GFS from Scratch

Ge Bian, Niket Agarwal, Wenli Looi
https://github.com/looi/CS244B

Dec 2017

Abstract

GFS from Scratch is our partial re-implementation of GFS, the Google File System. Like GFS, our system features a single master that stores only metadata and a group of chunkservers that store the actual file data. The master is able to make intelligent chunk placement decisions and automatically re-replicate chunks when chunkservers fail. We describe the design and implementation of our system, explain how it is used from a client's perspective, and report measurements from benchmarks that compare our system to raw disk performance.

1. Introduction

Google File System (GFS) is a large, distributed file system designed to provide fault tolerance while running on inexpensive commodity hardware [1]. It was widely deployed in Google and used extensively until it was later replaced by the Colossus file system [2]. Google, however, has not released many details on Colossus and we have thus decided to partially re-implement GFS.

2. Design and Implementation

GFS from Scratch is based on GFS, so here we focus on the key design points and how our system differs from GFS. More detailed rationale for the design can be found in the GFS paper [1].

Our system is implemented in about 3,000 lines of C++ using the gRPC framework and protocol buffers for inter-process communication.

A cluster in GFS from Scratch consists of a single master and multiple chunkservers, as drawn in Figure 1. Files are divided into 64MB fixed sized chunks and each chunk is identified by an immutable 64-bit chunkhandle assigned by the master upon chunk creation.

2.1. Master

The single master maintains metadata on all files in the system. Its involvement in reads and writes is minimized because clients contact chunkservers directly for reads and writes (and cache this information). Metadata is stored in an SQLite database which provides atomicity and durability. One table stores the list of filenames, which are internally associated with an int64 file ID. Another table stores the chunks associated with each file.

2.2. Chunkserver

Chunkservers are responsible for storing copies of chunks as assigned by the master. They communicate directly with clients during read and write operations, preventing the master from becoming a bottleneck. Chunks are stored as regular files on the Linux filesystem. Each chunk is 64MB, although it is allocated lazily so that

