Distributed File Systems

Paul Krzyzanowski
Rutgers University

November 2017

1 Introduction

The classic network file systems we examined, NFS, CIFS, AFS, Coda, were designed as client-server applications. Clients talk to a service that, in this case, happens to be a file service. The actual file service is offered by a single machine. Where multiple machines are used, the administrator is responsible for diving up file resources (e.g., home directories) among individual servers and the the entire resource (e.g., your home directory) is fully served from a single server. In Coda, which supports replication, it is up to the administrator to define the set of machines that hold volume replicas.

This is not the kind of approach that we want to use if we want to divide our files among thousands or tens of thousands of machines. For an example of handling this environment, we will look at two closely-related file systems: the **Google File System** (**GFS**) and the **Hadoop Distributed File System** (**HDFS**). The latter is an open source version (and minor variant) of the former.

2 Design principles

File systems need to be designed with some knowledge of how they will be used. It is pointless, for example, to offer a file system with session semantics when concurrent file updates are the norm. GFS and HDFS are not designed as replacements for normal file systems and are specifically built for handling batch processing on very large data sets. For GFS and HDFS, the following assumptions are made:

- High availability. The cluster can contain thousands of file servers and we have to expect some of them to be down at any point in time.
- Servers are distributed among racks and data centers.
- We prefer high throughput over low latency. The file system is not designed for interactive operations.
- The system should support lots of bit files. Files are likely to range in size from many gigabytes to many terabytes. The file system should support a large number (millions) of such files.

- Access to files is write-once, read-many. Files, once created, are not modified. However, we need to support append operations and allow file contents to be visible even while a file is being written. When files are read, they are often read sequentially rather than randomly.
- Communication is reliable among working machines. TCP/IP is used with an RPC communication abstraction.

In the case of HDFS, portability across operating systems and processor platforms is a design goal. THe entire system is written in Java to achieve this.

When dealing with a tremendous amount of data, one consideration is that it may be easier to move the computation (the program) rather than stream the data from afar. Keeping the processing close to the data improves throughput, reduces latency, and reduces the overall use of network resources (e.g., routers and switches between racks).

3 Google File System (GFS)

The Google File System was designed to provide a fault tolerant file system for an environment of thousands of machines. Multi-gigabyte and multi-terabyte files are the norm for the environment and it does not make sense to design a file system that is optimized for smaller files.

In addition to the aforementioned design assumptions, the assumption is that appends to files will be large and that hundreds of processes may append to the same file concurrently.

3.1 Interface

GFS does not provide a file system interface at the operating-system level (e.g., under the VFS layer). As such, file system calls are not used to access it. Instead, a user-level API is provided. GFS is implemented as a set of user-level services that store data onto native Linux file systems. Moreover, since GFS was designed with special considerations in mind, it does not support all the features of POSIX (Linux, UNIX, OS X, BSD) file system access. It provides a familiar interface of files organized in directories with basic *create*, *delete*, *open*, *close*, *read*, and *write* operations. In addition, two special operations are supported. A **snapshot** is an efficient way of creating a copy of the current instance of a file or directory tree. An **append** operation allows a client to append data to a file as an atomic operation without having to lock a file. Multiple processes can append to the same file concurrently without fear of overwriting one another's data.

3.2 Configuration

A file in GFS is broken up into multiple fixed-size chunks. Each chunk is 64 MB. The set of machines that implements an instance of GFS is called a **GFS cluster**. A GFS cluster consists of one **master** and many **chunkservers**. The master is responsible for storing all the metadata for the files in GFS. This includes their names, directories, and the mapping of files to the list of chunks that contain each file's data. The chunks themselves are stored on the chunkservers. For fault tolerance, chunks are replicated onto multiple systems. Figure 1 shows how a file is broken into chunks and distributed among multiple chunkservers.

The Google File System is a core part of the Google Cluster Environment. This environment has GFS and a cluster scheduling system for dispatching processes as its core services.

