SGFS: Simplified Google File System

Name: Yanlei Zhao SUNetID: yanlei
Name: Wei Wei SUNetID: wwei2

1 Introduction

In this paper, we present SGFS, a simplified implementation of GFS with 3000+ lines of Go programming language[1]. Performance is not our top goal here so we simplify some of the techniques used in GFS and we will discuss these issues in details later.

Currently, our system correctly handles read and write. Read location is randomly selected amongst all the replicas to avoid overloaded chunk servers. Write and record append are implemented using lease management which is similar to GFS's design. We have a heartbeat mechanism to detect chunk server failure. Once a chunk server is down, we will scheudle a re-replication for all chunks on that chunk server to ensure each file is replicated several times. Our system periodically checkpoints and can restore system status to previous checkpoints. (That means any change happened between two checkpoints will be lost.) We implemented bootstrapping for master and server. Chunk servers will report chunks they have to master and master will reconstruct in memory data structure to process requests. We also evaluate our system using several tests and conduct detailed analysis for them.

2 Design Overview

In this section we will discuss general design decisions we make.

2.1 Architecture

The overall architecture of SGFS is almost the same as GFS. We have a single master to deal with metadata, answer query of clients, some housekeeping message and commands to send to chunk servers. Chunk servers are responsible for serving read and write to clients. Client library will provide API to client application to make better use of SGFS.

2.2 Client APIs

In SGFS, Client library supports the following operations:

[1]Source code: `github.com/wweiw/simplegfs`

Create It will create a new file using absolute path. If the absolute path already exists in the namespace, or its parent directory doesn't exist, the function will return false to the client application. As an enhancement, we will define a set of error code and return that to client.

Mkdir It will make a new directory using absolute path. Returns true if succeeded.

List It will list all files and directories under an absolute path. Currently, we only return a set of strings to the client application. Potentially, we will return a list of file information to the client application. Each entry of that list will tell client application about the file name, is directory or not, file size, etc.

Delete It will delete a file (or a directory) given an absolute path. If that path is a directory, delete that directory only if the directory has no children.

Read Read takes in a byte array, an offset in the file. It will return the byte array filled in with contents with offset in the file. It will return the number of bytes it reads.

Write Write takes in a byte array with the content client wants to write and an offset in the file. It will return the number of bytes it reads.

Append Append takes in a byte array of the content client wants to write. It will return the offset that the content resides in. Our record append guarantees the content is appended at-least-once. Client applications have to provide their own methods to resolve duplicates.

2.3 System Interactions

SGFS is a pretty complex system and in the following section we will discuss the interactions between different entities in SGFS.

Client - Master As is the same with the original GFS design, our system limit the interactions between client and master to the minimum. We use caching heavily on the client side. Locations and lease holder are cached with a expiration time set to 1 minute and lease expiration time respectively. If the client is reading or writing to the same file, then it will not ask the master for the same information again. During read, clients will also call `GetFileLength` RPC to get the file length. We also utilize caching for the file length. Because for small random reads, clients will often make `GetFileLengthRPC` call for the same file and it puts a lot of burden on the master side.

Client - Chunk Server Clients directly contact chunk servers to read and write data. Clients will push data to chunk servers and send write request to the primary. Primary will return the status of the write.

Chunk Server - Master When chunk servers are first started, it will report all the chunks they have to the master. The bootstrapping part can be very time-consuming. Chunk server will also periodically send heartbeat message to master. If chunk server is a lease holder, then it might piggyback lease extention requests on heartbeat messages whenever necessary.

We use RPC heavily during the development of SGFS to implement the complex interactions between different processes in the system. Table. (1) shows the primary RPC calls that empowers SGFS. Heartbeat messages are regularly exchanged between chunk servers and the master server. Heartbeat messages are utilized for chunk server failure detection, chunk location management, and chunk lease extentions.

Some RPC calls between clients and master is not listed becauses they are pretty straight forward. For example, we have `Create`, `Mkdir`, `List`, `Delete` used by clients to operate on namespace.

3 Master

In this section, we will discuss the design of Master and the decisions we make during the design.

3.1 Metadata

Master stores several kinds of metadata:

- Chunk handle ID. Currently highest allocated chunk handle is stored persistently. Whenever the master allocated a chunk handle ID for a new chunk in the system, we increment the ID by 1. The reason we store the ID on disk is that we want the ID to be unique across failures.
- Namespace. All file and directory information operation will be recorded in the log and periodical checkpointing is executed. All operations associated with namespace is managed by the `namespace manager`.
- (path, chunk index) to chunk handle mapping and chunk handle to (path, chunk index) mapping are persistent. Chunk handle to chunk server location mapping is stored in memory. When chunk server is started and connected to master, chunk server will report all the chunk it has to master, master will use chunkhandle to (path, chunk index) mapping to reconstruct the server location mapping in memory. All the operations are performed by the `chunk manager`.