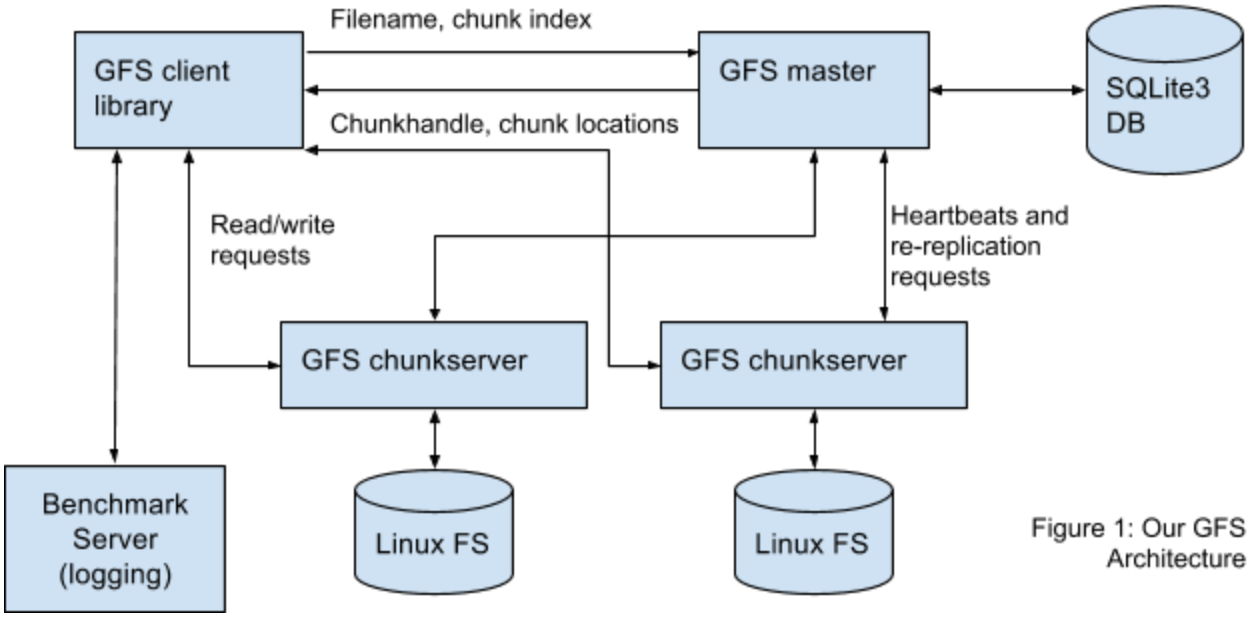


Figure 1: Our GFS Architecture

smaller chunks do not need to use 64MB of disk space. The master identifies chunks through their "IP:port" combination.

2.3. Client library

GFS does not implement the Unix/Linux file system interface and instead provides a C++ client library. The client library maintains and caches gRPC connections to the master and chunkservers, and provides various functions:

- **FindMatchingFiles:** Lists files matching a given prefix. Since GFS filenames contain the entire folder name as a prefix (such as "a/b/c.txt"), this function can be used to list directories.
- **Read:** Read a section of a file as a byte array.
- **Write:** Write a section of a file as a byte array. Creates the file if it does not already exist.
- **Append:** Atomically appends data to the end of the chunk. If the chunk doesn't enough space to fit the appended data, the chunkserver *pads* 0s and the client retries the next chunkhandle.
- **Move:** Change a file's name. This is currently handled as a master-only operation. Due to caching, clients may be able to access files using their old name for a short time.
- **Delete:** Delete a file. The file is immediately deleted at the master, unlike GFS where it is first renamed. The master periodically identifies chunks that are no longer referenced and instructs chunkservers holding them to delete the chunks.

2.4. Client application

We also implemented a client application based on the client library. The client application allows a users to interactively access the filesystem and also implements the benchmarks featured in this paper.

2.5. Main RPCs

Here, we list the main RPCs implemented. They are based on RPCs in the original GFS paper:

2.5.1. Client → Master RPCs

- **FindMatchingFiles:** For a given filename prefix, returns a list of filenames matching the prefix.
- **FindLocations:** For a read operation, returns the chunkservers for a given filename and chunk index. Chunkservers are identified by "IP:port" combination.
- **FindLeaseHolder:** For a write/append operation, returns the chunkservers and identifies the primary chunkserver for a given filename and chunk index. Unlike FindLocations, this RPC creates the chunk if it does not exist. In the full GFS, this operation will also duplicate the chunk upon copy-on-write of a snapshot (not implemented here).
- **GetFileLength:** Returns the number of chunks in a given file.
- **MoveFile:** Requests master to move (rename) a file.
- **DeleteFile:** Requests master to delete a file.

2.5.2. Client → Chunkserver RPCs

- **ReadChunk:** Reads a portion of a given chunkhandle.
- **PushData:** Pushes data into memory before writing or appending.
- **WriteChunk:** Writes a portion of a given chunkhandle, using data previously pushed into memory.
- **Append:** Appends data to end of a chunk, using data previously pushed into memory.

2.5.3. Chunkserver → Chunkserver RPCs

- **SerializedWrite:** Used by the primary chunkserver of a chunk to replicate a write request onto secondary chunkservers.
- **CopyChunks:** During re-replication initiated by the master, used by a chunkserver to send one or more chunks to another chunkserver.

2.5.4. Chunkserver → Master RPCs

- **Heartbeat:** Renews the chunkserver's lease at the master and informs the master of all chunks on the chunkserver.

2.5.5. Master → Chunkserver RPCs

- **ReplicateChunks:** Used by the master to inform a chunkserver to re-replicate a chunk to another chunkserver.
- **DeleteChunks:** Used by the master to inform a chunkserver that it is storing a chunk no longer referenced by any file. It can thus be safely deleted.

