ijecs open access www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 2 Issue 10 October, 2013 Page No. 2958-2965

AN OVERVIEW OF DISTRIBUTED FILE SYSTEM

Aditi Khazanchi, Akshay Kanwar, Lovenish Saluja

Abstract

This paper presents a distributed file system that is a client/server -based application that allows clients to access and process data stored on the server as if it were on their own computer. 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. It consists of an introduction of DFS, features of DFS, its background, its concepts, design goals and consideration. It also emphasized on Classical file system: Sun network file system and Andrew file system and a brief study of Google file system.

Introduction

A method of storing and accessing files based in client/server architecture. In a distributed file system, one or more central servers store files that can be accessed, with proper authorization rights, by any number of remote clients in the network. Much like an operating system organizes files in a hierarchical file management system; the distributed system uses a uniform naming convention and a mapping scheme to keep track of where files are located.

Distributed file systems can be advantageous because they make it easier to distribute documents to multiple clients and they provide a centralized storage system so that client machines are not using their resources to store files.

With Distributed File System (DFS), system administrators can make it easy for users to access and manage files that are physically distributed across a network. With DFS, you can make files distributed across multiple servers appear to users as if they reside in one place on the network. Users no longer need to know and specify the actual physical location of files in order to access them.

For example, if you have marketing material scattered across multiple servers in a domain, you can use DFS to make it appear as though all of the material resides on a single server. This eliminates the need for users to go to multiple locations on the network to find the information they need.

History

The Incompatible Timesharing System used virtual devices for transparent inter-machine file system access in the 1960s. More file servers were developed in the 1970s. In 1976 Digital Equipment Corporation created the File Access Listener (FAL), an implementation of the Data Access Protocol as part of DECnet Phase II which became the first widely used network file system. In 1985 Sun Microsystems created the file system called "Network File System" (NFS) which became the first widely used Internet Protocol based network file system. Other notable network file systems are Andrew File System (AFS), Apple Filing Protocol (AFP), NetWare Core Protocol (NCP), and Server Message Block (SMB) which is also known as Common Internet File System (CIFS).

Design Goals

Distributed file systems may aim for "transparency" in a number of aspects. That is, they aim to be "invisible" to client programs, which "see" a system which is similar to a local file system. Behind the scenes, the distributed file system handles locating files, transporting data, and potentially providing other features listed below.

- **Access transparency** is that clients are unaware that files are distributed and can access them in the same way as local files are accessed.
- **Location transparency** A consistent name space exists encompassing local as well as remote files. The name of a file does not give its location.
- **Concurrency transparency** All clients have the same view of the state of the file system. This means that if one process is modifying a file, any other processes on the same system or remote systems that are accessing the files will see the modifications in a coherent manner.
- **Failure transparency** The client and client programs should operate correctly after a server failure.
- **Heterogeneity** File service should be provided across different hardware and operating system platforms.
- **Scalability** The file system should work well in small environments (1 machine, a dozen machines) and also scale gracefully to huge ones (hundreds through tens of thousands of systems).
- **Replication transparency** To support scalability, we may wish to replicate files across multiple servers. Clients should be unaware of this.
- **Migration transparency** Files should be able to move around without the client's knowledge.

1. Design Consideration

- **Avoiding single point of failure**

The failure of disk hardware or a given storage node in a cluster can create a single point of failure that can result in data loss or unavailability. Fault tolerance and high availability can be provided through data replication of one sort or another, so that data remains intact and available despite the failure of any single piece of equipment. For examples, see the lists of distributed fault-tolerant file and distributed parallel fault-tolerant file systems.

- **Performance**

A common performance measurement of a clustered file system is the amount of time needed to satisfy service requests. In conventional systems, this time consists of a disk-access time and a small amount of CPU-processing time. But in a clustered file system, a remote access has additional overhead due to the distributed structure. This includes the time to deliver the request to a server, the time to deliver the response to the client, and for each direction, a CPU overhead of running the communication protocol software.

- **Concurrency**

Concurrency control becomes an issue when more than one person or client is accessing the same file or block and want to update it. Hence updates to the file from one client should not interfere with access and updates from other clients. This problem is more complex with file systems due to concurrent overlapping writes, where different writers write to overlapping regions of the file concurrently. This problem is usually handled by concurrency control or locking which may either be built into the file system or provided by an add-on protocol.

