System

- The sharing of resources is a key goal for distributed systems.
- The sharing of stored information is perhaps the most important aspect of distributed resource sharing.

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.

Distributed File System

- Web servers provide a restricted form of data sharing in which

  - Files stored locally in file systems at the server or in servers on a local network, are made available to clients throughout the Internet.

Distributed File System

- The <u>performance and reliability</u> experienced for access to files stored at a server should be comparable to that for files stored on local disks.

Distributed File System - Challenges

- The design of large-scale wide area read-write file storage systems poses problems of:

  - Load balancing,

  - Reliability,

  - Availability and

  - Security,

Distributed File System - Challenges

- A ***file service*** enables programs to store and access remote files exactly as they do local ones,
    - ➢ Allowing users to access their files from any computer in an intranet.

Distributed File System - Characteristics

- ***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.

Distributed File System - Characteristics

- 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.

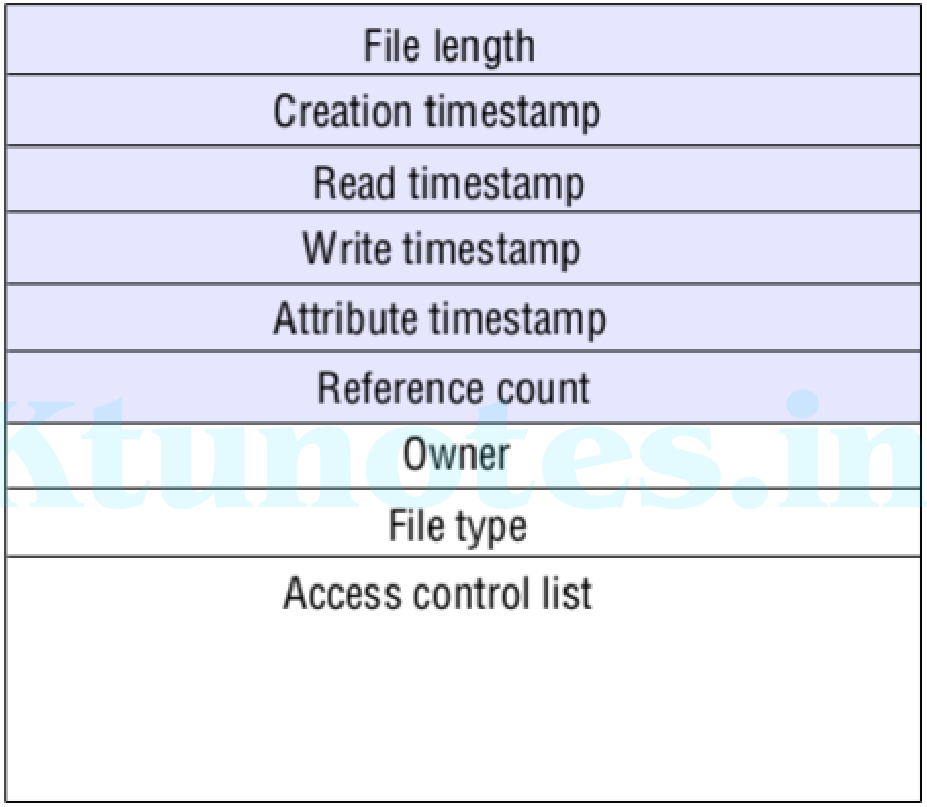
Distributed File System - Characteristics

➢ 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.

Distributed File System - Characteristics

File attribute record structure



File length
Creation timestamp
Read timestamp
Write timestamp
Attribute timestamp
Reference count
Owner
File type
Access control list

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 - Characteristics

- 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.

Distributed File System - Characteristics

- ➢ Directories may include the names of other directories.
- ➢ Leading to the familiar hierarchic filenaming scheme and the multi-part pathnames for files used in UNIX and other operating systems.

Distributed File System - Characteristics

➢ File systems also take responsibility for:

1. The control of access to files,

2.Restricting access to files according to users’ authorizations and

3. The type of access requested (reading, updating, executing and so on).

Distributed File System - Characteristics

- ➢ The main operations on files(create, open, close, read, and write etc...) that are available to applications in UNIX systems.
- ➢ These are the system calls implemented by the kernel.

Distributed File System - Characteristics

- ➢ The file system is responsible for applying access control for files.
- ➢ In local file systems such as UNIX, it does so when each file is opened,
- ➢ Checking the rights allowed for the user's identity in the access control list against the mode of access requested in the open system call.

Distributed File System - Characteristics

➢ If the rights match the mode,

- The file is opened and
- The mode is recorded in the open file state information.

Distributed file system requirements

1. **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

2. 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.

Distributed file system requirements

3. **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.