2.6. Write/Append process

We use a simplified write process that lacks the cut-through routing used by the original GFS paper. A client writes to a chunk using the following procedure:

1. The client asks the master for the locations (IP:port) and primary replica for the chunk. If the chunk does not exist, the master randomly chooses 3 available chunkservers, one of which is chosen as the primary. This information is recorded in the master's SQLite database.
2. The client pushes the data directly to all replicas.

3. Once all the replicas have acknowledged receipt of the data, the client sends a write request to the primary. Currently, the primary can only process one write at a time which ensures that all writes are processed in serialized order.
4. The primary forwards the write request to all secondary replicas, and also writes the data to its own disk.
5. When the secondaries have written the data to disk, they reply to the primary.
6. The primary replies to the client. If any errors occurred, the data may have succeeded at an arbitrary subset of replicas and the region is left in an inconsistent state.

Append operations are similar, except that the client must ask the master for the number of chunks in order to append to the last chunk. If the data to be appended cannot fit in the last chunk, the primary pads the chunk with zeroes and returns an error RESOURCE_EXHAUSTED. The client is then retries the append at the next chunk index.

2.7. Heartbeats and Leases

Chunkservers maintain a lease at the master and must send periodic Heartbeat RPCs to the master to maintain the lease. Heartbeats also inform the master of the chunks that a chunkserver is storing. The master does not keep this information on disk, so when the master restarts, it waits for Heartbeat messages from chunkservers to know what chunks they are storing. When a chunkserver's lease expires, the master automatically re-replicates chunks stored on that server by asking a chunkserver with a copy of the chunk to send it to a newly assigned chunkserver.

The chunkserver currently sends a heartbeat every 2 seconds and leases expire after 10 seconds.

2.8. Benchmarks

We also implemented benchmarks in the client application and a benchmark (BM) server to assist in evaluating the system. During the benchmark, the client sends the current throughput to the BM server approximately once per second. The BM server then periodically prints out the throughput of all clients as well as the global aggregate throughput. This allowed us to perform benchmarks more easily and determine when the throughput had reached steady state.

3. Evaluation

3.1 Single client read/write performance

We benchmarked GFS with a single client and 6 chunkservers on separate machines. These machines had conventional hard disks and were located in the same rack with >10Gbps links between them. First, we created a 1.6GB file, consisting of 25 chunks. We then performed a series of random and sequential reads and writes to this file using request sizes of 4KB and 1MB. After waiting for the throughput to reach steady-state, we recorded the numbers here. To provide a baseline for comparison, we also ran the same benchmark on a regular 1.6GB file on the client's local hard disk, replacing calls to our GFS client library with C++ file I/O. The hard disk on the client was the same as in the chunkservers.

Table 1: Single client benchmarks with 6 chunkservers on separate machines under the same top-of-rack switch connected by >10Gbps network links

Mode	Op	Req size	GFS HDD (MB/s)	Local HDD (MB/s)
Seq	Read	4K	45	85
Rand	Read	4K	15	66
Seq	Write	4K	2	84
Rand	Write	4K	2	67
Seq	Read	1M	312	395
Rand	Read	1M	330	350
Seq	Write	1M	114	380
Rand	Write	1M	73	390

Overall, as shown in Table 1, throughput is much higher with 1MB requests compared with 4KB requests. This is due to the larger relative overhead of GFS in small requests compared to large once. Writes are slower than reads because reads are served from one replica while data must be written to all replicas. Since this is a single-client benchmark, the GFS throughput is always lower than the local hard disk. This illustrates the overall overhead of our system.

3.2. Single client master operation performance

We also benchmarked the performance of master operations since the single master is a bottleneck that restricts the overall system performance. For this benchmark, the client requested the master to create a large number of chunks and we recorded the rate at which they could be created. We also benchmarked how fast the master could lookup the location of chunks for a read operation. The master's SQLite database was located on a hard disk and SQLite's write-ahead logging (WAL) mode was enabled for higher performance.

