Review of Distributed File Systems: Case Studies

Sunita Suralkar, Ashwini Mujumdar, Gayatri Masiwal, Manasi Kulkarni
Department of Computer Technology, Veermata Jijabai Technological Institute

Abstract

Distributed Systems have enabled sharing of data across networks. In this paper four Distributed File Systems Architectures: Andrew, Sun Network, Google and Hadoop will be reviewed with their implemented architectures, file system implementation, replication and concurrency control techniques employed. For better understanding of the file systems a comparative study is required.

Keywords— Distributed File Systems, Andrew File System, Sun Network File System, Google File System, Hadoop File System**.**

I. INTRODUCTION

File system is a subsystem of an operating system whose purpose is to organize, retrieve, store and allow sharing of data files. A Distributed File System is a distributed implementation of the classical time-sharing model of a file system, where multiple users who are geographically dispersed share files and storage resources. Accordingly, the file service activity in a distributed system has to be carried out across the network, and instead of a single centralized data repository there are multiple and independent storage devices.

The DFS can also be defined in terms of the abstract notation of a file. Permanent storage is a fundamental abstraction in computing. It consists of a named set of objects that come into existence by explicit creation, are immune to temporary failures of the system, and persist until explicitly destroyed. A file system is the refinement of such an abstraction. A DFS is a file system that supports the sharing of files in the form of persistent storage over a set of network connected nodes. The DFS has to satisfy three important requirements: Transparency, Fault Tolerance and Scalability.

II. CASE STUDY 1: ANDREW FILE SYSTEM

Andrew is a distributed computing environment being developed in a joint project by Carnegie Mellon University and IBM. One of the major components of Andrew is a distributed file system. The goal of the Andrew File System is to support growth up to at least 7000 workstations (one for each student, faculty member, and staff at Carnegie Mellon) while providing users, application programs, and system administrators with the amenities of a shared file system.

Architecture

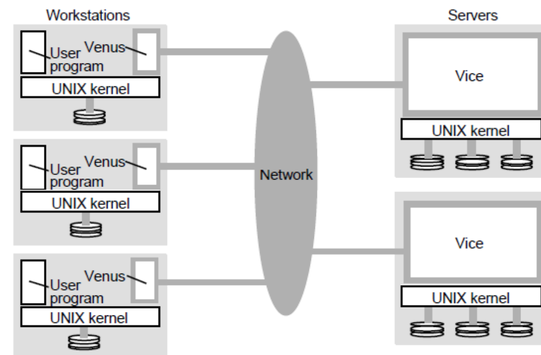

Figure 1: Andrew File System Architecture

File System

The general goal of widespread accessibility of computational and informational facilities, coupled with the choice of UNIX, led to the decision to provide an integrated, campus-wide file system with functional characteristics as close to that of UNIX as possible. The first design choice was to make the file system compatible with UNIX at the system call level.

The second design decision was to use whole files as the basic unit of data movement and storage, rather than some smaller unit such as physical or logical records. This is undoubtedly the most controversial and interesting aspect of the Andrew File System. It means that before a workstation can use a file, it must copy the entire file to its local disk, and it must write modified files back to the file system in their entirety. This in turn requires using a local disk to hold recently-used files. On the other hand, it provides significant benefits in performance and to some degree in availability. Once a workstation has a copy of a file it can use it independently of the central file system. This dramatically reduces network traffic and file server loads as compared to record-based distributed file systems. Furthermore, it is possible to cache and reuse files on the local disk, resulting in further reductions in server - loads and in additional workstation autonomy.

Two functional issues with the whole file strategy are often raised. The first concerns file sizes: only files small enough to fit in the local disks can be handled. Where this matters in the environment, the large files had to be broken into smaller parts which fit. The second has to do with updates. Modified files are returned to the central system only when they are closed, thus rendering

record-level updates impossible. This is a fundamental property of the design. However, it is not a serious problem in the university computing environment. The main application for record-level updates is databases. Serious multi-user databases have many other requirements (such as record- or field-granularity authorization, physical disk write ordering controls, and update serialization) which are not satisfied by UNIX file system semantics, even in a non-distributed environment.

The third and last key design decision in the Andrew File System was to implement it with many relatively small servers rather than a single large machine. This decision was based on the desire to support growth gracefully, to enhance availability (since if any single server fails, the others should continue), and to simplify the development process by using the same hardware and operating system as the workstations. At the present time, an Andrew. file server consists of a workstation with three to six 400-megabyte disks attached. A price/performance goal of supporting at least 50 active workstations per file server is acheived, so that the centralized costs of the file system would be reasonable. In a large configuration like the one at Carnegie Mellon, a separate "system control machine" to broadcast global information (such as where specific users' files are to be found) to the file servers is used. In a small configuration the system control machine is combined with a (the) server machine.

