MCA Part-II

Paper-XIII: Operating System

Topic: DISTRIBUTED FILE SYSTEM

PREPARED BY: DR. KIRAN PANDEY

(School of Computer Science)

Email-id: kiranpandey.nou@gmail.com

INTRODUCTION

A distributed system is a collection of loosely coupled computers interconnected by a communication network. These computers can share physically dispersed files by using a Distributed File System (DFS). To explain the structure of a DFS, we need to define the terms *service, server,* and *client.* A **service** is a software entity running on one or more machines and providing a particular type of function to clients. A **server** is the service software running on a single machine. A **client** is a process that can invoke a service using a set of operations that form its Sometimes a lower-level interface is defined for the actual cross-machine interaction; it is the **inter-machine interface**.

We can say that a file system provides file services to clients. A client interface for a file service is formed by a set of primitive file operations, such as create a file, delete a file, read from a file, and write to a file. The primary hardware component that a file server controls is a set of

local secondary-storage devices (usually, magnetic disks) on which files are stored and from which they are retrieved according to the clients' requests.

The performance of a DFS is measured in terms of the amount of time needed to satisfy service requests. In conventional systems, this time consists of disk-access time and a small amount of CPU-processing time. In a DFS, however, a remote access has the additional overhead attributed to the distributed structure. This overhead includes the time to deliver the request to a server, as well as the time to get the response across the network back to the client. For each direction, in addition to the transfer of the information, there is the CPU overhead of running the communication protocol software.

The fact that a DFS manages a set of dispersed storage devices is the DFS's key distinguishing feature. The overall storage space managed by a DFS is composed of different and remotely located smaller storage spaces. Usually, these constituent storage spaces correspond to sets of files. A **component unit** is the smallest set of files that can be stored on a single machine, independently from other units. All files belonging to the same component unit must reside in the same location.

NAMING AND TRANSPARENCY

Naming is a mapping between logical and physical objects. For instance, users deal with logical data objects represented by file names, whereas the system manipulates physical blocks of data stored on disk tracks. Usually, a user refers to a file by a textual name. The latter is mapped to a lower-level numerical identifier that in turn is mapped to disk blocks. This multilevel mapping provides users with an abstraction of a file that hides the details of how and where on the disk the file is stored.

In a transparent DFS, a new dimension is added to the abstraction: that of hiding where in the network the file is located. In a conventional file system, the range of the naming mapping is an address within a disk. In a DFS, this range is expanded to include the specific machine on whose disk the file is stored. Going one step further with the concept of treating files as abstractions leads to the possibility of file replication.Given a file name, the mapping returns a set of the locations of this file's replicas. In this abstraction, both the existence of multiple copies and their locations are hidden.

Naming Structures

We need to differentiate two related notions regarding name mappings in a DFS:

Location transparency: The name of a file does not reveal any hint of the file's physical storage location.

Location independence: The name of a file does not need to be changed when the file's physical storage location changes.

Both definitions relate to the level of naming discussed previously, since files have different names at different levels (that is, user-level textual names and system-level numerical identifiers). A location-independent naming scheme is a dynamic mapping, since it can map the same file name to different locations at two different times. Therefore, location independence is a stronger property than is location transparency.

Some aspects that differentiate location independence and static location transparency are:

- A file name should denote the file's most significant attributes, which are its contents rather than its location. Location-independent files can be viewed as logical data containers that are not attached to a specific storage location.

- Static location transparency provides users with a convenient way to share data. Users can share remote files by simply naming the files in a location-transparent manner, as though the files were local. Location independence promotes sharing the storage space itself, as well as the data objects. When files can be mobilized, the overall, system-wide storage space looks like a single virtual resource. A possible benefit of such a view is the ability to balance the utilization of disks across the system.