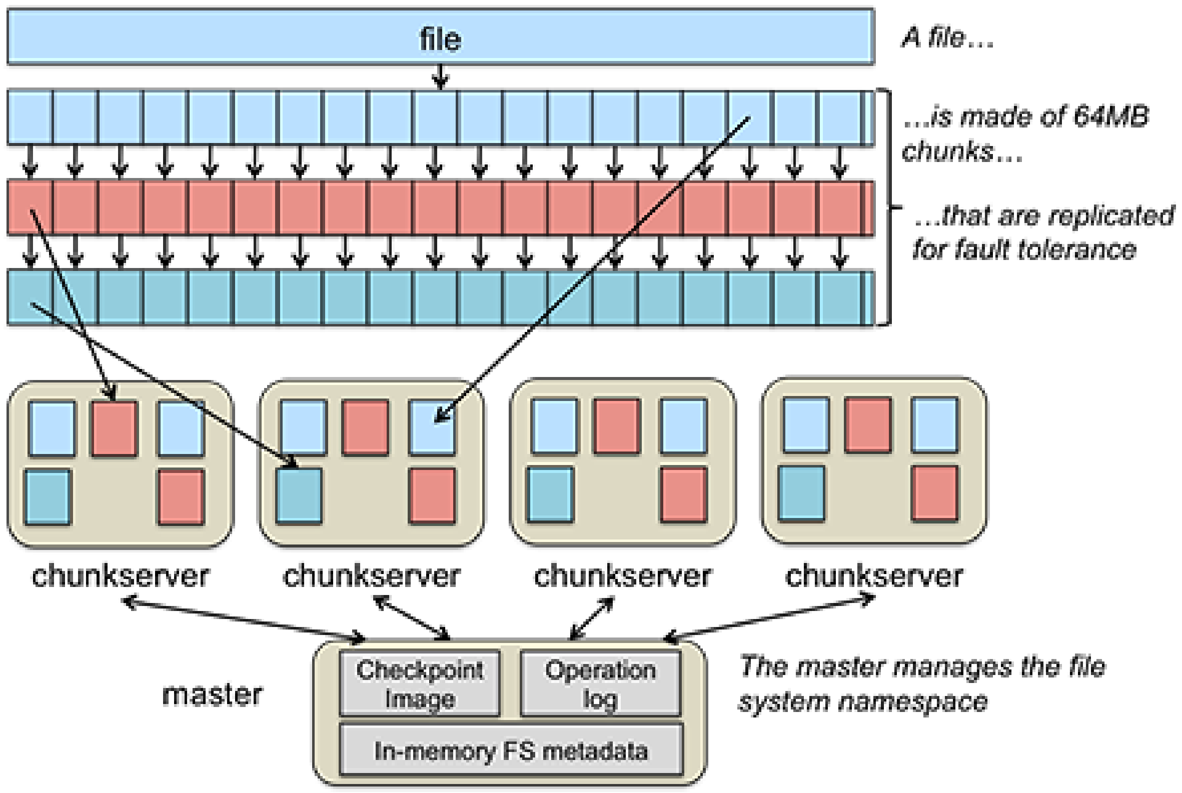
file
A file...
...is made of 64MB chunks...
...that are replicated for fault tolerance
chunkserver
chunkserver
chunkserver
chunkserver
master
Checkpoint Image
Operation log
In-memory FS metadata
The master manages the file system namespace

Figure 1: Figure 1. GFS data chunking and distribution

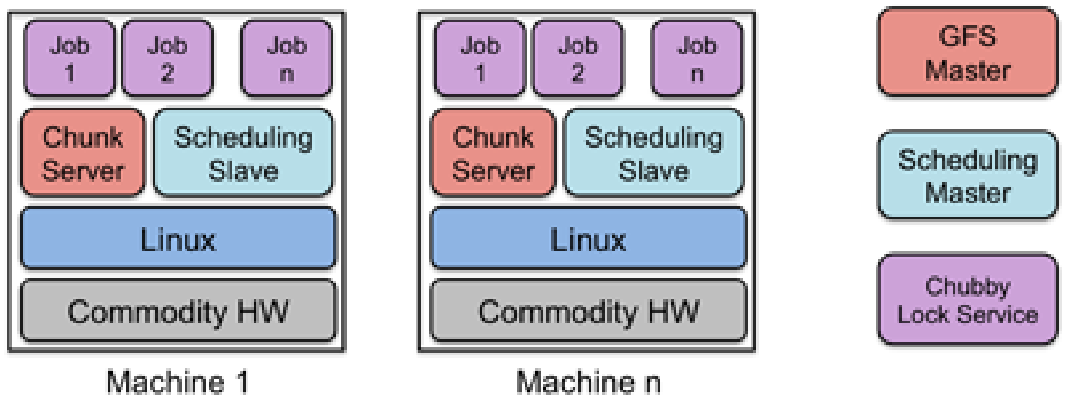

Figure 2: Figure 2. Google Cluster Environment

Typically, hundreds to thousands of active jobs are run. Over 200 clusters are deployed, many with thousands of machines. Pools of thousands of clients access the files in this cluster. The file systems exceed four petabytes and provide reaa/write loads of 40 GB/s. Jobs are often on the same machines that implement GFS, which are commodity PCs running Linux. The environment is shown in Figure 2.