III. CASE STUDY 2: SUN NETWORK FILE SYSTEM

NFS views a set of interconnected workstations as a set of independent machines with independent file systems. The goal is to allow some degree of sharing among these file systems in a transparent manner. Sharing is based on server-client relationship. A machine may be, and often is, both a client and a server. Sharing is allowed between any pair of machines, not only with dedicated server machines. Consistent with the independence of a machine is the critical observation that NFS sharing of a remote file system affects only the client machine and no other machine. Therefore, there is no notion of a globally shared file system as in Locus, Sprite, UNIX United, and Andrew.

To make a remote directory accessible in a transparent manner from a client machine, a user of that machine first has to carry out a mount operation. Actually, only a superuser can invoke the mount operation. Specifying the remote directory as an argument for the mount operation is done in a nontransparent manner; the location (i.e., hostname) of the remote directory has to be provided. From then on, users on the client machine can access files in the remote directory in a totally transparent manner, as if the directory were local. Since each machine is free to configure its own name space, it is not guaranteed that all machines have a common view of the shared space. The convention is to configure the system to have a uniform name space. By mounting a shared file system over user home directories on all the machines, a user can log in to any workstation and get his or her home environment. Thus, user mobility can be provided, although again by convention.

Subject to access rights accreditation, potentially any file system or a directory within a file system can be remotely mounted on top of any local directory. In the latest NFS version, diskless workstations can even mount their own roots from servers (Version 4.0, May 1988 described in Sun Microsystems Inc. . In previous NFS versions, a diskless workstation depends on the Network Disk (ND) protocol that provides raw block I/O service from remote disks; the server disk was partitioned and no sharing of root file systems was allowed.

One of the design goals of NFS is to provide file services in a heterogeneous environment of different machines, operating systems, and network architecture. The NFS specification is independent of these media and thus encourages other implementations.

This independence is achieved through the use of RPC primitives built on top of an External Date Representation (XDR) protocol-two implementation independent interfaces [Sun Microsystems Inc. 19881. Hence, if the system consists of heterogeneous machines and file systems that are properly interfaced to NFS, file systems of different types can be mounted both locally and remotely.

Architecture

In general, Sun's implementation of NFS is integrated with the SunOS kernel for reasons of efficiency (although such integration is not strictly necessary).

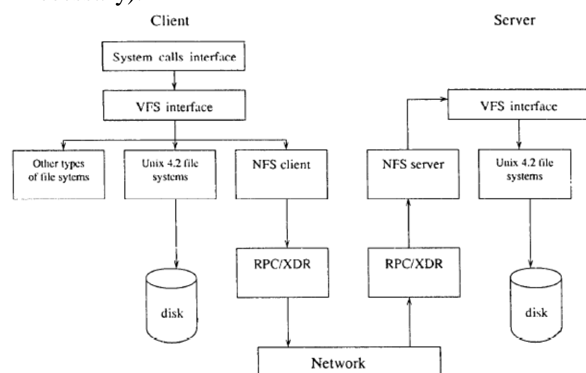

Figure 2: Schematic View of NFS Architecture

The NFS architecture is schematically depicted in Figure 6. The user interface is the UNIX system calls interface based on the Open, Read, Write, Close calls, and file descriptors. This interface is on top of a middle layer called the Virtual File System (VFS) layer. The bottom layer is the one that implements the NFS protocol and is

called the NFS layer. These layers comprise the NFS software architecture. The figure also shows the RPC/XDR software layer, local file systems, and the network and thus can serve to illustrate the integration of a DFS with all these components. The VFS serves two important functions:

It separates file system generic operations from their implementation by defining a clean interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to a variety of types of file systems mounted locally (e.g., 4.2 BSD or MS-DOS).

The VFS is based on a file representation structure called a unode, which contains a numerical designator for a file that is networkwide unique. (Recall that UNIXi- nodes are unique only within a single file system.) The kernel maintains one vnode structure for each active node (file or directory).
Essentially, for every file the vnode structures complemented by the mount table provide a pointer to its parent file system, as well as to the file system over which it is mounted. Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file system types. The VFS activates file system specific operations to handle local requests according to their file system types and calls the NFS protocol procedures for remote requests. File handles are constructed from the relevant vnodes and passed as arguments to these procedures.