Distributed file system requirements

4. Performance transparency: - Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.

Distributed file system requirements

5. Scaling transparency:- The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.

Distributed file system requirements

6. 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.

Distributed file system requirements

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

Distributed file system requirements

File replication has two benefits : -

– it enables multiple servers to share the load of providing a service to clients accessing the same set of files and

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

Distributed file system requirements

8. **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

9.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.

Distributed file system requirements

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

Distributed file system requirements

One-copy update semantics

➢ 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.

Distributed file system requirements

11. 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.

Distributed file system requirements

Security continues…-

➢ So that access control at the server is based on correct user identities.

➢ Protect the contents of request and reply messages with digital signatures and (optionally) encryption of secret data.

Distributed file system requirements

11. Efficiency - A distributed file service should offer facilities that are of at least the same power.

➢ Generality as those found in conventional file systems.

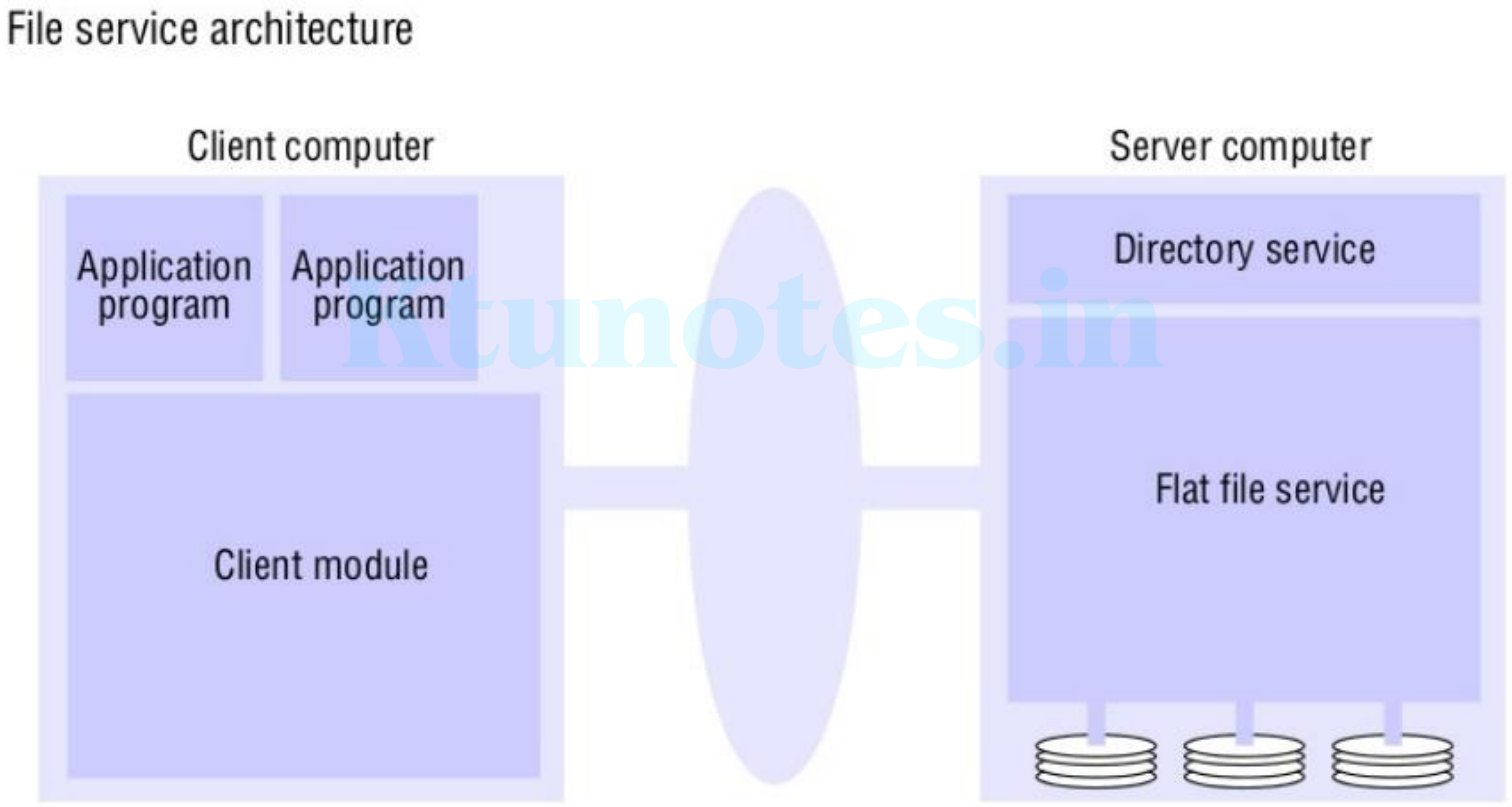
➢ Should achieve a comparable level of performance.

