DISTRIBUTED COMPUTING
MODULE 5

# Consensus and agreement algorithms

A consensus algorithm is a process that achieves agreement on a single data value among distributed processes or systems. Consensus algorithms necessarily assume that some processes and systems will be unavailable and that some communications will be lost. Hence these algorithms must be fault-tolerant. Examples of consensus algorithm: (1) Deciding whether to commit a distributed transaction to a database. (2) Designating node as a leader for some distributed task. (3) Synchronizing state machine replicas and ensuring consistency among them.

Assumptions in Consensus algorithms:

- **Failure models:**

Some of the processes may be faulty in distributed systems. A faulty process can behave in any manner allowed by the failure model assumed. Some of the well known failure models includes fail-stop, send omission and receive–omission, and Byzantine failures. Fail stop model: a process may crash in the middle of a step, which could be the execution of a local operation or processing of a message for a send or receive event. it may send a message to only a subset of the destination set before crashing. Byzantine failure model: a process may behave arbitrarily. The choice of the failure model determines the feasibility and complexity of solving consensus.

- **Synchronous/asynchronous communication:**

If a failure-prone process chooses to send a message to process but fails, then– intended process cannot detect the non-arrival of the message. This is because scenario is indistinguishable from the scenario in which the message takes a very long time in transit. This is a major hurdle in asynchronous system. In a synchronous system, a unsent message scenario can be identified by the intended– recipient, at the end of the round. The intended recipient can deal with the non-arrival of the expected message by assuming the arrival of a message containing some default data, and then proceeding with the next round of the algorithm.

- **Network connectivity:**

The system has full logical connectivity, i.e., each process can communicate with any– other by direct message passing.

- **Sender identification:**

A process that receives a message always knows the identity of the sender process. When multiple messages are expected from the same sender in a single round, a scheduling algorithm is employed that sends these messages in sub-rounds, so that each message sent within the round can be uniquely identified.

- **Channel reliability:**

The channels are reliable, and only the processes may fail.

- **Authenticated vs non-authenticated messages:**

With unauthenticated messages, when a faulty process relays a message to other processes (i) it can forge the message and claim that it was received from another process, (ii) it can also tamper with the contents of a received message before relaying it. When a process receives a message, it has no way to verify its authenticity. This is known as un authenticated message or oral message or an unsigned message. Using authentication via techniques such as digital signatures, it is easier to solve the agreement problem because, if some process forges a message or tampers with the contents of a received message before relaying it, the recipient can detect the forgery or tampering. Thus, faulty processes can inflict less damage.

- **Agreement variable:**

The agreement variable may be boolean or multivalued, and need not be an integer. This simplifying assumption does not affect the results for other data types, but helps in– the abstraction while presenting the algorithms.

The Byzantine agreement and other problems:

- **The Byzantine agreement problem**

The *Byzantine agreement* problem requires a designated process, called the *source process*, with an *initial value*, to reach agreement with the other processes about its initial value, subject to the following conditions:

Agreement All non-faulty processes must agree on the same value.

Validity If the source process is non-faulty, then the agreed upon value by all the non-faulty processes must be the same as the initial value of the source.
Termination Each non-faulty process must eventually decide on a value.

If the source process is faulty, then the correct processes can agree upon any value. It is irrelevant what the faulty processes agree upon – or whether they terminate and agree upon anything at all.

- **The consensus problem**

The consensus problem differs from the Byzantine agreement problem in that each process has an initial value and all the correct processes must agree on a single value. Formally:

Agreement All non-faulty processes must agree on the same (single) value.
Validity If all the non-faulty processes have the same initial value, then the agreed upon value by all the non-faulty processes must be that same value.
Termination Each non-faulty process must eventually decide on a value.

- **The interactive consistency problem**

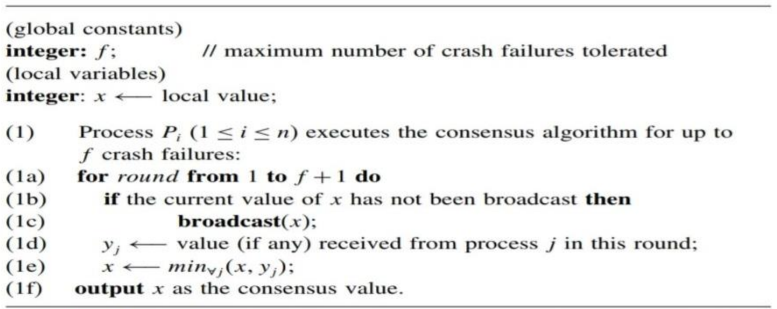
The interactive consistency problem differs from the Byzantine agreement problem in that each process has an initial value, and all the correct processes must agree upon a set of values, with one value for each process The formal specifications are:

Agreement: All non-faulty processes must agree on the same array of values A[v1, …,vn].
Validity: If process i is non-faulty and its initial value is vi, then all non faulty processes agree on vi as the ith element of the array A. If process j is faulty, then the non-faulty processes can agree on any value for A[j].
Termination: Each non-faulty process must eventually decide on the array A. The difference between the agreement problem and the consensus problem is that, in the agreement problem, a single process has the initial value, whereas in the consensus problem, all processes have an initial value.

# Agreement in (message-passing) synchronous systems with failures

Consensus algorithm for crash failures (synchronous system):

- The consensus algorithm for n processes where up to f processes where f < n may fail in a fail stop failure model.
- Here the consensus variable x is integer value; each process has initial value xi. If up to f failures are to be tolerated than algorithm has f+1 rounds, in each round a process i sense the value of its variable xi to all other processes if that value has not been sent before.
- So, of all the values received within that round and its own value xi at that start of the round the process takes minimum and updates xi occur f + 1 rounds the local value xi guaranteed to be the consensus value.
- If one process is faulty, among three processes then f = 1. So the agreement requires f + 1 that is equal to two rounds.
- If it is faulty let us say it will send 0 to 1 process and 1 to another process i, j and k. Now, on receiving one on receiving 0 it will broadcast 0 over here and this particular process on receiving 1 it will broadcast 1 over here.
- So, this will complete one round in this one round and this particular process on receiving 1 it will send 1 over here and this on the receiving 0 it will send 0 over here.



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

(1)    Process $P_i$ ($1 \leq i \leq n$) 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, $1 \leq i \leq n$.

- 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.
- Thus, the local values at the end of the round are the same, say x^r_i 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^r_i.
- The validity condition is satisfied because processes do not send fictitious values in this failure model.
- 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.
- The termination condition is seen to be satisfied.

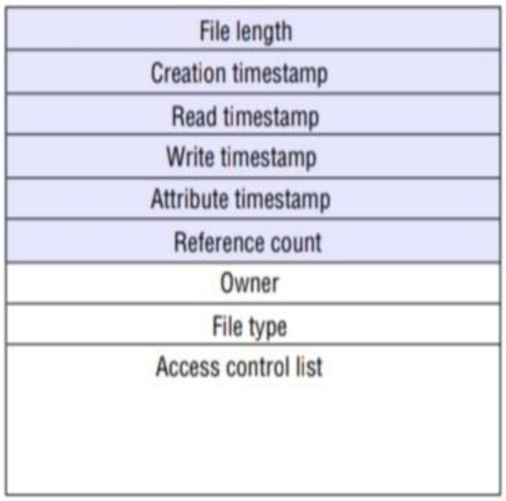
Complexity: The complexity of this particular algorithm is it requires f + 1 rounds where f < n and the number of messages is $O(n^2)$in each round and each message has one integers hence the total number of messages is $O((f+1)\cdot n^2)$ is the total number of rounds and in each round n 2 messages are required.

Lower bound on the number of rounds: At least f + 1 rounds are required, where f < n. In the worst-case scenario, one process may fail in each round; with f + 1 rounds, there is at least one round in which no process fails. In that guaranteed failure-free round, all messages broadcast can be delivered reliably, and all processes that have not failed can compute the common function of the received values to reach an agreement value.

# Distributed File System

- A file system is a subsystem of the operating system that performs file management activities such as organization, storing, retrieval, naming, sharing, and protection of files.
- A distributed file system(DFS) is a method of storing and accessing files based in a client/server architecture.
- 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. The shaded attributes in the Fig. are managed by the file system and are not normally update by user programs.