As an illustration of the architecture, let us trace how an operation on an already open remote file is handled (follow the example in Figure 6). The client initiates the operation by a regular system call. The operating system layer maps this call to a VFS operation on the appropriate vnode. The VFS layer identifies the file as a remote one and invokes the appropriate NFS procedure. An RPC call is made to the NFS service layer at the remote server. This call is reinjected into the VFS layer, which finds that it is local and invokes the appropriate file system operation. This path is retraced to return the result. An advantage of this architecture is that the client and the server are identical; thus, it is possible for a machine to be a client, or a server, or both. The actual service on each server is performed by several kernel processes, which provide a temporary substitute to a LWP facility.

IV. CASE STUDY 3: GOOGLE FILE SYSTEM

The Google File System (GFS) is a proprietary DFS developed by Google. It is designed to provide efficient, reliable access to data using large clusters of commodity hardware. The files are huge and divided into chunks of 64 megabytes. Most files are mutated by appending new data rather than overwriting existing data: once written, the files are only read and often only sequentially. This DFS is best suited for scenarios in which many large files are created once but read many times. The GFS is optimized to run on computing clusters where the nodes are cheap computers. Hence, there is a need for precautions against the high failure rate of individual nodes and data loss.

Motivation for the GFS Design:
The GFS was developed based on the following assumptions:
a. Systems are prone to failure. Hence there is a need for self monitoring and self recovery from failure.
b. The file system stores a modest number of large files, where the file size is greater than 100 MB.
c. There are two types of reads: Large streaming reads of 1MB or more. These types of reads are from a contiguous region of a file by the same client. The other is a set of small random reads of a few KBs.
d. Many large sequential writes are performed. The writes are performed by appending data to files and once written the files are seldom modified.
e. Need to support multiple clients concurrently appending the same file.

Architecture

Master – Chunk Servers – Client

A GFS cluster consists of a single master and multiple chunkservers and is accessed by multiple clients. 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. Neither the client nor the chunkserver caches file data.
The GFS architecture diagram is shown below:

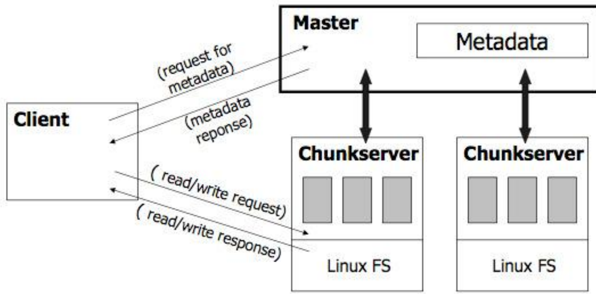

Figure 3: Google File System Architecture

Chunk Size

The GFS uses a large chunk size of 64MB. This has the following advantages:

a. Reduces clients' need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information.

b. Reduce network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time.

c. Reduces the size of the metadata stored on the master. This allows keeping the metadata in memory of master.

File Access Method

File Read

A simple file read is performed as follows:

a. Client translates the file name and byte offset specified by the application into a chunk index within the file using the fixed chunk size.

b. It sends the master a request containing the file name and chunk index.

c. The master replies with the corresponding chunk handle and locations of the replicas. The client caches this information using the file name and chunk index as the key.

d. The client then sends a request to one of the replicas, most likely the closest one. The request specifies the chunk handle and a byte range within that chunk.

e. Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened.

This file read sequence is illustrated below:

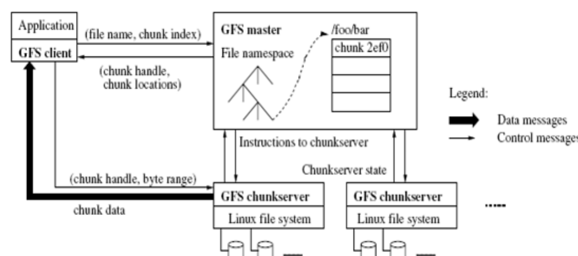

Figure 4: File Read System

File Write

The control flow of a write is given below as numbered steps:

1. Client translates the file name and byte offset specified by the application into a chunk index within the file using the fixed chunk size. It sends the master a request containing the file name and chunk index.
2. The master replies with the corresponding chunk handle and locations of the replicas
3. The client pushes the data to all the replicas. Data stored in internal buffer of chunkserver.
4. Client sends a write request to the primary. The primary assigns serial numbers to all the write requests it receives. Perform write on data it stores in the serial number order
5. The primary forwards the write request to all secondary replicas
6. The secondaries all reply to the primary on completion of write
7. The primary replies to the client.