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 :-

- Flat file service
- Directory service
- Client module

File Service Architecture



File Service Architecture

The ***flat file service and the directory service*** each export an interface:

➢ For use by client programs, and

➢ Their Request Procedure Call (RPC) interfaces,

➢ Taken together, provide a comprehensive set of operations for access to files.

File Service Architecture

The ***client module*** provides a single programming interface with operations on files similar to those found in conventional file systems.

File Service Architecture

The ***design is open*** in the sense that :-

- Different client modules can be used to implement different programming interfaces,
- Simulating the file operations of a variety of different operating systems.
- Optimizing the performance for different client and server hardware configurations.

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.

File Service Architecture

Flat file service:

➢ UFIDs are long sequences of bits and 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.

File Service Architecture

Directory service: -

➢ The <u>*directory service provides a mapping between text names for files and their UFIDs*</u>.

➢ Clients may obtain the UFID of a file by quoting its text name to the directory service.

File Service Architecture

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.

➢ *<u>It is a client of the flat file service</u>*; 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.

File Service Architecture

Client module : -

➢ The client module also holds information about the network locations of the flat file server and directory server processes.

➢ 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.

File Service Architecture

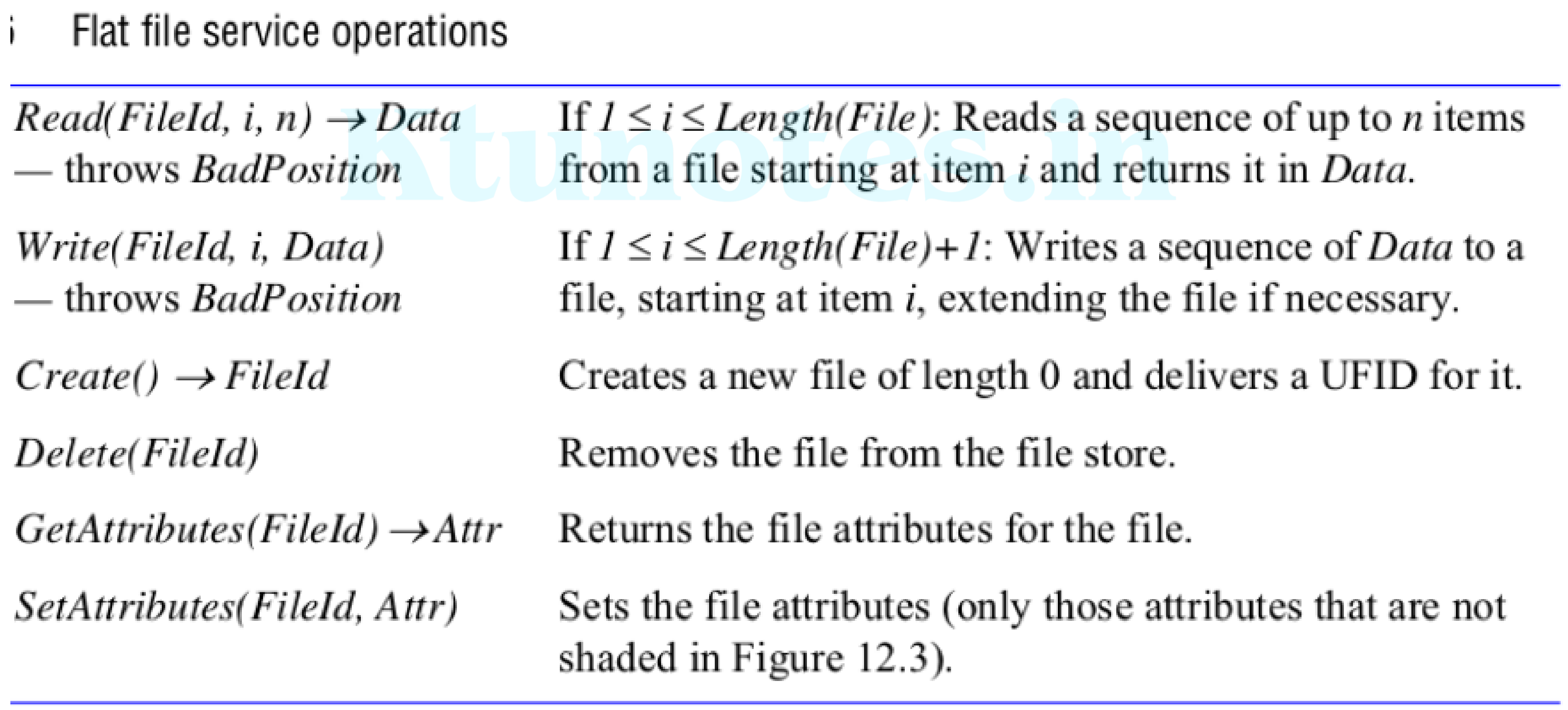
Flat file service interface

➢ This is the RPC interface used by client modules.

