**MODULE 5: DISTRIBUTED FILE SYSTEM AND CACHE COHERENCE**

**5.1 Introduction**

Let’s begin by reminding ourselves what a File is, before delving into what a Distributed File System refers to.

What is a File?

A File is a collection of data organized and understood by the user. The operating system need not understand the data within a File, the user or creator of the file does. A File system on the other hand refers to the component of an operating system that is responsible for *managing files*. Then what does *Manage File* really mean? The following actions are some examples of what the phrase means:

* Name files in meaningful ways
* Access files
* Allocate space to files
* Secure and protect files
* Administer Resource: Enforce quotas, implement priorities, etc.

So, what does Distributed File System mean?

A traditional or conventional file system typically has all of the users and all of the storage resident on the same machine. On the other hand, a distributed file system typically operates in an environment where the data may be spread out across many, many hosts on a network; and the users of the system may be equally distributed.

**5.2 Distributed File System - Definitions**

A distributed file system (DFS) is a file system in which data stored on a server is accessed by a client or clients. The data is accessed and processed as if it was stored on the local client machine. The DFS makes it possible and convenient to share data and files among users on a network in a controlled and authorized way.

In other words, a Distributed file system (DFS, also called Network File System is a method of storing and accessing files based on a client/server architecture. In a distributed file system, one or more central servers store files that can be accessed, with proper authorization rights, by any number of remote clients on the network.

It can also be mentioned that a [distributed file system](https://en.wikipedia.org/wiki/Distributed_file_system) (DFS) or network file system is any [file system](https://en.wikipedia.org/wiki/File_system) that allows access to files sharing from multiple hosts via a [computer network](https://en.wikipedia.org/wiki/Computer_network). This makes it possible for multiple users on multiple machines to share files and storage resources.

The purpose of a distributed file system (DFS) is to allow users of physically distributed computers to share data and storage resources by using a common file system. A typical configuration for a DFS is a collection of workstations and mainframes connected by a local area network (LAN). A DFS is implemented as part of the operating system of each of the connected computers.

**5.3 Why Distributed File System (DFS)?**

There are many reasons for desiring to adopt DFS. A few of the more common ones are as follows:

1. There is more storage on a DFS than what can fit on a single system
2. There is more fault tolerance than can be achieved if "*all of the eggs are in one basket*"
3. Since the user is "distributed" he can access the file system from many places, he is not restricted in accessing files.

**5.4 Distributed File System Concepts and Design**

A DFS is a distributed implementation of the classical timesharing model of a file system, where multiple users share files and storage resources. The UNIX time-sharing file system is usually regarded as the model. The purpose of a DFS is to support the same kind of sharing when users are physically dispersed in a distributed system. A distributed system is a collection of loosely-coupled machines - either a mainframe or a workstation-interconnected by a communication network. Unless specified otherwise, the network is a local area network (LAN). From the point of view of a specific machine in a distributed system, the rest of the machines and their respective resources are remote and the machine’s own resources are local.

In order to explain the structure of a DFS, some terms need to be defined. These are S*ervice*, *Server*, and *Client*. A service is a software entity running on one or more machines and providing a particular type of function to a priori unknown 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 client interface.

Using the above definitions, it suffices to say that a file system provides file services to clients with the help of a server. A client interface for a file service is formed by a set of file operations. The most primitive operations are Create a file, Delete a file, Read from a file, and Write to a file. The primary hardware component a file server controls is a set of secondary storage devices (e.g, magnetic disks) on which files are stored and from which they are retrieved according to the client’s requests. When it is said that a server, or a machine, stores a file, it means the file resides on one of its attached devices. A file system offered by a uniprocessor, timesharing operating system (e.g., UNIX 4.2 BSD) is referred to as a conventional or traditional file system.

A DFS is a file system, whose clients, servers, and storage devices are dispersed among the machines of a distributed system. Accordingly, service activity has to be carried out across the network, and instead of a single centralized data repository there are multiple and independent storage devices. As will become obvious, the concrete configuration and implementation of a DFS may vary. There are configurations where servers run on dedicated machines, as well as configurations where a machine can be both a server and a client. A DFS can be implemented as part of a distributed operating system or, alternatively, by a software layer whose task is to manage the communication between conventional operating systems and file systems. The distinctive features of a DFS are the multiplicity and autonomy of clients and servers in the system.

**5.5 Features of a Distributed File System**

**5.5.1 Transparency**

A DFS should ideally look to its clients like a conventional, centralized file system. This means the multiplicity and dispersion of servers and storage devices should be transparent to clients. Apparently, the issue of transparency in DFSs is in various dimensions. One of these is *Network Transparency* which implies that clients should be able to access remote files using the same set of file operations applicable to local files. That is, the client interface of a DFS should not distinguish between local and remote files.