- Location independence separates the naming hierarchy from the storage-devices hierarchy and from the inter-computer structure. By contrast, if static location transparency is used (although names are transparent), we can easily expose the correspondence between component units and machines. The machines are configured in a pattern similar to the naming structure. This configuration may restrict the architecture of the system unnecessarily and conflict with other considerations.

Once the separation of name and location has been completed, clients can access files residing on remote server systems. In fact, these clients may be and rely on servers to provide all files, including the operating-

system kernel. Special protocols are needed for the boot sequence, however.

Naming Schemes

There are three main approaches to naming schemes in a DFS:

a. In the simplest approach, a file is identified by some combination of its host name and local name, which guarantees a unique system-wide name. In Ibis, for instance, a file is identified uniquely by the name *host: local-name,* where *Local-name* is a UNIX-like path. This naming scheme is neither location transparent nor location independent. Nevertheless, the same file operations can be used for both local and remote files.
b. The second approach was popularized by Sun's network file system, NFS. NFS is the file-system component of ONC+, a networking package supported by many UNIX vendors. NFS provides a means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree. Unlike early NFS versions which allowed only previously mounted remote directories to be accessed transparently today mounts are done on demand, based on a table of mount points and file-structure names. Components are integrated to support transparent sharing, although this integration is limited and is not uniform, because each machine may attach different remote directories to its tree. The resulting structure is versatile.
c. Using the third approach we can achieve total integration of the component file systems. Here, a single global name structure spans all the files in the system. Ideally, the composed file-system structure is the same as the structure of a conventional file system. In practice, however, the many special files (for example,

UNIX device files and machine-specific binary directories) make this goal difficult to attain.

To evaluate naming structures, we look at their administrative complexity. The most complex and most difficult-to-maintain structure is the NFS structure. Because any remote directory can be attached anywhere onto the local directory tree, the resulting hierarchy can be highly unstructured. If a server becomes unavailable, some arbitrary set of directories on different machines becomes unavailable. In addition, a separate accreditation mechanism controls which machine is allowed to attach which directory to its tree. Thus, a user might be able to access a remote directory tree on one client but be denied access on another client.

Implementation Techniques

Implementation of transparent naming requires a provision for the mapping of a file name to the associated location. To keep this mapping manageable, we must aggregate sets of files into component units and provide the mapping on a component-unit basis rather than on a single-file basis. UNIX-like systems use the hierarchical directory tree to provide name-to-location mapping and to aggregate files recursively into directories.

To enhance the availability of the crucial mapping information, we can use replication, local caching, or both. As we noted, location independence means that the mapping changes over time; hence, replicating the mapping makes a simple yet consistent update of this information impossible. A technique to overcome this obstacle is to introduce low-level location independent file identifier. Textual file names are mapped to lower-level file identifiers that indicate to which

component unit the file belongs. These identifiers are still location independent. They can be replicated and cached freely without being invalidated by migration of component units.

A common way to implement low-level identifiers is to use structured names. These names are bit strings that usually have two parts. The first part identifies the component unit to which the file belongs; the second part identifies the particular file within the unit. Variants with more parts are possible. The invariant of structured names, however, is that individual parts of the name are unique at all times only within the context of the rest of the parts. We can obtain uniqueness at all times by taking care not to reuse a name that is still in use, by adding sufficiently more bits (this method is used in AFS), or by using a timestamp as one part of the name.

REMOTE FILE ACCESS

Let us consider a user who requests access to a remote file. The server storing the file has been located by the naming scheme, and now the actual data transfer must take place.

One way to achieve this transfer is through a whereby requests for accesses are delivered to the server, the server machine performs the accesses, and their results are forwarded back to the user. One of the most common ways of implementing remote service is the remote procedure call (RPC) paradigm. **Remote Procedure Call** (**RPC**) is a protocol that one program can use to request a service from a program located in another computer on a network without having to understand the network's details. A procedure call is also sometimes known as a function call or a subroutine call. A direct analogy exists between disk-access methods in conventional file systems and the remote-service

method in a DFS: using the remote-service method is analogous to performing a disk access for each access request.