➢ It is not normally used directly by user-level programs.

File Service Architecture

The information below contains a definition of the interface to a flat file service.



Flat file service operations

Read(FileId, i, n) $\rightarrow$ *Data* — throws *BadPosition*	If $1 \leq i \leq Length(File)$: 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 $1 \leq i \leq Length(File)+1$: Writes a sequence of *Data* to a file, starting at item i, extending the file if necessary.
Create() $\rightarrow$ *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) $\rightarrow$ *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

Flat file service interface

➢ 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.

File Service Architecture

Flat file service interface

➢ 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.

File Service Architecture

Flat file service interface

- ➢ 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.

File Service Architecture

Flat file service interface

➢ Create creates a new, empty file and returns the UFID that is generated.

➢ Delete removes the specified file.

File Service Architecture

Flat file service interface

➢ 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 would normally be restricted to the directory service that provides access to the file.

THANK YOU !

Vidya Academy of Science & Technology, Thrissur

Network File System

CS402

Mr. Sivadasan E.T.
Associate Professor
Computer Science & Engineering

Network File System

- Sun Microsystem’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.

Network File System

- The design and implementation of NFS have achieved success both technically and commercially.

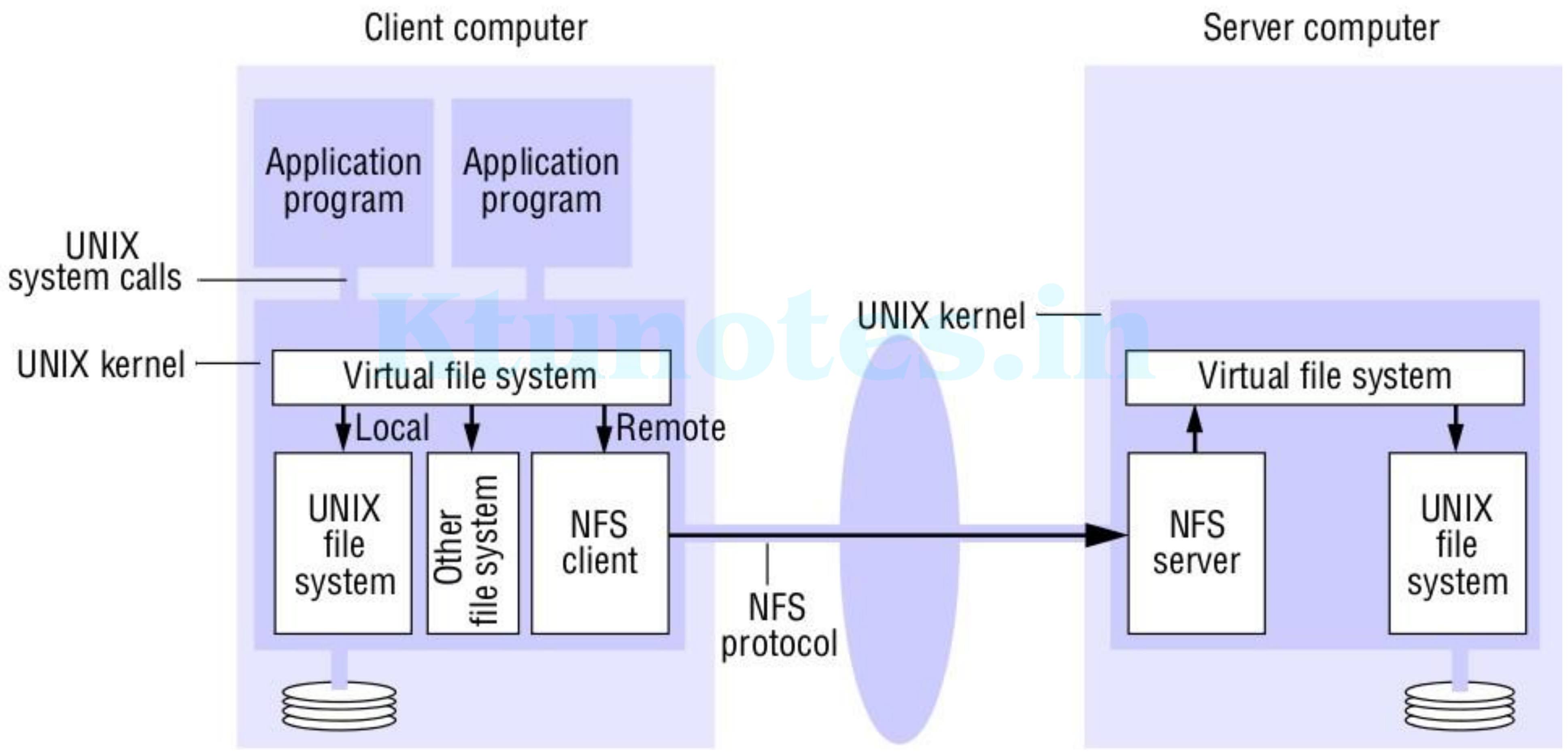
- NFS provides transparent access to remote files for client programs running on UNIX and other systems.