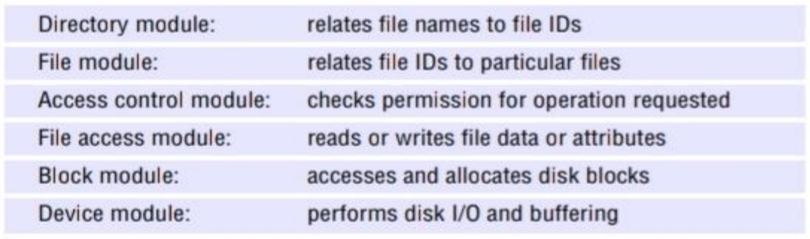
Figure 12.3 File attribute record structure



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

- A distributed file system contains the following set of modules:

Figure 12.2 File system modules



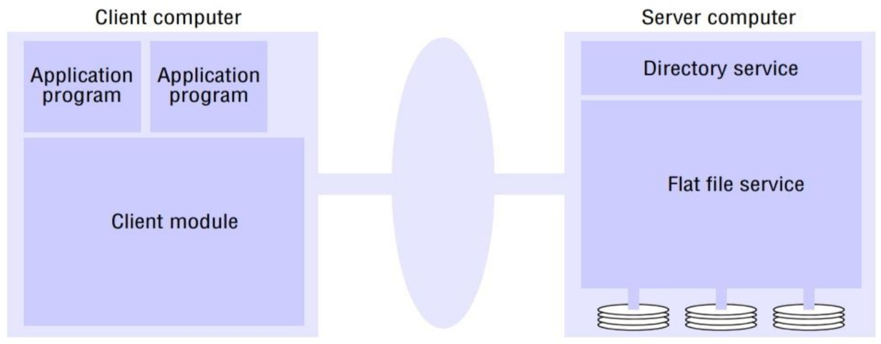
Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	performs disk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

Distributed file system requirements:

1. **Transparency:** The file service is usually the most heavily loaded service in an intranet, so its functionality and performance are critical. The following forms of transparency are partially or wholly addressed by current file services:
   - *Access transparency*: Client programs should be unaware of the distribution of files. A single set of operations is provided for access to local and remote files. Programs written to operate on local files are able to access remote files without modification.
   - *Location transparency*: Client programs should see a uniform file name space. Files or groups of files may be relocated without changing their pathnames, and user programs see the same name space wherever they are executed.
   - *Mobility transparency*: Neither client programs nor system administration tables in client nodes need to be changed when files are moved. This allows file mobility – files or, more commonly, sets or volumes of files may be moved, either by system administrators or automatically.
   - *Performance transparency*: Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.
   - *Scaling transparency*: The service can be expanded by incremental growth to deal with a wide range of loads and network sizes.
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. This has two benefits – it enables multiple servers to share the load of providing a service to clients accessing the same set of files, enhancing the scalability of the service, and it enhances fault tolerance. Few file services support replication fully, but most support the caching of files or portions of files locally, a limited form of replication.
4. **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.
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. To cope with transient communication failures, the design can be based on *at-most-once* invocation semantics or it can use the simpler *at-least-once* semantics. Tolerance of disconnection or server failures requires file replication.
6. **Consistency** Conventional file systems such as that provided in UNIX offer *one-copy update semantics*. If any changes made to one file, that changes must do in other replicated copies.
7. **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.
8. **Efficiency** A distributed file service should offer facilities that are of at least the same power and generality as those found in conventional file systems and should achieve a comparable level of performance.

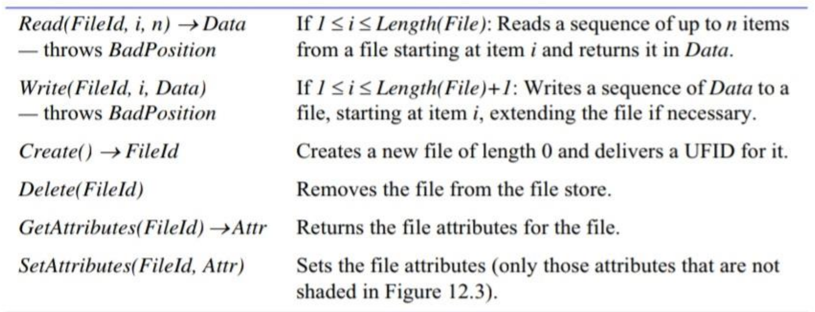
# File Service Architecture

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



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 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.
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. UFIDs are long sequences of bits chosen so that each file has a UFID that is unique among all of the files in a distributed system. When the flat file service receives a request to create a file, it generates a new UFID for it and returns the UFID to the requester.