To ensure reasonable performance of a remote-service mechanism, we can use a form of caching. In conventional file systems, caching is used to reduce disk I/O (thereby increasing performance), whereas in DFSs, the goal is to reduce both network traffic and disk I/O. In the following discussion, we describe the implementation of caching in a DFS and contrast it with the basic remote-service paradigm.

Basic Caching Scheme

The concept of caching is simple. If the data needed to satisfy the access request are not already cached, then a copy of those data is brought from the server to the client system. Accesses are performed on the cached copy. The idea is to retain recently accessed disk blocks in the cache, so that repeated accesses to the same information can be handled locally, without additional network traffic.

A replacement policy (for example, the LRU algorithm) keeps the cache size bounded. No direct correspondence exists between accesses and traffic to the server. Files are still identified with one master copy residing at the server machine, but copies (or parts) of the file are scattered in different caches. When a cached copy is modified, the changes need to be reflected on the master copy to preserve the relevant consistency semantics. The problem of keeping the cached copies consistent with the master file is the cache-consistency problem which we will discuss later. DFS caching can be called network virtual memory. It acts similarly to demand-paged virtual memory, except that the backing store usually is not a local disk but rather a remote server. NFS allows the swap space to be mounted remotely, so it actually can implement virtual memory over a network, notwithstanding the resulting performance penalty.

Block size and total cache size are important for block-caching schemes. In UNIX-like systems, common block sizes are 4 KB and 8 KB. For large caches (over 1MB), large block sizes (over 8 KB) are beneficial. For smaller caches, large block sizes are less beneficial because they result in fewer blocks in the cache and a lower hit ratio.

Cache Location

One very important question is:

Where should the cached data be stored-on disk or in main memory?

Disk caches have one clear advantage over main-memory caches: they are reliable. Modifications to cached data are lost in a crash if the cache is kept in volatile memory. Moreover if the cached data are kept on disk, they are still there during recovery, and there is no need to fetch them again. Main-memory caches have several advantages of their own, however:

- Main-memory caches permit workstations to be diskless.
- Data can be accessed more quickly from a cache in main memory than from one on a disk.

- Technology is moving toward larger and less expensive memory. The resulting performance speedup is predicted to outweigh the advantages of disk caches.
- The server caches (used to speed up disk I/O) will be in main memory regardless of where user caches are located; if we use main-memory caches on the user machine, too, we can build a single caching mechanism for use by both servers and users.

Many remote-access implementations can be thought of as hybrids of caching and remote service.

The NFS protocol and most implementations do not provide disk caching. Recent Solaris implementations of NFS (Solaris 2.6 and beyond) include a client- side disk-caching option, the **cachefs** file system. Once the NFS client reads blocks of a file from the server it caches them in memory as well as on disk. If the memory copy is flushed, or even if the system reboots, the disk cache is referenced. If a needed block is neither in memory nor in the cachefs disk cache, an RPC is sent to the server to retrieve the block, and the block is written into the disk cache as well as stored in the memory cache for client use.

Cache Update Policy

- **Write-through:**
    - Write data through to the master disk as soon as they are placed on any cache.
    - Reliable, but poor performance.

- **Delayed-write**
    - Modifications written to the cache and then written through to the server later.
    - Fast: some data may be overwritten before they are written back, and so need never be written at all.

- Poor reliability: unwritten data will be lost whenever a user machine crashes.

Variations of the delayed-write policy differ in when modified data blocks are flushed to the server. One alternative is to flush a block when it is about to be ejected from the client's cache. This option can result in good performance, but some blocks can reside in the client's cache a long time before they are written back to the server.

A compromise between this alternative and the write-through policy is to scan the cache at regular intervals and to flush blocks that have been modified since the most recent scan, just as UNIX scans its local cache. NFS treats meta data (directory data and file-attribute data) differently. Any metadata changes are issued synchronously to the server. Thus, file-structure loss and directory-structure corruption are avoided when a client or the server crashes.