Network File System

- 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.

Figure 12.8 NFS architecture



Network File System

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

- The NFS server module resides in the kernel on each computer that acts as an NFS server.

Network File System

- 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.

Network File System

- The NFS client and server modules communicate using remote procedure calls.

- 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.

Network File System

- 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

NFS provides access transparency:

- User programs can issue file operations for local or remote files without distinction.

- The integration for local or remote files without distinction is achieved by a virtual file system (VFS) module, which has been added to the UNIX kernel to distinguish between local and remote files.

Virtual file system

File handle:	Filesystem identifier	i-node number of file	i-node generation number

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

Virtual file system

- 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.

- The i-node generation number is needed because in the conventional UNIX file system i-node numbers are reused after a file is removed.

Virtual file system

- 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.

- A VFS structure relates a remote file system to the local directory on which it is mounted.

Virtual file system

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

- The v-node contains an indicator to show whether a file is local or remote.

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.

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

- It shares the same buffer cache that is used by the local as well as remote input-output system.

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.

NFS server interface

- 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.

NFS server interface

- The file and directory operations are integrated in a single service;

- The creation and insertion of file names in directories is performed by a single create operation,

- Which takes the text name of the new file and the file handle for the target directory as arguments.

- The other NFS operations on directories are create, remove, rename, link etc..

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.