3.3 Client interaction model

Since the GFS client is not implemented in the operating system at the VFS layer, GFS client code is linked into each application that uses GFS. This library interacts with the GFS master for all metadata-related operations (looking up files, creating them, deleting them, etc.). For accessing data, it interacts directly with the chunkservers that hold that data. This way, the master is not a point of congestion. Except for caching within the buffer cache on the chunkservers, neither clients nor chunkservers cache a file's data. However, client programs do cache the metadata for an open file (for example, the location of a file's chunks). This avoids additional traffic to the master.

3.4 Implementation

Each chunkserver stores chunks. A chunk is identified by a **chunk handle**, which is a globally unique 64-bit number that is assigned by the master when the chunk is first created. On the chunkserver, every chunk is stored on the local disk as a regular Linux file. For integrity, the chunkserver stores a 32-bit checksum for each chunk (and logged to disk) for each chunk on that chunkserver.

Every chunk is replicated onto multiple chunkservers. By default, there are three replicas of a chunk althrough different levels can be specified on a per-file basis. Files that are accessed by lots of processes may need more replicas to avoid congestion at any server.

3.5 Master

The primary role of the master is to maintain all of the file system metadata. This include the names and directories of each file, access control information, the mapping from each file to a set of chunks, and the current location of chunks on chunkservers. Metadata is stored only on the master. This simplifies the design of GFS as there is no need to handle synchronizing information for a changing file system among multiple masters.

For fast performance, all metadata is stored in the master's main memory. This includes the entire filesystem namespace as well as all the name-to-chunk maps. For fault tolerance, any changes are written to the disk onto an **operation log**. This operation log is also replicated onto remote machines. The operation log is similar to a journal. Every operation to the file system is logged into this file. Periodic checkpoints of the file system state, stored in a B-tree structure, are performed to avoid having to recreate all metadata by playing back the entire log.

Having a single master for a huge file system sounds like a bottleneck but the role of the master is only to tell clients which chunkservers to use. The data access itself is handled between the clients and chunkservers.

The file system namespace of directories and file names is maintained by the master. Unlike most file systems, there is no separate directory structure that contains the names of all the files within that directory. The namespace is simply a single lookup table that contains pathnames (which can look like a directory hierarchy if desired) and maps them to metadata. GFS does not support hard links or symbolic links.

The master manages chunk leases (locks on chunks with expiration), garbage collection (the freeing of unused chunks), and chunk migration (the movement and copying of chunks to different chunkservers). It periodically communicates with all chunkservers via heartbeat messages to get the state of chunkservers and sends commands to chunkservers. The master *does not* store chunk locations persistently on its disk. This information is obtained from queries to chunkservers and is done to keep consistency problems from arising.

The master is a single point of failure in GFS and replicates its data onto backup masters for fault tolerance.

3.6 Chunk size

The default chunk size in GFS is 64MB, which is a lot bigger than block sizes in normal file systems (which are often around 4KB). Small chunk sizes would not make a lot of sense for a file system designed to handle huge files since each file would then have a map of a huge number of chunks. This would greatly increase the amount of data a master would need to manage and increase the amount of data that would need to mbe communicated to a client, resulting in extra network traffic. A master stores less than 64 bytes of metadata for each 64MB chunk. By using a large chunk size, we reduce the need for frequent communication with the master to get chunk location information. It becomes feasible for a client to cache all the information related to where the data of large files is located. To reduce the risk of caching stale data, client metadata caches have timeouts. A large chunk size also makes it feasible to keep a TCP connection open to a chunkserver for an extended time, amortizing the time of setting up a TCP connection.

3.7 File access

To read a file, the client contacts the master to read a file's metadata; specifically, to get the list of chunk handles. It then gets the location of each of the chunk handles. Since chunks are replicated, each chunk handle is associated with a list of chunkservers. The client can contact any available chunkserver to read chunk data.

File writes are expected to be far less frequent than file reads. To write to a file, the master grants a **chunk lease** to one of the replicas. This replica will be the **primary replica** chunkserver and will be the first one to get updates from clients. The primary can request lease extensions if needed. When the master grants the lease, it increments the chunk version number and informs all of the replicas containing that chunk of the new version number.

The actual writing of data is split into two phases: **sending** and **writing**.

1. First, the client is given a list of replicas that identifies the primary chunkserver and secondaries. The client sends the data to the closest replica chunkserver. That replica forwards the data to another replica chunkserver, which then forwards it to yet another replica, and so on. Eventually all the replicas get the data, which is not yet written to a file but sits in a cache.