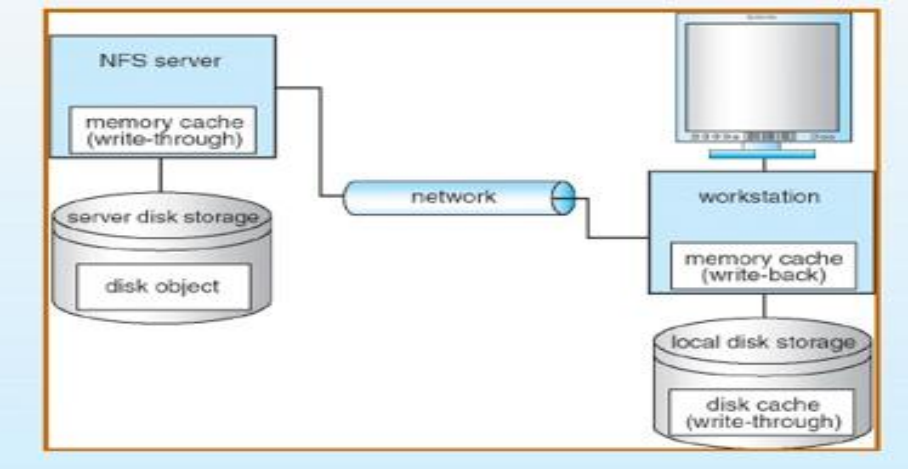

Figure 1: Caches and its use of caching.

For NFS with cachefs, writes are also written to the local disk cache area when they are written to the server, to keep all copies consistent. Thus, NFS with cachefs improves performance over standard NFS on a read request with a cachefs cache hit but

decreases performance for read or write requests with a cache miss. As with all caches, it is vital to have a high cache hit rate to gain performance. **Figure 1** shows how cachefs uses write-through and write-back caching.

Yet another variation on delayed write is to write data back to the server when the file is closed. This is used in AFS. The write-on-close policy requires the closing process to delay while the file is written through, which reduces the performance advantages of delayed writes.

Consistency

A client machine is faced with the problem of deciding whether a locally cached copy of the data is consistent with the master copy (and hence can be used). If the client machine determines that its cached data are out of date, accesses can no longer be served by those cached data. An up-to-date copy of the data needs to be cached. There are two approaches to verifying the validity of cached data:

- **Client-initiated approach:** The client initiates a validity check, in which it contacts the server and checks whether the local data are consistent with the master copy. The validity checking can range from a check before every access to a check only on first access to a file (on file open, basically). Every access coupled with a validity check is delayed, compared with an access served immediately by the cache. Alternatively, checks can be initiated at fixed time intervals. Depending on its frequency, the validity check can load both the network and the server.

- **Server-initiated approach:** The server records, for each client, the files (or parts of files) that it caches. When the server detects a potential inconsistency, it must react. A potential for inconsistency occurs when two different clients in conflicting modes cache a file. The server must be notified whenever a file is

opened, and the intended mode (read or write) must be indicated for every open. The server can then act when it detects that a file has been opened simultaneously in conflicting modes by disabling caching for that particular file.

STATEFUL VERSUS STATELESS SERVICES

There are two approaches for storing server-side information when a client accesses remote files are:

- The server tracks each file being accessed by each client. The service provided is called *stateful.*
- The server simply provides blocks as they are requested by the client without knowledge of how those blocks are used. This is called *stateless.*

The typical scenario involving a stateful file service is as follows:

A client must perform an open () operation on a file before accessing that file. The server fetches information about the file from its disk, stores it in its memory, and gives the client a connection identifier that is unique to the client and the open file. (In UNIX terms, the server fetches the inode and gives the client a file descriptor, which serves as an index to an in-core table of inodes.) This identifier is used for subsequent accesses ru1.til the session ends. A stateful service is characterized as a connection between the client and the server during a session. Either on closing the file or through a garbage-collection mechanism, the server must reclaim the main-memory space used by clients that are no longer active. The key point regarding fault tolerance in a stateful service approach is that the server keeps main-memory information about its clients. AFS is a stateful file service.