Another dimension of transparency is *User Mobility*, which implies that users can log in to any machine in the system; that is, they are not forced to use a specific machine. A transparent DFS facilitates user mobility by bringing the user’s environment (e.g., home directory) to wherever he or she logs in.

Lastly, one more dimension of transparency is the *Performance* of a DFS. The most important performance measurement of a DFS is 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, a remote access has the additional overhead attributed to the distributed structure. This overhead includes the time needed to deliver the request to a server, as well as the time needed to get the response across the network back to the client. For each direction, in addition to the actual transfer of the information, there is the CPU overhead of running the communication protocol software. What is being said here regarding transparency in a DFS in a nutshell is that the performance of a DFS should be comparable to that of a conventional file system.

**5.5.2 Fault Tolerance**

Broadly speaking, Communication faults, machine failures (of type fail stop), storage device crashes, and decays of storage media are all considered to be faults that should be tolerated to some extent. A fault-tolerant system should continue functioning, perhaps in a degraded form, in the face of these failures. The degradation can be in performance, functionality, or both but should be proportional, in some sense, to the failures causing it. A system that grinds to a halt when a small number of its components fail is not fault tolerant.

**5.5.3 Scalability**

This refers to the capability of a system to adapt to increased service load. Systems have bounded or limited resources and can become completely saturated under increased load. Regarding a file system, saturation occurs, for example, when a server’s CPU runs at very high utilization rate or when disks are almost full. As for a DFS in particular, server saturation is even a bigger threat because of the communication overhead associated with processing remote requests. Scalability is a relative property; a scalable system should react more gracefully to increased load than a non-scalable one will. First, its performance should degrade more moderately than that of a non-scalable system. Second, its resources should reach a saturated state later, when compared with a non-scalable system.

In a distributed system, the ability to scale up gracefully is of special importance, since expanding the network by adding new machines or interconnecting two networks together is commonplace. In short, a scalable design should withstand high-service load, accommodate growth of the user community, and enable simple integration of added resources.

Fault tolerance and scalability are mutually related to each other. A heavily loaded component can become paralyzed and behave like a faulty component. Also, shifting a load from a faulty component to its backup can saturate the latter. Generally, having spare resources is essential for reliability, as well as for handling peak loads gracefully.

An advantage of distributed systems over centralized systems is the potential for fault tolerance and scalability because of the multiplicity of resources. Inappropriate design can, however, obscure this potential and, worse, hinder the system’s scalability and make it failure prone. Fault tolerance and scalability considerations call for a design demonstrating distribution of control and data. Any centralized entity, whether it is a central controller, administrator or a central data repository, introduces both a severe point of failure and a performance bottleneck. Therefore, a scalable and fault-tolerant DFS should have multiple and independent servers controlling multiple and independent storage devices.

**5.6 Naming and Transparency in DFS**

*Naming* is a mapping between logical and physical objects. 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, which 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 the file is actually stored on the disk. 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 name mapping is an address within a disk; in a DFS it is augmented to include the specific machine on whose disk the file is stored. Going further with the concept of treating files as abstractions leads to the notion 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.

**5.6.1 Naming Schemes**

There are three main approaches to naming schemes in a DFS. In the simplest approach, files are named by some combination of their host name and local name, which guarantees a unique system-wide name. In Ibis for instance, a file is uniquely identified by the name *host: local-name*, where local name is a UNIX-like path. This naming scheme is neither location transparent nor location independent [*Location Transparency: The name of a file does not reveal any hint as to its physical storage location; Location Independence: The name of a file need not be changed when the file’s physical storage location changes*]. Nevertheless, the same file operations can be used for both local and remote files; that is, at least the fundamental network transparency is provided. The structure of the DFS is a collection of isolated component units that are entire conventional file systems. In this first approach, component units remain isolated, although means are provided to refer to a remote file.

The second approach, made popular by Sun’s NFS, provides means for individual machines to attach (or mount as in UNIX terms) remote directories to their local name spaces. Once a remote directory is attached locally, its files can be named in a location-transparent manner. The resulting name structure is versatile; usually it is a forest of UNIX trees, one for each machine, with some overlapping (i.e., shared) subtrees. A prominent property of this scheme is the fact that the shared name space may not be identical at all the machines. Usually this is perceived as a serious disadvantage; however, the scheme has the potential for creating customized name spaces for individual machines.