- 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.*

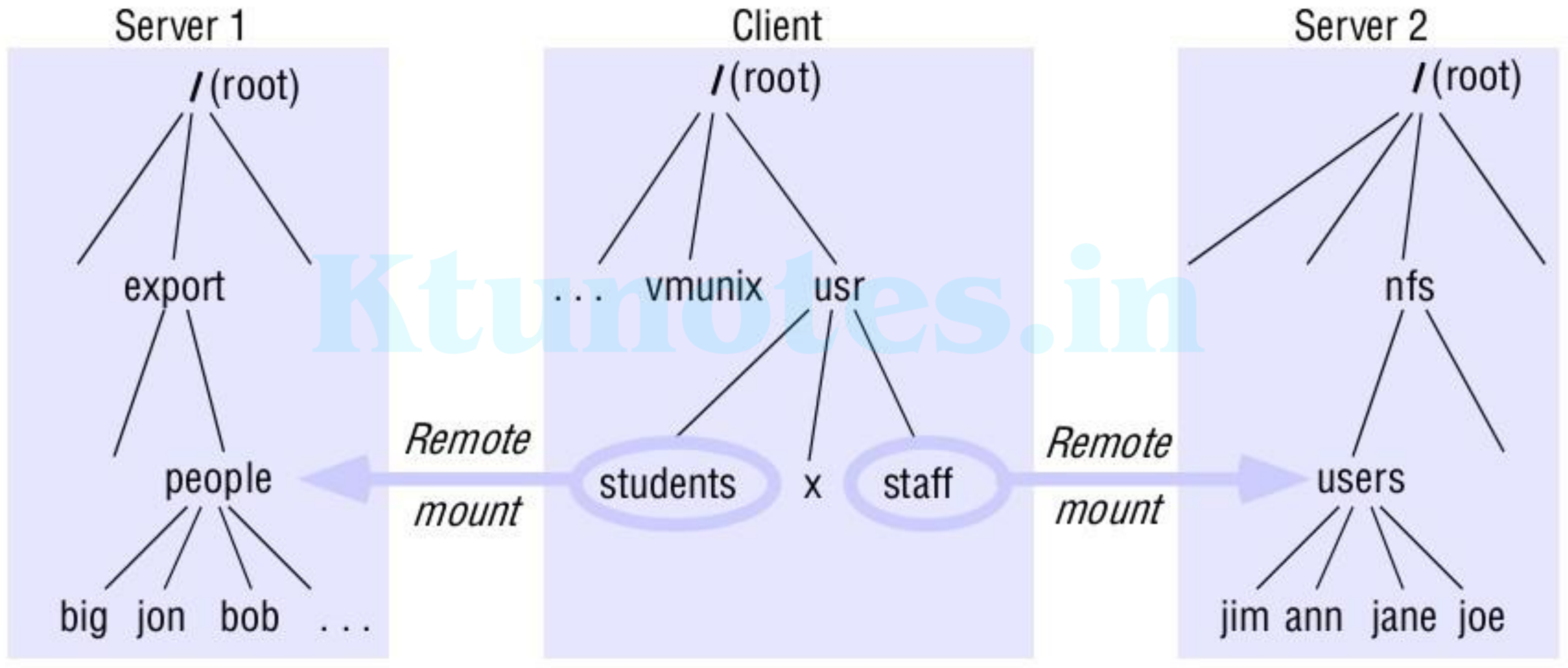
Mount service

- ➢ The remote directory may be any subtree of the required remote filesystem, enabling clients to mount any part of the remote filesystem.

- ➢ The location (IP address and port number) of the server and the file handle for the remote directory are passed on to the VFS layer and the NFS client.

Mount service

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.

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.
- 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.
- The lookup operation looks for a single part of a pathname in a given directory and returns the corresponding file handle and file attributes.

Automounter

- The automounter maintains a table of mount points (pathnames) with a reference to one or more NFS servers listed against each.
- It behaves like a local NFS server at the client machine.

Automounter

- When the NFS client module attempts to resolve a pathname that includes one of these mount points, it passes to the local automounter a lookup() request that locates the required filesystem in its table and sends a ‘probe’ request to each server listed.

- The filesystem on the first server to respond is then mounted at the client using the normal mount service.

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 issues a read or a write request for a page that is already in the cache, it can be satisfied without another disk access.
- Read-ahead anticipates read accesses and fetches the pages following those that have most recently been read, and delayed-write optimizes writes.

Client caching

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

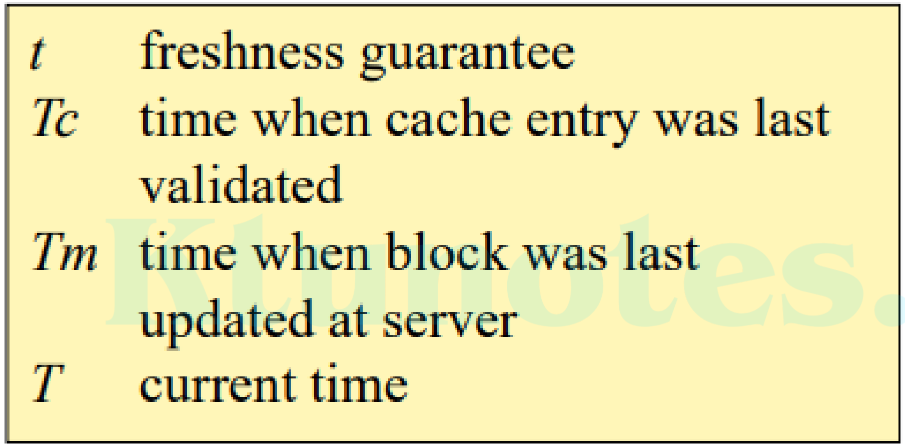
Client caching

- 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 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.

Client caching

- 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.

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.

Thank you !..

Vidya Academy of Science & Technology, Thrissur

Andrew File System

CS402

Mr. Sivadasan E.T.
Associate Professor
Computer Science & Engineering

Andrew File System

- 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.

Andrew File System

AFS has two unusual design characteristics:

- *1. Whole-file serving:* The entire contents of directories and files are transmitted to client computers by AFS servers.

Andrew File System

- 2. *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.

Andrew File System

Scenario illustrating the operation of AFS:

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

Andrew File System

Scenario illustrating the operation of AFS:

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.

Andrew File System

Scenario illustrating the operation of AFS:

3. Subsequent read, write and other operations on the file by processes in the client computer are applied to the local copy.