Concepts of DFS

Naming and Transparency

Naming is a mapping between logical and physical objects. In DFS, users represent a logical data object with a file name that is a textural name in user-level, but the system physically stores data blocks on disk tracks which is numerical identifiers mapped to disk blocks in system – level. These two levels mapping provides users with an abstraction of a file that hides the details of how and where the file is stored on a disk. In DFS, the location of the file in the network is added to the abstraction. In conventional file system, the range of the naming mapping is represented as an address in a disk. The file abstraction leads us to the notion of file replication. When we want to access a specific file name the mapping returns a set of locations of the file replicas. But, this abstraction allows hiding both their locations and existence of multiple copies.

There are two notion related mapping in DFS:

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 not be changed when the file's physical storage location change.

Both definitions are relative to naming of files. According to location independence is a stronger property than location transparency because it can map the same file name to different locations at two different instance of time.

File Replication

File replication is a vital concept in all distributed systems. In a system that supports replication, a file may be represented by several copies of its contents at different location. Replicas of file are useful redundancy for accessing wait and delay of files because it helps to share the loads between servers. If a server copy is not available at some point, clients may simply request that file from a different server.

Almost all distributed systems uses file replication and backup. However, we have to notice that file replication is more powerful feature because there is a key difference between those two. Since backup create checkpoint in the file system to which the system can be restored that file replication is a policy of keeping redundant copies of a file. This replication policy makes sure that files are accessible even if one or more components of DFS fail.

Replication techniques can be divided into three main classes:

Explicit replication: The client explicitly writes files to multiple servers. This approach requires explicit support from the client and does not provide transparency.

Lazy file replication: The server automatically copies files to other servers after the files are written. Remote files are only brought up to date when the files are sent to the server. How often this happens is up to the implementation and affects the consistency of the file state .

Group file replication: write requests are simultaneously sent to a group of servers. This keeps all the replicas up to date, and allows clients to read consistent file state from any replica.

Semantics of sharing

When more than one user access a file, it is crucial for the distributed file system to specify how it will behave. These semantics define how read and write operations (file accesses) are related to each other during a file session - series of file accesses between opening and closing of a file. We can use different semantics of sharing to deal with shared accesses depending on the goal. Four semantics of sharing are cited: UNIX semantics, session semantics, immutable shared file semantics and transaction-like semantics. Since UNIX semantics and session semantics are the most employed, we will focus on them.

UNIX semantics- In the UNIX semantics, each file has only a single image shared by all users. Write operations on it will instantaneously affect clients that have the same file opened. Moreover, read operations always correspond to the last previous write operation.

Session semantics- Different from UNIX semantics, session semantics does not provide a single file image. When performing a remote access on the latter, a local copy of the file is done on each client who opens it. File accesses are done locally in such a way that their modifications will only be updated when the file is opened.

Caching

In distributed file system context, files are frequently accessed, thus the system will have to deal with a great number of requests. Furthermore, the scenario can be worst if the distributed file system is relied on a remote service. In this method of access, each request becomes a message to the server and its result is also sent as a message. Instead of always using remote accesses, it is encouraged to employ caching on clients to improve performance.