In the third approach, a single global name structure that spans all the files in the system is employed. Total integration between the component file systems is achieved. In this way, the same name space is visible to all clients. Ideally, the composed file system structure should be isomorphic to the structure of a conventional file system. In practice, however, there are many special files that make the ideal goal difficult to attain. (In UNIX, for example, I/O devices are treated as ordinary files and are represented in the directory **/dev**; object code of system programs reside in the directory **/bin**. These are special files specific to a particular hardware setting.)

**5.7 Cache Coherence**

What is Cache? A **Cache** is a special storage space for temporary files that makes a device, browser, or app run faster and more efficiently. After opening an app or website for the first time, a **cache** stashes files, images, and other pertinent data on your device.

In another way, a cache is a hardware or software component that stores data so that future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere.

In its simplest form, a Cache is hardware or software that is used to store something, usually data, temporarily in a computing environment. Cache is used because bulk, or main, storage can't keep up with the demands of the cache clients. Cache shortens data access times, reduces latency and improves input/output. Because almost all application workloads depend on I/O operations, caching improves application performance.

**5.7.1 How Cache Works**

When a cache client needs to access data, it first checks the cache. When the requested data is found in a cache, it's called a *cache hit*. The percent of attempts that result in cache hits is known as the *cache hit rate* or *ratio*.

If the requested data isn't found in the cache -- a situation known as a *cache miss* -- it is pulled from main memory and copied into the cache. How this is done, and what data is ejected from the cache to make room for the new data, depends on the caching algorithm or policies the system uses.

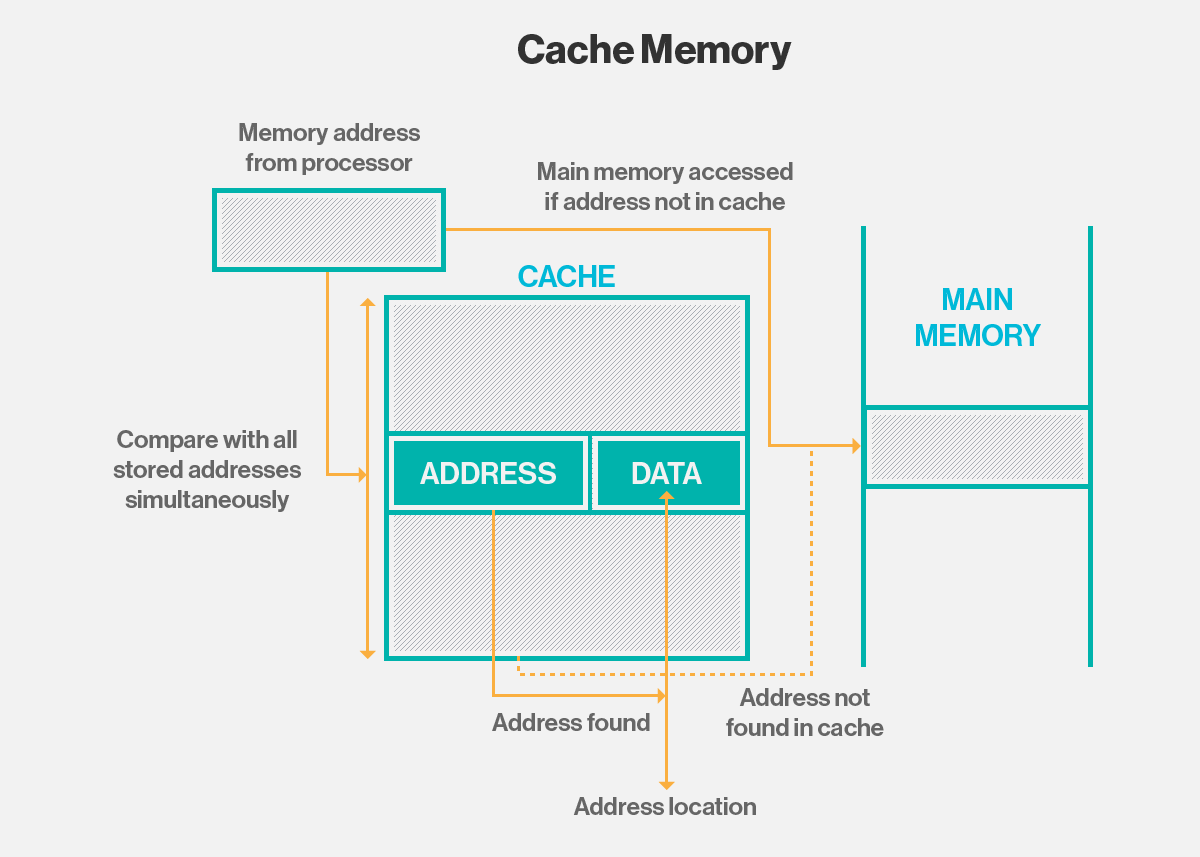


Fig. 5.1 – Working of Cache

The popular web browsers, such as Internet Explorer, Firefox, Safari and Chrome, use a browser cache to improve performance of frequently accessed webpages. When you visit a webpage, the requested files are stored in your computing storage in the browser's cache.