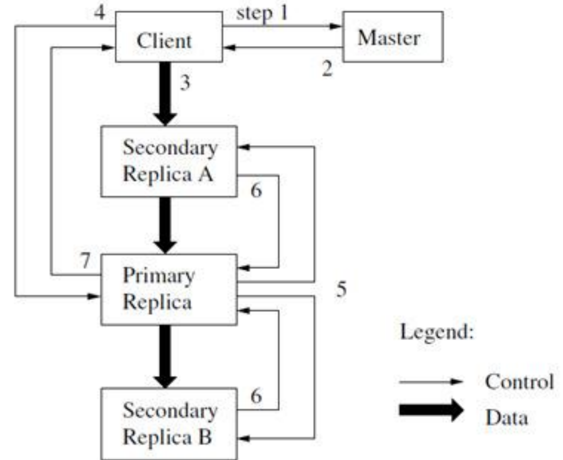

Figure 5: A File Write Sequence

Replication and Consistency

Consistency

The GFS applies mutations to a chunk in the same order on all its replicas. A mutation is an operation that changes the contents or metadata of a chunk such as a write or an append operation. It uses the chunk version numbers to detect any replica that has become stale due to missed mutations while its chunkserver was down. The chance of a client reading from a stale replica stored in its cache is small. This is because the cache entry uses a timeout mechanism. Also, it purges all chunk information for that file on the next file open.

Replication

The GFS employs both chunk replication and master replication for added reliability.

System Availability

The GFS supports Fast Recovery to ensure availability. Both the master and the chunkserver are

designed to restore their state and start in seconds no matter how they terminated. The normal and abnormal termination are not distinguished. Hence, any fault will result in the same recovery process as a successful termination.

Data Integrity

The GFS employs a checksum mechanism to ensure integrity of data being read / written. A 32 bit checksum is included for every 64KB block of chunk. For reads, the chunkserver verifies the checksum of data blocks that overlap the read range before returning any data to the requester. For writes (append to end of a chunk) incrementally update the checksum for the last partial checksum block, and compute new checksums for any brand new checksum blocks filled by the append.

Limitations of the GFS

1. No standard API such as POSIX for programming.
2. The Application / client have opportunity to get a stale chunk replica, though this probability is low.
3. Some of the performance issues depend on the client and application implementation.
4. If a write by the application is large or straddles a chunk boundary, it may be added fragments from other clients.

V. CASE STUDY 4: HADOOP FILE SYSTEM

The Hadoop is a distributed parallel fault tolerant file system inspired by the Google File System. It was designed to reliably store very large files across machines in a large cluster. Each file stored as a sequence of blocks; all blocks in a file except the last block are the same size. Blocks belonging to a file are replicated for fault tolerance. The block size and replication factor are configurable per file. The files are "write once" and have strictly one writer at any time. This DFS has been used by Facebook and Yahoo.

Architecture

The file system metadata and application data stored separately. The Metadata stored on a dedicated server called the NameNode. Application data are stored on other servers called DataNodes. All servers are fully connected and communicate with each other using TCP-based protocols. File content is split into large blocks (typically 128MB) and each block of the file is independently replicated at multiple DataNodes for reliability.

Name Node

The files and directories represented by inodes, which record attributes like permissions, modification and access times, namespace and disk space quotas. The metadata comprising the inode data and the list of blocks belonging to each file is called the Image. Checkpoints are the persistent record of the image stored in the local host's native files system. The modification log of the image stored in the local host's native file system is referred to as the Journal. During restarts the NameNode restores the namespace by reading the namespace and replaying the journal.

Data Node

Each block replica on a DataNode is represented by two files in the local host's native file system. The first file contains the data itself and the second file is block's metadata including checksums for the block data and the block's generation stamp. The size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in traditional file systems. Thus, if a block is half full it needs only half of the space of the full block on the local drive.
During startup each DataNode connects to the NameNode and performs a handshake. The purpose of the handshake is to verify the namespace ID and the software version of the DataNode. If either does not match that of the NameNode the DataNode automatically shuts down. A DataNode identifies block replicas in its possession to the NameNode by sending a block report. A block report contains the block id, the generation stamp and the length for each block replica the server hosts. The first block report is sent immediately after the DataNode registration. Subsequent block reports are sent every hour and provide the NameNode with an up-to date view of where block replicas are located on the cluster.

File Access Method

File Read

When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. It then contacts a DataNode directly and requests the transfer of the desired block.