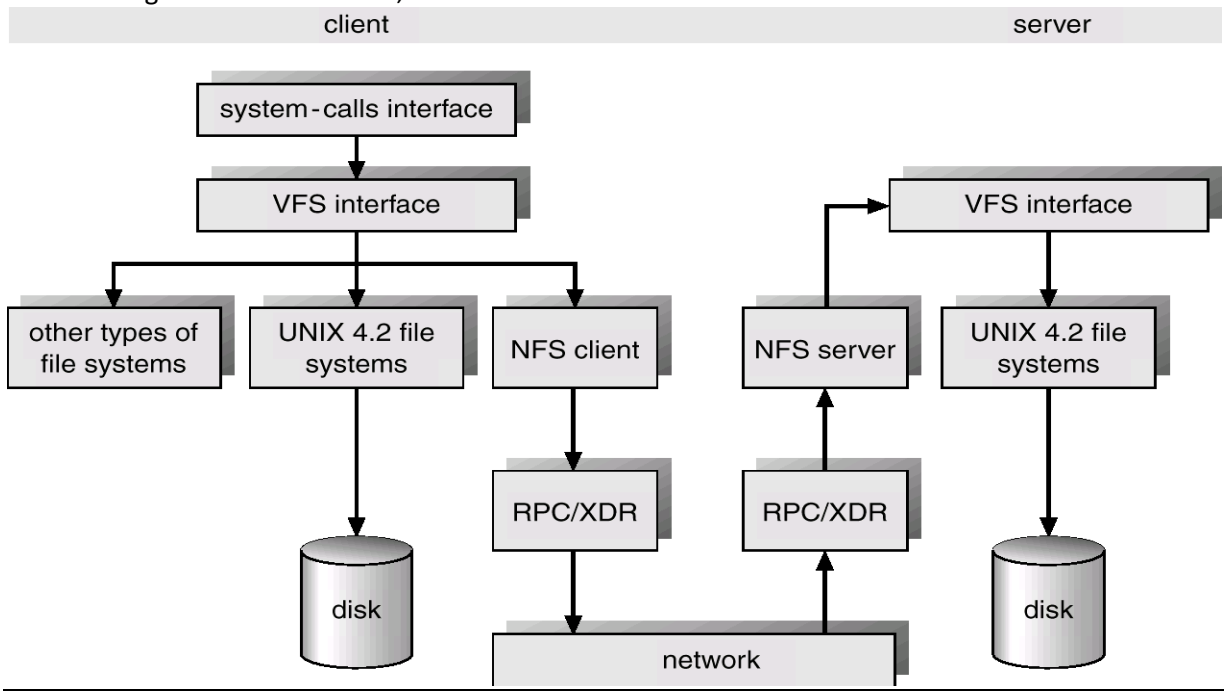
When caching is employed, a part of the file (or even the whole file) is copied. However, caching does not mean file replication. This part of the file (or block) is commonly recorded in main memory, besides other storage devices can also be used. Caching is very useful to provide fault tolerance and scalability. The caching usage is encouraged to improve performance as well as replication technique.

2. CLASSICAL DISTRIBUTED FILE SYSTEM –

NFS- Sun Network File system

Network File System (**NFS**) is a distributed file system protocol originally developed by Sun Microsystems in 1984, allowing a user on a client computer to access files over a network in a manner similar to how local storage is accessed. NFS, like many other protocols, builds on the Open Network Computing Remote Procedure Call (ONC RPC) system.

6.1.1. NFS ARCHITECTURE



6.1.2. NFS IMPLEMENTATION

- It consists of three layers:
  - *System call layer*: This handles calls like OPEN, READ, and CLOSE.
  - *Virtual file system (VFS)*: The task of the VFS layer is to maintain a table with one entry for each open file, analogous to the table of i-nodes for open files in UNIX. VFS layers have an entry, called a v-node (virtual i-node) for every open file telling whether the file is local or remote.
  - *NFS client code*: to create an r-node (remote i-node) in its internal tables to hold the file handles. The v-node points to the r-node. Each v-node in the VFS layer will ultimately contain either a pointer to an r-node in the NFS client code, or a pointer to an i-node in the local operating system. Thus from the v-node it is possible to see if a file or directory is local or remote, and if it is remote, to find its file handle.

6.1.3. PATH NAME TRANSLATION

- Break the complete pathname into components.
- For each component, do an NFS lookup using the

Component name + directory v-node

- After a mount point is reached, each component piece will cause a server access.
- Can't hand the whole operation to server since the client may have a second mount on a subsidiary directory (a mount on a mount).

- A directory name cache on the client speeds up lookups.

6.1.4 Caches of Remote Data

- The client keeps:

  File block cache - (the contents of a file)

  File attribute cache - (file header info (i-node in UNIX)).

- The local kernel hangs on to the data after getting it the first time.
- On an open, local kernel, it checks with server that cached data is still OK.
- Cached attributes are thrown away after a few seconds.
- Data blocks use read ahead and delayed write.
- Mechanism has:

  Server consistency problem

  Good performance.

6.2 Andrew File System

The **Andrew File System** (**AFS**) is a distributed network file system which uses a set of trusted servers to present a homogeneous, location-transparent file name space to all the client workstations. It was developed by Carnegie Mellon University as part of the Andrew Project. It is named after Andrew Carnegie and Andrew Project. Its primary use is in distributed computing.