A stateless file service avoids state information by making each request self-contained. That is, each request identifies the file and the position in the file (for read and write accesses) in full. The server does not need to keep a table of open files in main memory, although it usually does so for efficiency reasons. Moreover, there is no need

to establish and terminate a connection through open() and close() operations. They are totally redundant since each file operation stands on its own and is not considered part of a session. A client process would open a file, and that open would not result in the sending of a remote message. Reads and writes would take place as remote messages (or cache lookups). The final close by the client would again result in only a local operation. NFS is a stateless file service.

The advantage of a stateful over a stateless service is increased performance. File information is cached in main memory and can be accessed easily via the connection identifier, thereby saving disk accesses. In addition, a stateful server knows whether a file is open for sequential access and can therefore read ahead the next blocks. Stateless servers cannot do so, since they have no knowledge of the purpose of the client's requests.

The distinction between stateful and stateless service becomes more evident when we consider the effects of a crash that occurs during a service activity. A stateful server loses all its volatile state in a crash. Ensuring the graceful recovery of such a server involves restoring this state, usually by a recovery protocol based on a dialog with clients. Less graceful recovery requires that the operations that were underway when the crash occurred be aborted. A different problem is caused by client failures. The server needs to become aware of such failures so that it can reclaim space allocated to record the state of crashed client processes. This phenomenon is sometimes referred to as orphan detection and elimination.

A stateless computer server avoids these problems, since a newly reincarnated server can respond to a self-contained request without any difficulty. The stateless service imposes additional constraints on the design of the DFS. First, since each request identifies the target file, a uniform, system-wide, low-level naming scheme should be used. Translating remote to local names for each request would cause even slower processing of the requests. Second, since clients retransmit requests for file operations,

these operations must be idempotent; that is, each operation must have the same effect and return the same output if executed several times consecutively. However, we must be careful when implementing destructive operations (such as deleting a file) to make them idempotent, too.

In some environments, a stateful service is a necessity. If the server employs the server-initiated method for cache validation, it cannot provide stateless service, since it maintains a record of which files are cached by which clients.

FILE REPLICATION

Replication of files on different machines in a Distributed File System is a useful redundancy for improving availability. Multi-machine replication can benefit performance too: selecting a nearby replica to serve an access request results in shorter service time.

The basic requirement of a replication scheme is that different replicas of the same file reside on failure-independent machines. That is, the availability of one replica is not affected by the availability of the rest of the replicas. This obvious requirement implies that replication management is inherently a location-opaque activity. Provisions for placing a replica on a particular machine must be available.

It is desirable to hide the details of replication from users. Mapping a replicated file name to a particular replica is the task of the naming scheme. The existence of replicas should be invisible to higher levels. At lower levels, however, the replicas must be distinguished from one another by different lower-level names. Another transparency requirement is providing replication control at higher levels. Replication control includes determination of the degree of replication and of the placement of replicas. Under certain circumstances, we may want to expose these details to users. Locus, for instance, provides users and system administrators with mechanisms to control the replication scheme.

The main problem associated with replicas is updating. From a user's point of view, replicas of a file denote the same logical entity, and thus an update to any replica must be reflected on all other replicas. More precisely, the relevant consistency semantics must be preserved when accesses to replicas are viewed as virtual accesses to the replicas' logical files. If consistency is not of primary importance, it can be sacrificed for availability and performance. In this fundamental tradeoff in the area of fault tolerance, the choice is between preserving consistency at all costs, thereby creating a potential for indefinite blocking, and sacrificing consistency under some (we hope, rare) circumstances for the sake of guaranteed progress.