3.2 Namespace Manager

We implemented a namespace manager that is responsible for all the namespace management including locking. We use a `map` to store all information of the namespace and we use a read write lock to control concurrent access to the namespace. The key for the map is path string and the value is all the information (e.g. a boolean to `isDir` indicate if it is a directory, if it is a file, we will record the length of that file.) that should be stored persistently.

For example, path `/var/tmp` will be a key in the map and its value is {isDir: true}. A file `/var/tmp/file` will be a key with its value as {isDir: false, fileLength: 100}.

When creating a file or a directory, we will first grab the write lock and then examine if its parent exists and its parent is a directory. If either condition fails, we will not create a new file.

For delete operation, our solution is similar to GFS's. We will rename it and will delete the file when we are doing garbage collection.

3.3 Chunk Manager

We implemented a chunk manager to manage all chunk information at the master.

It maintains a mapping from (path, chunk index) to chunk handle and a mapping from chunk handle to (path, chunk index). The first map is for `findLocations()` API. Clients will issue a RPC call to find the location associated with a certain path and chunk index. The second map is to reconstruct in-memory data structures when chunk servers report chunks they have when they start or through heartbeat messages.

Chunk manager periodically checkpoints its data to disk. One thing to note is that volatile in-

Name	Sender	Receiver	Notes
FindLocations	Client	Master	Returns the server locations for a given file and chunk index. It's useful for client's `Read()` and `Write()`.
FindLeaseHolder	Client	Master	Returns the server location of current lease holder. It's useful for client's `Write()`.
AddChunk	Client	Master	Add a new chunk for (path, chunk index). We will give the chunk a unique chunk handle and allocate 3 (This number of replica is configurable) servers to that handle. Returns chunk handle and servers' locations to the client.
ReportChunk	Chunk server	Master	Chunk server use ReportChunk to tell the master what chunks it has. Main use case: after chunk server is started, it will scan all the chunks it has on disk and send them to the master. Master will reconstruct in-memory data structure to serve clients' FindLocations calls.
GetFileLength	Client	Master	When client wants to read a file, it will first get the file length of that file. We use lazy allocation for a file. So a write at offset 10,000,000 will not result in creating a bunch of chunks but we will only create a single chunk for that offset. Getting the file length first ensures that we will not read out of file's boundary.
Read	Client	Chunk server	Chunk server will return the contents it reads from disk and the number of bytes it reads back to client. Clients' `Read()` use this RPC as a building block.
PushData	Client	Chunk server	The client pushes data into chunk servers' memory before issues a write request. Chunk servers returns an error code if this operation is not successful. Clients' `Write()` use this RPC as a building block.
Write	Client	Primary chunk server	Primary chunk server will return the number of bytes it successfully writes to disk. Clients' `Write()` use this RPC as a building block.
SerializedWrite	Primary chunk server	Secondary chunk server	The primary chunk server sends write request to secondary chunk servers with an array of serialized write requests, in the order of the primary chunk server applies them to its own local disk.
Append	Client	Primary chunk server	Primary chunk server will return the offset of the content we just appended in the file if successful.

Table 1: Important RPC Calls

memory data structure is not stored to disk which relies on chunk servers' `reportChunk` message to reconstruct.

3.4 Heartbeat Mechanism

Heartbeat is a very important mechanism in our system. We rely on heartbeats sent by chunk servers to detect which chunk server is dead and which chunk server is alive. If a server is found dead, all chunks that are stored on this server will update their locations and the master will add these chunks into pending re-replication queue. Periodically, the master will schedule a re-replication (based on priorities. Chunk that is replicated only once has higher priority than the chunk that is replicated twice.) for a chunk handle to ensure replication factor is reach. (One thing to note is that, if the failed server come back again, our current re-replication scheme can not handle it.)

4 Chunk Server Lease

We followed one of the key design principles of GFS, which is to minimize the master's involvement in all operations. Therefore we decided to use chunk server as the primary entity of lease management.

4.1 Lease Management

Leases are managed on chunk granularity. For each chunk, the master server maintains an in-memory mapping of chunkhandle to its current lease holder. Whenever needed, the master grants a lease to one of the chunk's replicas, and that chosen replica becomes the lease holder. A lease holder can renew its lease indefinitely, as long as there is write request being sent to that lease holder. A lease holder's lease extension requests are piggybacked onto its regular heartbeat messages.