6.2.1 AFS Architecture

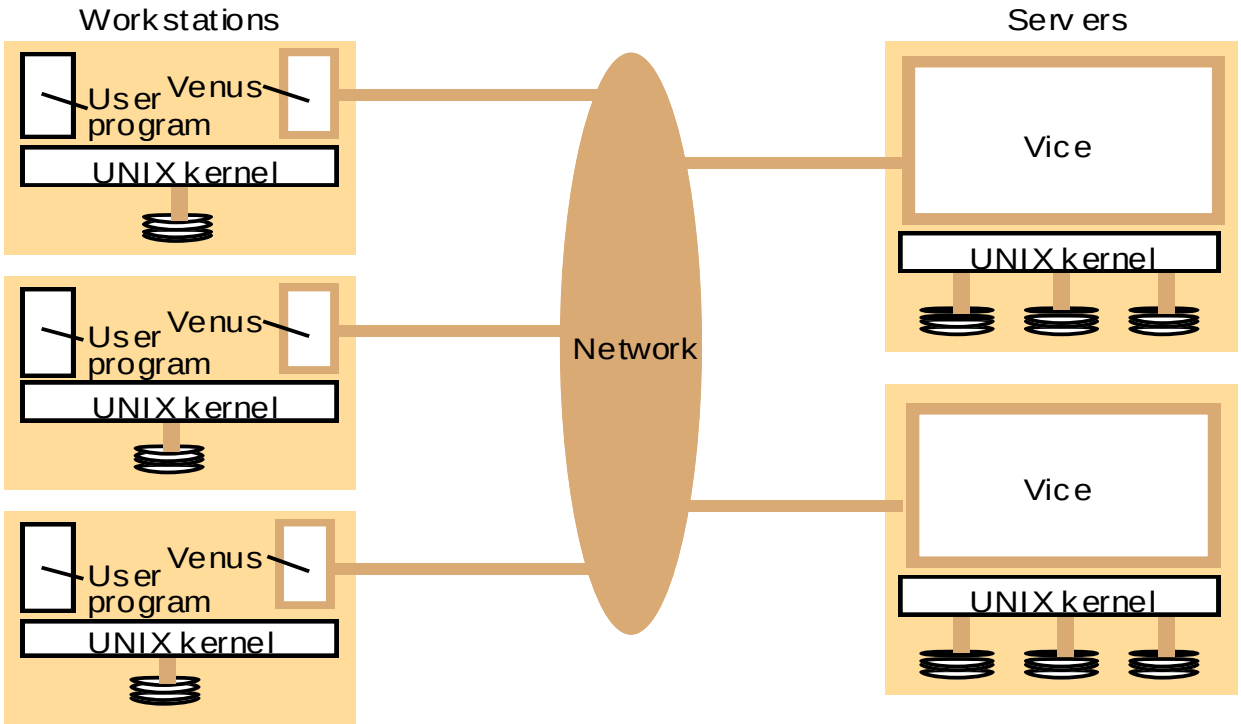


Figure .

Distribution of processes in the Andrew File System

- Like NFS, AFS provides transparent access to remote shared files for UNIX programs running on workstations.
- AFS is implemented as two software components that exist at UNIX processes called Vice and Venus.
- The files available to user processes running on workstations are either local or shared.
- Local files are handled as normal UNIX files.
- They are stored on the workstation's disk and are available only to local user processes.
- Shared files are stored on servers, and copies of them are cached on the local disks of workstations.
- The UNIX kernel in each workstation and server is a modified version of BSD UNIX.
- The modifications are designed to intercept open, close and some other file system calls when they refer to files in the shared name space and pass them to the Venus process in the client computer.

6.2.2 Shared Name Space

- The server file space is divided into volumes. Volumes contain files of only one user. It's these volumes that are the level of granularity attached to a client.
- A vice file can be accessed using a fid = <volume number, v-node >. The fid doesn't depend on machine location. A client queries a volume-location database for this information.
- Volumes can migrate between servers to balance space and utilization. Old server has "forwarding" instructions and handles client updates during migration.
- Read-only volumes (system files, etc.) can be replicated. The volume database knows how to find these.

6.2.3. File Operations And Consistency Semantics

- If a file is remote, the client operating system passes control to a client user-level process named Venus.
- The client talks to vice server only during open/close; reading/writing are only to the local copy.
- A further optimization - if data is locally cached, it's assumed to be good until the client is told otherwise.
- A client is said to have a callback on a file.
- When a client encaches a file, the server maintains state for this fact.
- Before allowing a write to a file, the server does a callback to anyone else having this file open; all other cached copies are invalidated.
- When a client is rebooted, all cached data is suspect.
- If too much storage used by server for callback state, the server can break some callbacks.
- The system clearly has consistency concerns.