Figure 12.6 Flat file service operations



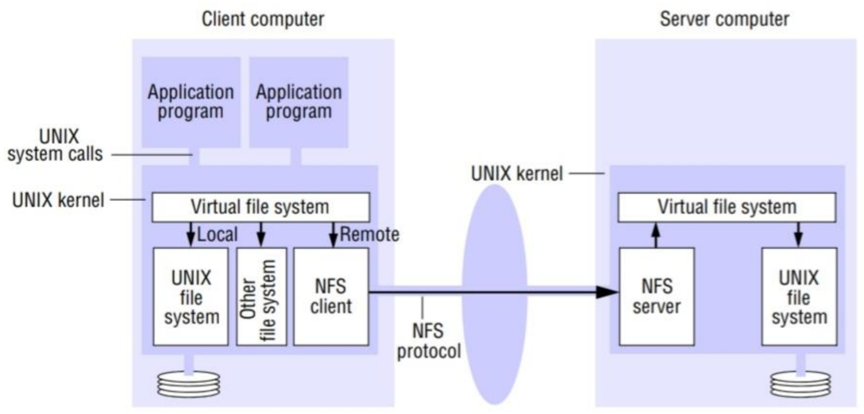
Read(FileId, i, n) → Data — throws *BadPosition*	If $1 \le i \le Length(File)$: Reads a sequence of up to n items from a file starting at item i and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \le i \le Length(File)+1$: Writes a sequence of *Data* to a file, starting at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

Access control: An access check is made whenever a file name is converted to a UFID**.** A user identity is submitted with every client request, and access checks are performed by the server for every file operation.
File groups • A *file group* is a collection of files located on a given server. A server may hold several file groups, and groups can be moved between servers, but a file cannot change the group to which it belongs. In a distributed file service, file groups support the allocation of files to file servers.
File group identifiers must be unique throughout a distributed system. Since file groups can be moved and distributed systems that are initially separate can be merged to form a single system, the only way to ensure that file group identifiers will always be distinct in a given system is to generate them with an algorithm that ensures global uniqueness. For example, whenever a new file group is created, a unique identifier can be generated by concatenating the 32-bit IP address of the host creating the new group with a 16-bit integer derived from the date, producing a unique 48-bit integer.

file group identifier:	32 bits	16 bits
	IP Address	date

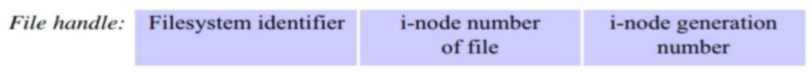
# SUN NETWORK FILE SYSTEM (NFS)

Sun Network File System- Sun NFS
Network File System (NFS) is a distributed file system protocol originally developed by Sun Microsystems (Sun)
NFS allow a user on a client computer to access files from remote server.
NFS client and server modules communicate using remote procedure calls (RPC).



Virtual file system (VFS)

- Virtual file system (VFS) module, is added to the UNIX kernel to distinguish between local and remote files.The file identifiers used in NFS are called file handles.



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

- 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 sysytem 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
- 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 cooperates with the virtual file system in each client machine.
- If the file is local, access the unix file system for the local file.
- If the file is remote, NFS client send request to NFS server
- 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.

Access control and authentication

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 on each request, to see whether the user is permitted to access the file in the manner requested.

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,

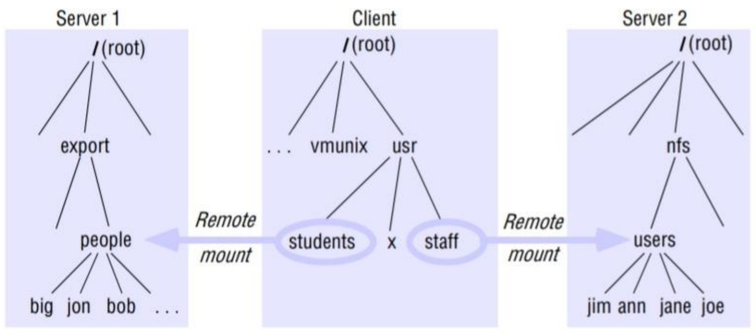
Figure 12.9 NFS server operations (NFS version 3 protocol, simplified)

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*.

Mount services:

- Mount the remote directories to the local directories.
- Mount is to make a group of files in a file system structure accessible to a user or user group.
- Mount operation: mount(remotehost, remotedirectory, localdirectory)

Figure 12.10 Local and remote filesystems accessible on an NFS client



- Client with two remotely mounted file stores. The nodes people and users in file systems at Server 1 and Server 2 are mounted over nodes students and staff in Client's local file store. The meaning of this is that programs running at Client can access files at Server 1 and Server 2 by using pathnames such as /usr/students/jon and /usr/staff/ann.
- Mount can be of 3 types:
  1. Soft mount: a time bound is there. (send failure message)
  2. Hard mount: no time bound. (retry the request until it is satisfied)
  3. Auto mount: mount operation done on demand.