Andrew File System

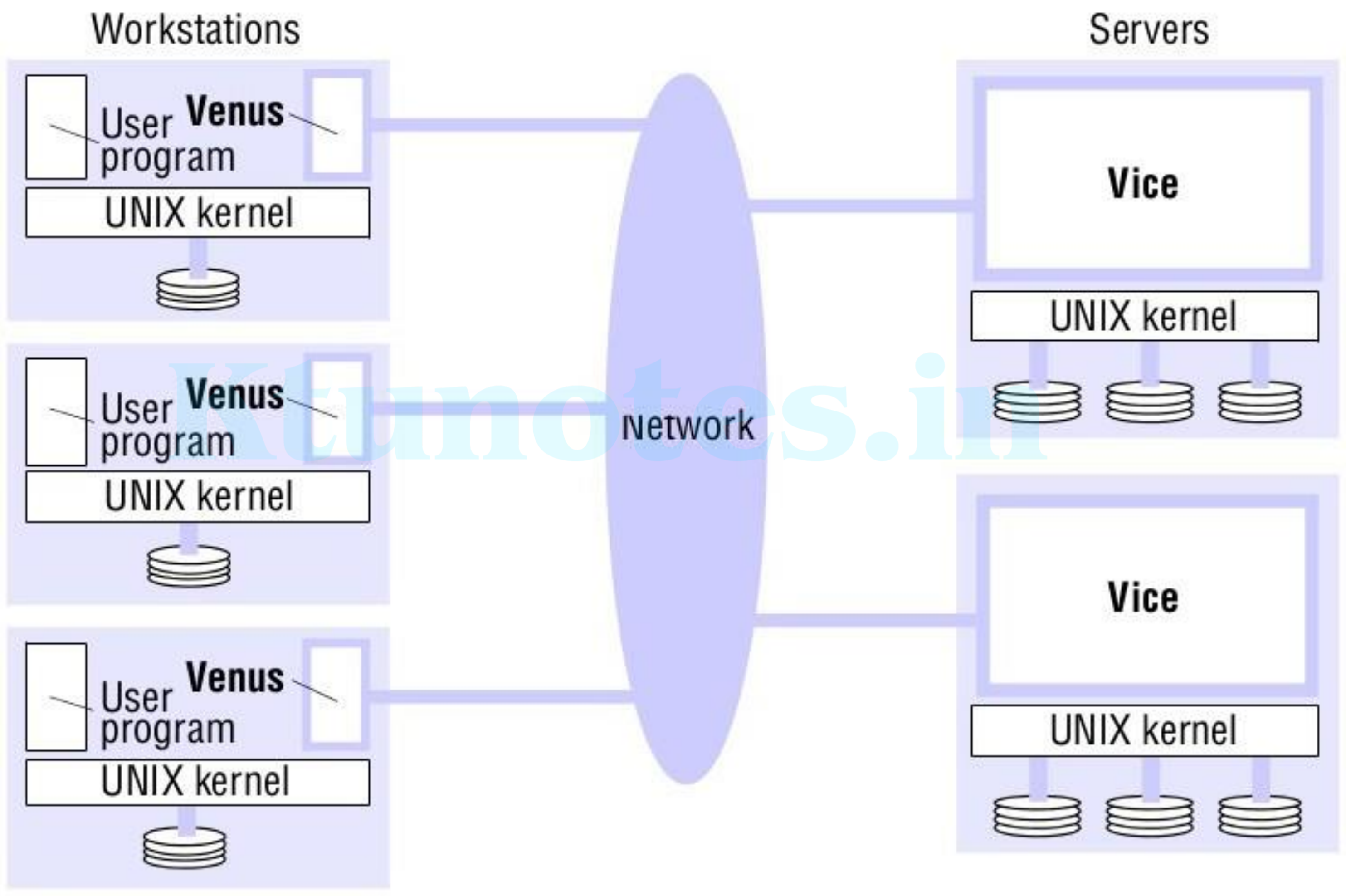
Scenario illustrating the operation of AFS:

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.

Distribution of processes in the Andrew File System



Andrew File System

Implementation

- AFS is implemented as two software components that exist as UNIX processes called Vice and Venus.

- 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.