2. When the client gets an acknowledgement from all replicas that the data has been received it then sends a **write** request to the primary, identifying the data that was sent in the previous phase. The primary is responsible for serialization of writes. It

assigns consecutive serial numbers to all *write* requests that it has received, applies the *writes* to the file in serial-number order, and forwards the *write* requests in that order to the secondaries. Once the primary gets acknowledgements from all the secondaries, the primary responds back to the client and the *write* operation is complete.

The key point to note is that **data flow** is different from **control flow**. The data flows from the client to a chunkserver and then from that chunkserver to another chunkserver, and from that other chunkserver to yet another one until all chunkservers that store replicas for that chunk have received the data. The control (the *write* request) flow goes from the client to the primary chunkserver for that chunk. The primary then forwards the request to all the secondaries. This ensures that the primary is in control of the order of writes even if it receives multiple *write* requests concurrently. All replicas will have data written in the same sequence. Chunk version numbers are used to detect if any replica has stale data that was not updated because that chunkserver was down during some update.

4 Hadoop Distributed File System (HDFS)

4.1 File system operations

The Hadoop Distributed File System is inspired by GFS. The overall architecture is the same, although some terminology changes.

GFS Name	HDFS Name
Master	NameNode
Chunkserver	DataNode
chunk	block
Checkpoint image	FsImage
Operation log	EditLog

The file system provides a familiar file system interface. Files and directories can be created, deleted, renamed, and moved and symbolic links can be created. However, there is no goal of providing the rich set of features available through, say, a POSIX (Linux/BSD/OS X/Unix) or Windows interface. That is, synchronous I/O, byte-range locking, seek-and-modify, and a host of other features may not be supported. Moreover, the file system is provided through a set of user-level libraries and not as a kernel module under VFS. Applications have to be compiled to incorporate these libraries.

A file is made up of equal-size data blocks, except for the last block of the file, which may be smaller. These data blocks are stored on a collection of servers called **DataNodes**. Each block of a file may be replicated on multiple DataNodes for high availability. The block size and replication factor is configurable per file. DataNodes are responsible for storing blocks, handling read/write requests, allocating and deleting blocks, and accepting commands to replicate blocks on another DataNode. A single **NameNode** is responsible for managing the name space of the file system and coordinating file access. It stores keeps track of which block numbers belong to which file and implements open, close, rename, and move operations on files and directories. All knowledge of files and directories resides in the **NameNode**. See Figure 3.