File Write

When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data. When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file. Each choice of DataNodes is likely to be different.

Synchronization

The Hadoop DFS implements a single-writer, multiple-reader model. The Hadoop client that opens a file for writing is granted a lease for the file; no other client can write to the file. The writing client periodically renews the lease. When the file is closed, the lease is revoked. The writer's lease does

not prevent other clients from reading the file; a file may have many concurrent readers.
The interactions involved are shown in the figure below:

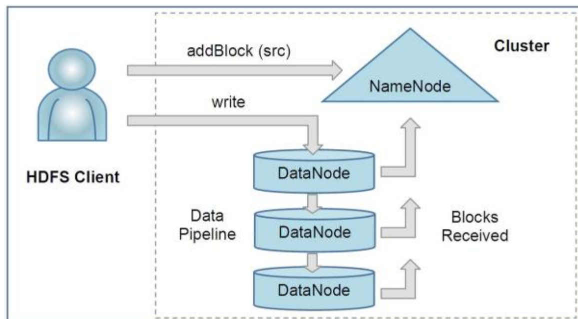

Figure 4: File Access in Hadoop

Replication Management

The blocks are replicated for reliability. Hadoop has a method to identify and overcome the issues of under-replication and over- replication. The default number of replicas for each block is 3. NameNode detects that a block has become under- or over-replicated based on DataNode's block report. If over replicated, the NameNode chooses a replica to remove. Preference is given to remove from the DataNode with the least amount of available disk space. If under-replicated, it is put in the replication priority queue. A block with only one replica has the highest priority, while a block with a number of replicas that is greater than two thirds of its replication factor has the lowest priority. It also ensures that not all replicas of a block are located on same physical location.

Consistency

The Hadoop DFS use checksums with each data block to maintain data consistency. The checksums are verified by the client while reading to help detect corruption caused either by the client, the DataNodes, or network. When a client creates an HDFS file, it computes the checksum sequence for each block and sends it to a DataNode along with the data. DataNode stores checksums in a metadata file separate from the block's data file. When HDFS reads a file, each block's data and checksums are returned to the client.

Limitations of Hadoop DFS

1. Centralization: The Hadoop system uses a centralized master server. So, the Hadoop cluster is effectively unavailable when its NameNode is down. Restarting the NameNode has been a satisfactory recovery method so far and steps being taken towards automated recovery.
2. Scalability: Since the NameNode keeps all the namespace and block locations in memory, the size of the NameNode heap limits number of files and blocks addressable. One solution is to allow multiple namespaces (and NameNodes) to share the physical storage within a cluster.

VI. COMPARISON OF FILE SYSTEMS

File System	AFS	NFS	GFS	Hadoop
Architecture	symmetric	symmetric	Clustered-based, asymmetric, parallel, objectbased	Clustered-based, asymmetric, parallel, objectbased
Processes	Stateless	Stateless	Stateful	Stateful
Communication	RPC/ TCP	RPC/ TCP or UDP	RPC/TCP	RPC/TCP & UDP
Naming	-	-	Central metadata server	Central metadata server
Synchronization	Callback promise	Read-ahead, delayed -write	Write-once-read-many, Multiple producer/Single consumer, Give locks on objects to clients, using leases	Write-once-read-many, give locks on objects to clients, using leases
Consistency and Replication	Callback mechanism	One copy semantics, read only file stores can be replicated	Server side replication, Asynchronous replication, checksum, relax consistency Among replications of data objects	Server side replication, Asynchronous replication, Checksum
Fault Tolerance	Failure as norm	Failure as norm	Failure as norm	Failure as norm

Table 1: Comparision of AFS, NFS, GFS and Hadoop

VII. CONCLUSION

In this paper the Distributed Systems implementations of Andrew, Sun Network, Google and Hadoop are reviewed with their Architecture, File System implementation, replication and concurrency control techniques. The filesystems are also compared based on their implementation.

REFERENCES

[1] Howard J. "An Overview of the Andrew File System".
[2] Sandberg R., Goldberg D., Kleiman S., Walsh D., Lyon B., "Design and Implementation of the Sun Network Filesystem".
[3] Ghemawat S., Gobioff H., Leung S., "The Google File System".
[4] Shvacko K., Kunang H., Radia S., Chansler R., "The Hadoop Distributed File System", IEEE 2010.
[5] Dean J., Ghemawat S., "MapReduce: Simplified Data Processing on Large Clusters", OSDI 2004.
[6] Keerthivasan M., "Review of Distributed File Systems: Concepts and Case Studies", ECE 677 Distributed Computing Systems