Table 2: Master performance

Create chunk	5472 chunks/sec
Lookup chunk	16133 chunks/sec

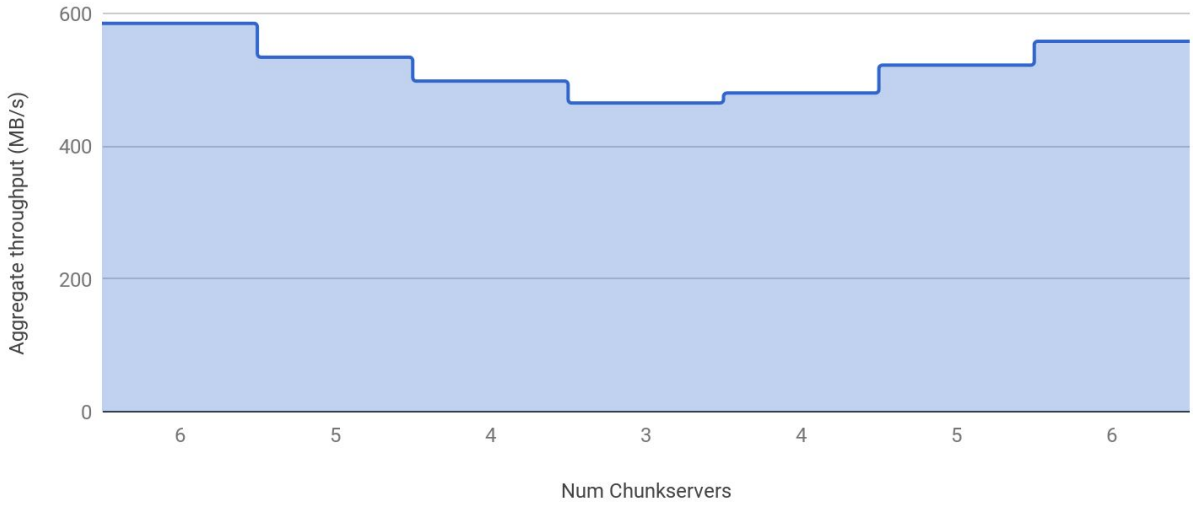
As shown in table 2, the master is able to support a relatively large number of creation and lookup operations. The numbers correspond to 5472×64MB = 350 GB/s of new chunks and 16133×64MB = 1033 GB/s of chunk accesses. The low rate of chunk creations compared to lookups is due to a SQLite transaction required for each chunk creation.

3.3. Multiple client performance

To illustrate the scalability of our system, we also benchmarked it with multiple clients. We started 3 clients and each client created a separate 1.6GB file on GFS (with 6 chunkservers on separate machines). Each client continuously performed sequential 1 MB reads from GFS. After reaching steady state, we then proceeded to kill 3 chunkservers in sequence and then bring them back up in sequence. The aggregate throughput was then recorded by the BM server and the steady-state throughputs recorded is shown in Figure 2.

As shown in the figure, multiple clients allow our system to achieve higher performance compared to the single client case. This is because when clients access chunks on different chunkservers, they are

Figure 2: Aggregate Throughput in MB/s (3 clients) vs. Num Chunkservers



able to take advantage of the combined bandwidth of the servers. The throughput with 3 clients, however, is not 3 times the throughput with a single client. This is because the clients' files were on the same 6 chunkservers and would often read data from the same chunkserver.

4. Future work

There were many improvements that considered making but were unable to do so due to time constraints, such as:

- Implementation of snapshots using copy-on-write.
- Checksums and error correcting codes. Deliberately corrupt data on a chunkserver and observe that the system detects and fixes it.
- Shadow read-only masters in case the primary master fails.
- GUI tool to display the chunks and metadata on each server and visualize operations like re-replication.

5. Conclusion

In this paper, we described GFS from Scratch, our partial re-implementation of the Google File System. We built a relatively simple C++ based system that is able to achieve reasonable performance. From this project, we learned a great deal from the project about how a large-scale distributed and fault-tolerant system is implemented and evaluated.

References

[1] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". Proc. of SOSP. Dec. 2003, pp. 29–43.

[2] Fikes, Andrew. "Storage architecture and challenges." Talk at the Google Faculty Summit (2010).