Andrew File System

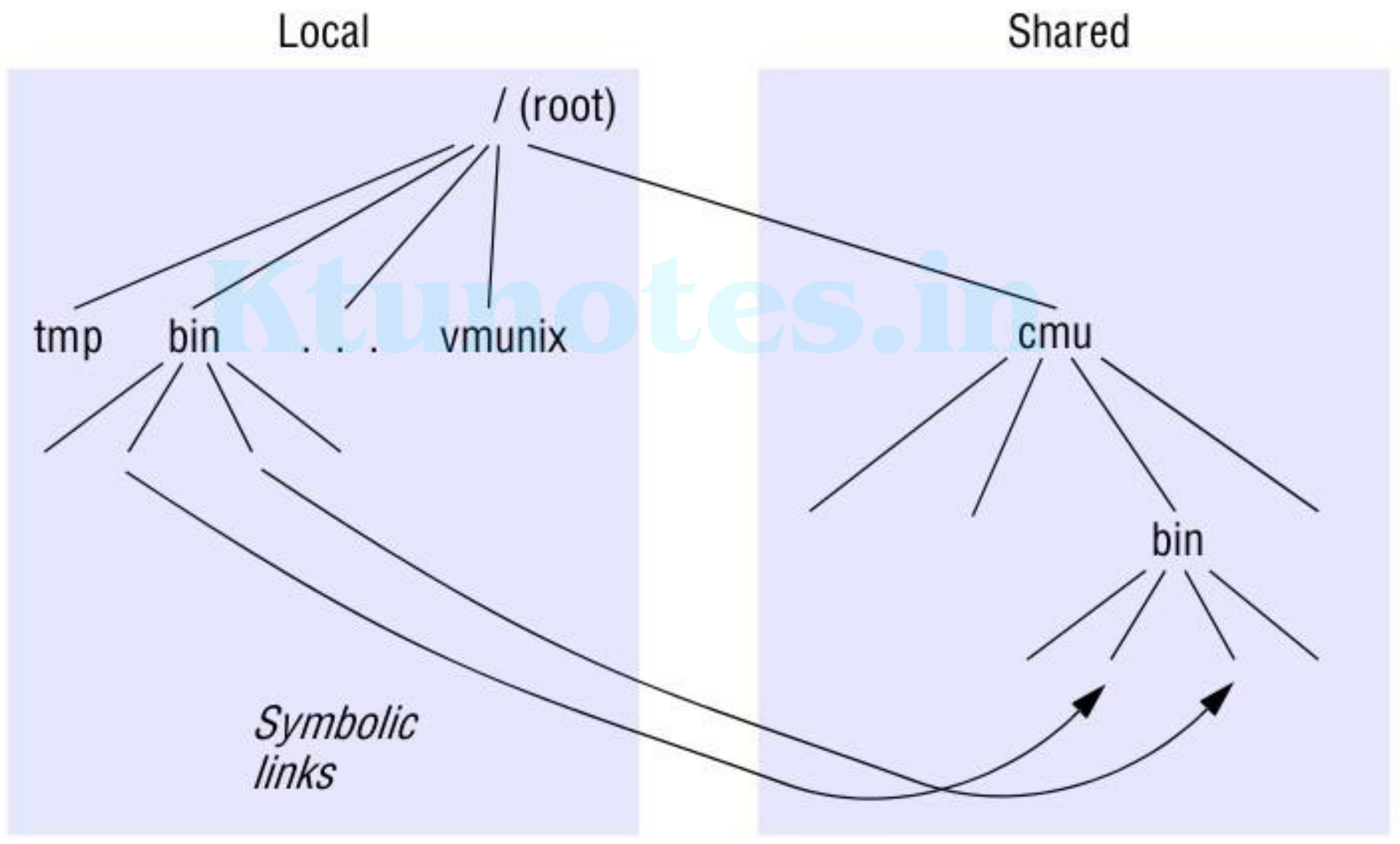
- The files available to user processes running on workstations are either local or shared.

- Local files are handled as normal UNIX files. They are stored on 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

File name space seen by clients of AFS



Andrew File System

- It is a conventional UNIX directory hierarchy, with a specific subtree (called cmu - Carnegie Mellon University) 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.

Andrew File System

- Local files are used only for temporary files (/tmp) and processes that are essential for workstation startup.
- Other standard UNIX files (such as those normally found in /bin, /lib and so on) are implemented as symbolic links from local directories to files held in the shared space.
- Users’ directories are in the shared space, enabling users to access their files from any workstation.

Andrew File System

- 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.

Andrew File System

➢ 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.

Andrew File System

- The abstract file service model of Andrew File System.
  - A flat file service is implemented by the Vice servers.

  -The hierarchic directory structure required by UNIX user programs is implemented by the set of Venus processes in the workstations.

Andrew File System

- The abstract file service model of Andrew File System.

  - 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*.

Andrew File System

- 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.

Andrew File System

32 bits	*32 bits*	*32 bits*
Volume number	File handle	Uniquifier

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

Andrew File System

- User programs use conventional UNIX pathnames to refer to files.
- But AFS uses *fids* in the communication between the Venus and Vice processes.
- The Vice servers accept requests only in terms of *fids*.
- Venus translates the pathnames supplied by clients into fids using a step-by-step lookup to obtain the information from the file directories held in the Vice servers.

Andrew File System

- Figure below describes the actions taken by Vice, Venus and the UNIX kernel when a user process issues each of the system calls mentioned in our outline scenario above.

- 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.

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. 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.

Andrew File System

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.

- Callback promises are stored with the cached files on the workstation disks and have two states: *valid* or *cancelled*.

Andrew File System

Cache consistency:

- 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.*

Andrew File System

Cache consistency:

- 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.

Andrew File System

Cache consistency:

- 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.

Figure 12.15 The main components of the Vice service interface

Fetch(fid) $\rightarrow$ *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() $\rightarrow$ *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.

Note: Directory and administrative operations (*Rename*, *Link*, *Makedir*, *Removedir*, *GetTime*, *CheckToken* and so on) are not shown.

Thank you !..