Server caching

The use of the server's cache to hold recently read disk blocks does not raise any consistency problems but when a server performs write operations, extra measures are needed to ensure that clients can be confident that the results of the write operations are persistent, even when server crashes occur. The write operation offers two options for this :

- Data in write operations received from clients is stored in the memory cache at the server and written to disk before a reply is sent to the client. This is called write- through caching.
- Data in write operations is stored only in the memory cache. It will be written to disk when a commit operation is received for the relevant file.

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. 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.

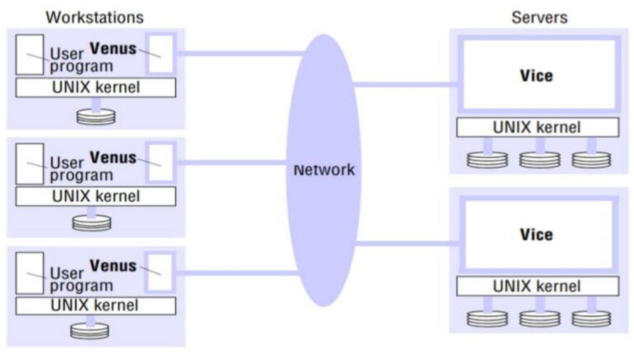
# ANDREW FILE SYSTEM

Andrew File System (AFS)
AFS is a distributed file system which uses a set of trusted servers to access file.
AFS uses a local cache to reduce the workload and increase the performance of a distributed computing environment.

Implementation and Distribution of processes in AFS

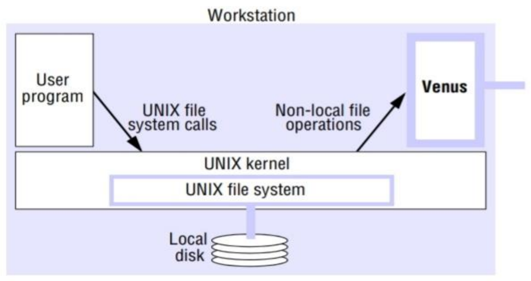
- AFS is implemented as two software components that exist at UNIX processes called Vice and Venus.
- Vice is the name given to the server software that runs as a user-level UNIX process in each server computer
- Venus is a user-level process that runs in each client computer and corresponds to the client module in our abstract model.
- The set of files in one server is referred to as a volume.
- The files available to user processes running on workstations are either local or shared.
- Local files are handled as normal UNIX files.
- Shared files are stored on servers, and copies of them are cached on the local disks of workstations.
- Step 1: A first request for data to a server from a workstation is satisfied by the server and placed in a local cache.

Figure 12.11 Distribution of processes in the Andrew File System



- Step 2: A second request for the same data is satisfied from the local cache.

Figure 12.13 System call interception in AFS



- Stateful servers in AFS allow 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.
- Callbacks to all clients with a copy of that file is ensured as a callback promise that is issued by the server to a client when it requests for a copy of a file.

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.

- Whenever Venus handles an open on behalf of a client, it checks the cache. If the required file is found in the cache, then its token is checked. If its value is cancelled, then a fresh copy of the file must be fetched from the Vice server, but if the token is valid, then the cached copy can be opened and used without reference to vice.
- 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.
- When a workstation is restarted after a failure or a shutdown, Venus generates a cache validation request containing the file modification timestamp to the server.
- 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*.

AFS has two unusual design characteristics:
• **Whole-file serving:** The entire contents of directories and files are transmitted to client computers by AFS servers (in AFS-3, files larger than 64 kbytes are transferred in 64-kbyte chunks).
• **Whole-file caching:** Once a copy of a file or a chunk has been transferred to a client computer it is stored in a cache on the local disk. The cache contains several hundred of the files most recently used on that computer. The cache is permanent, surviving reboots of the client computer. Local copies of files are used to satisfy clients' open requests in preference to remote copies whenever possible.

Operation of AFS

1. When a user process in a client computer issues an open system call for a file in the shared file space and there is not a current copy of the file in the local cache, the server holding the file is located and is sent a request for a copy of the file.
2. The copy is stored in the local UNIX file system in the client computer. The copy is then opened and the resulting UNIX file descriptor is returned to the client.
3. Subsequent read, write and other operations on the file by processes in the client computer are applied to the local copy.
4. When the process in the client issues a close system call, if the local copy has been updated its contents are sent back to the server. The server updates the file contents and the timestamps on the file. The copy on the client's local disk is retained in case it is needed again by a user-level process on the same workstation.