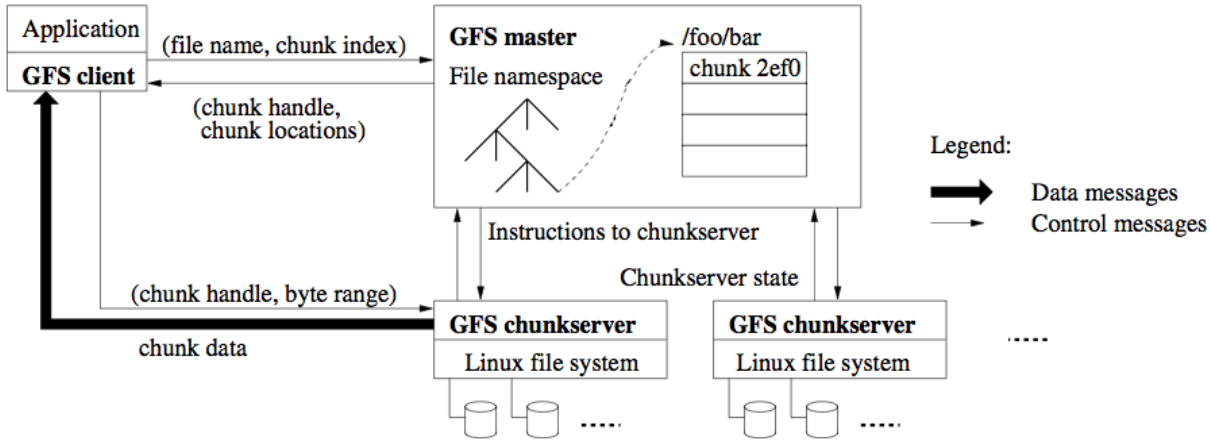
6. CASE STUDY: Google File System

Google File System (**GFS** or **GoogleFS**) is a proprietary distributed file system developed by Google for its own use. It is designed to provide efficient, reliable access to data using large clusters of commodity hardware.

7.1 GFS Key Design Goals

- **Scalability:**
  High throughput, parallel reads/writes
- **Fault tolerance** built in:
  Commodity components might fail often
  Network partitions can happen
- **Re-examine** standard I/O semantics:
  Complicated POSIX semantics *vs.* scalable primitives *vs.* common workloads
  Co-design files system and applications

7.2 ARCHITECTURE



A GFS cluster consists of a single master and multiple chunkservers and is accessed by multiple clients, as shown in Figure. Each of these is typically a commodity Linux machine running a user level server process. It is easy to run both a chunkserver and a client on the same machine, as long as machine resources permit and the lower reliability caused by running possibly flaky application code is acceptable.

Files are divided into fixed-size chunks. Each chunk is identified by an immutable and globally unique 64 bit chunk handle assigned by the master at the time of chunk creation. Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers. By default, we store three replicas, though users can designate different replication levels for different regions of the file namespace.

The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers. The master periodically communicates with each chunkserver in Heartbeat messages to give it instructions and collect its state.

GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application. Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunkservers. We do not provide the POSIX API and therefore need not hook into the Linux v-node layer.

Neither the client nor the chunkserver caches file data.

Client caches over little benefit because most applications stream through huge files or have working sets too large to be cached. Not having them simplifies the client and the overall system by eliminating cache coherence issues.(Clients do cache metadata, however.) Chunkservers need not cache file data because chunks are stored as local files and so Linux's buffer cache already keeps frequently accessed data in memory.

7. Conclusion

The DFS is one of the most important and widely used forms of shared permanent storage. 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. Architecture, Naming space, File replication, Caching, introduction to distributed file system, different file system are the key terms that are to be taken into consideration. We now understand the distributed file system, its evolution, other file systems, their architecture and implementation, different concepts of DFS, its design goals and consideration.

8. References

http://searchcio-midmarket.techtarget.com/definition/distributed-file-system

http://technet.microsoft.com/en-us/library/cc738688(v=ws.10).aspx

http://www.webopedia.com/TERM/D/distributed_file_system.html

www.cs.gsu.edu/~cscyqz/courses/aos/slides11/ch6.5-Fall11.pptx - slide ch6.5

http://en.wikipedia.org/wiki/Clustered_file_system

http://www.comp.leeds.ac.uk/mscproj/reports/1011/mista.pdf

ftp://ftp.irisa.fr/local/caps/DEPOTS/BIBLIO2008/biblio_Lage-Freitas_Andre.pdf

www.cs.rice.edu/~gw4314/lectures/dfs.ppt

http://en.wikipedia.org/wiki/Andrew_File_System

http://en.wikipedia.org/wiki/Google_File_System