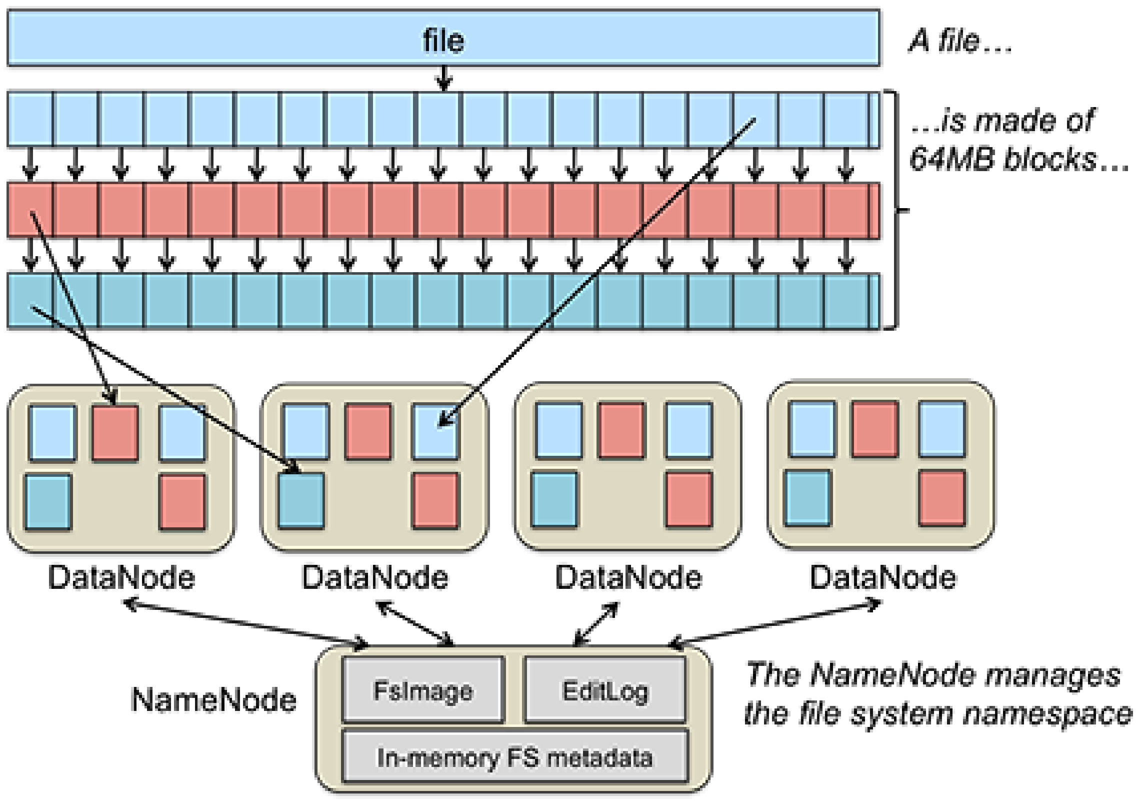

Figure 3: Figure 3. HDFS data chunking

4.2 Heartbeating and Replication

DataNodes periodically send a **heartbeat message** and a **block report** to the NameNode. The heartbeat informs the NameNode that the DataNode is functioning. The block report contains a list of all the blocks on that DataNode. A block is considered **safely replicated** if the minimum number of replica blocks have been sent by block reports from all available DataNodes. The NameNode waits for a configured percentage of DataNodes to check in and then waits an additional 30 seconds. After that time, if any data blocks do not have their minimum number of replicas, the NameNode sends replica requests to DataNodes, asking them to create replicas of specific blocks.

The system is designed to be rack-aware and data center-aware in order to improve availability and performance. What this means is that the NameNode knows which DataNodes occupy the same rack and which racks are in one data center. For performance, it is desirable to have a replica of a data block in the same rack. For availability, it is desirable to have a replica on a different rack (in case the entire rack goes down) or even in a different data center (in case the entire data center fails). HDFS supports a pluggable interface to support custom algorithms that decide on replica placement. In the default case of three replicas, the first replica goes to the local rack and both the second and third replicas go to the same remote rack.

The NameNode chooses a list of DataNodes that will host replicas of each block of a file.

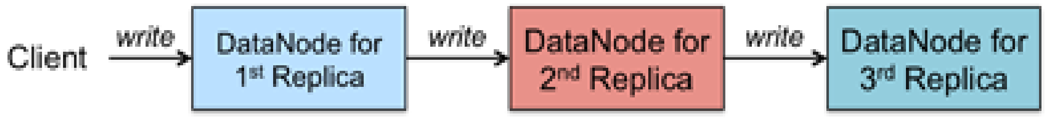

Figure 4: Figure 4. HDFS data replication

A client writes directly to the first replica. As the first replica gets the data from the client, it sends it to the second replica even before the entire block is written (e.g., it may get 4 KB out of a 64 MB block). As the second replica gets the data, it sends it to the third replica, and so on (see Figure 4).

4.3 Implementation

The NameNode contains two files: EditLog and FsInfo. The **EditLog** is a persistent record of changes to any HDFS metadata (file creation, addition of new blocks to files, file deletion, changes in replication, etc.). It is stored as a file on the server's native file system.

The **FsInfo** file stores the entire file system namespace. This includes file names, their location in directories, block mapping for each file, and file attributes. This is also stored as a file on the server's native file system.

The entire active file system image is kept in memory. On startup, the NameNode reads FsInfo and applies the list of changes in EditLog to create an up-to-date file system image. Then, the image is flushed to the disk and the EditLog is cleared. This sequence is called a **checkpoint**. From this point, changes to file system metadata are logged to EditLog but FsInfo is not modified until the next checkpoint.

On DataNodes, each block is stored as a separate file in the local file system. The DataNode does not have any knowledge of file names, attributes, and associated blocks; all that is handled by the NameNode. It simply processes requests to create, delete, write, read blocks, or replicate blocks. Any use of directories is done strictly for local efficiency - to ensure that a directory does not end up with a huge number of files that will impact performance.

To ensure data integrity, each HDFS file has a separate checksum file associated with it. This file is created by the client when the client creates the data file/. Upon retrieval, if there is a mismatch between the block checksum and the computed block checksum, the client can request a read from another DataNode.

5 References

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, The Google File System. Google, SOSP'03, October 19–22, 2003.
- Google File System, Wikipedia article.
- Robin Harris, Google File System Eval: Part 1, StorageMojo blog.
- HDFS Architecture Guide, Apache Hadoop project, December 4, 2011.

This is an updated version of the original that was published on October, 2012.