Clicking back and returning to a previous page enables your browser to retrieve most of the files it needs from the cache instead of having them all resent from the web server. This approach is called Read cache. The browser can read data from the browser cache much faster than it can reread the files from the webpage.

**5.7.2 Benefits of Cache**

1. The use of cache reduces latency for active data. This results in higher performance for a system or application.
2. It also diverts I/O to cache, reducing I/O operations to external storage and lower levels of Storage Area Network (SAN)\* traffic.
3. Data can stay permanently on traditional storage or external storage arrays. This maintains the consistency and integrity of the data using features provided by the array, such as snapshots or replication.
4. Flash is used only for the part of the workload that will benefit from lower latency. This results in the cost-effective use of more expensive storage.

\**A***Storage Area Network** *is a network of storage devices that can be accessed by multiple servers or computers, providing a shared pool of storage space.*

**5.7.3 Cache Algorithms**

Instructions on how the cache should be maintained are provided by cache algorithms, some of which include:

1. **Least Frequently Used (LFU): This**keeps track of how often an entry is accessed. The item that has the lowest count gets removed first.
2. **Least Recently Used (LRU): This** puts recently accessed items near the top of the cache. When the cache reaches its limit, the least recently accessed items are removed.
3. **Most Recently Used (MRU): This** removes the most recently accessed items first. This approach is best when older items are more likely to be used.

**Now to Cache Coherence, what does it really mean?**

**Cache Coherence** refers to the concept of shared resource data being stored in various local caches uniformly at the same time.

In another way, Cache Coherence refers to the uniformity of shared resource data that ends up stored in multiple local caches. When clients in a system maintain caches of a common memory resource, problems may arise with incoherent data, which is particularly the case with CPUs in a multiprocessing system.

Consider the illustration in Fig. 5.2

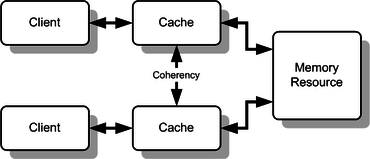


Fig. 5.2Multiple caches of some memory, which acts as a shared resource

In the figure, both clients are considered to have a cached copy of a particular memory block from a previous read. Suppose the lower client updates/changes that memory block, the upper client could be left with an invalid cache of memory without any notification of the change. Cache coherence is intended to manage such conflicts by maintaining a coherent view of the data values in multiple caches.

Explaining further, in a shared memory multiprocessor system with a separate cache memory for each processor, it is possible to have many copies of shared data: one copy in the main memory and one in the local cache of each processor that requested it. When one of the copies of data is changed, the other copies must reflect that change. Cache coherence is the process which ensures that the changes in the values of shared data are propagated throughout the system in a timely fashion.

**5.7.4 Coherence Mechanisms**

The two most common mechanisms of ensuring coherency are **Snooping** and **Directory-based**, each having their own benefits and drawbacks. Snooping-based protocols tend to be faster, if enough bandwidth is available, since all transactions are a request/response mode seen by all the processors. The drawback is that snooping is not scalable. Every request must be broadcast to all nodes in a system, meaning that as the system gets larger, the size of the bus (whether logical or physical) and the bandwidth it provides must also increase. Directory-based, on the other hand, tends to have longer latencies (with a 3-hop request/forward/respond) but use much less bandwidth since messages are point to point and not broadcast. For this reason, many of the larger systems (>64 processors) use this type of cache coherence.

Snooping is a process where the individual cache monitors address lines for access to memory locations that they have cached, whereas in a directory-based system, the data being shared is placed in a common directory that maintains the coherence between caches. The directory acts as a filter through which the processor must ask permission to load an entry from the primary memory to its cache. When an entry is changed, the directory either updates or invalidates the other caches with that entry.

Credits

Ritchie and Thompson 1974

Mitchell 1982

E. Levy and A. Silberschatz, ACM Computing Surveys, Vol. 22, No. 4, December 1990

Ellis and Floyd 1983

Barak et al. 1986

Tichy and Ruan 1984

https://searchstorage.techtarget.com/