4.2 Control Flow

Our data flow and control flow are separated, just like GFS. When a client intends to write to a chunk, it must first RPC the master server to find lease holder and locations of all the replicas that holds the chunk (The client also caches the primary chunk server information with expiration time set to the lease expiration time). The client then pushes data to all the replicas, which will be placed into each replicas in memory key-value store, with (clientId, timestamp) as a unique key. The client then issues a write request to the primary replica, with the (clientId, timestamp) key. Upon receiving the write request, the primary finds the data in its memory with the key provided by the write request and apply it to its local state. Once the local write succeeds on primary, the primary sends the write request to all of the secondaries and wait for them to finish applying data to their local state. Once every replica applies data to their local state successfully, the primary respond back to the client indicating write success.

4.3 Data Flow

Currently, to simplify our design, we choose to send data from client to each chunk server replicas concurrently, instead of pipelining the data flow from chunk servers to chunk servers. But with the basic

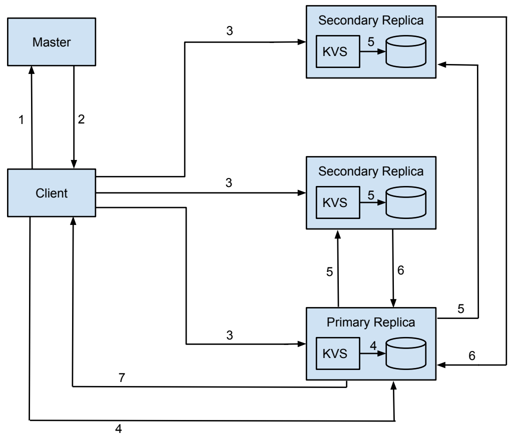

Figure 1: Write Process Diagram

inter-chunk server communication setup, it would be trivial to switch to pipelining client data.

4.4 Write Process Diagram

In Figure 1, we illustrate the life cycle of a write reuqest in SGFS.

1. Client sends `FindLocations` and `FindLeaseHolder` RPC to master.
2. Master sends the primary chunk server's address and all replicas location back to the Client. If there is no current lease holder, master grants a new leaes to one of the replicas.
3. Client use `PushData` to send data to all replicas' in memory key-value store.
4. Client sends `Write` request to the primary chunk server. The primary chunk server extracts the data from its key-value store and apply it to its local storage.
5. The primary chunk server sends `SerializedWrite` to secondary chunk servers. The secondary chunk servers extract data from their in memory key-value store and apply it to their local storage.
6. Once the secondary chunk servers successfully stores data on persistent storage, it replies back to the primary chunk server.
7. Upon receving all secondary chunk servers' replies, the primary chunk server replies back to the client.

4.5 Record Append

Record append is similar to the write process with extra logic at the primary.

- Client issues `Append` RPC to the primary.
- The primary verifies the size limit of the data, and pads the current chunk if there is not enough space for appending.
- The primary sends `SerializedWrite` to all secondaries to apply either padding or append.
- The primary will return the offset it writes the content back to the client who issues the write request.

5 Evaluation

Currently, the entire system is not optimized for speed but we will definitely look into efficiency issues in the future.

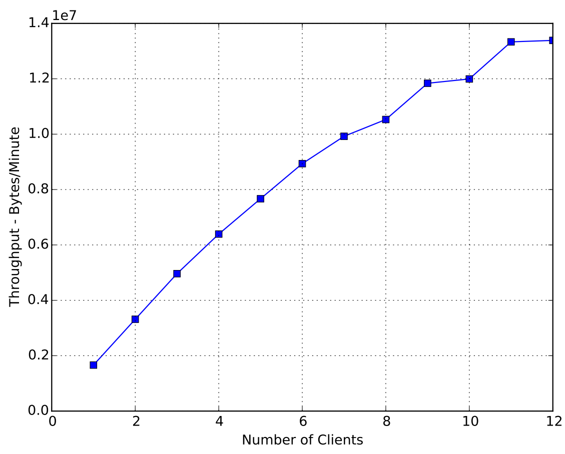

Figure 2: Random Read Performance

5.1 Read Performance

We setup 10 chunk server process running in our local machine and we configure our chunk size to be 1000 bytes. we have a file `/a` which contains 1100000 bytes. We have clients constantly reading 1000 bytes out of `/a` at a random offset. We vary the number of clients in our system and get the throughput SGFS achieves in Figure (2).

A client can read from one of three replicas of a chunk so read throughput should monotonically increase with the number of clients in the system. However as the number of readers increases, the chance that they are reading from the same chunk increases. That's why the throughput line is almost linear when number of clients is small. When the number of clients grows larger, it hurts performance and the trend is sublinear.

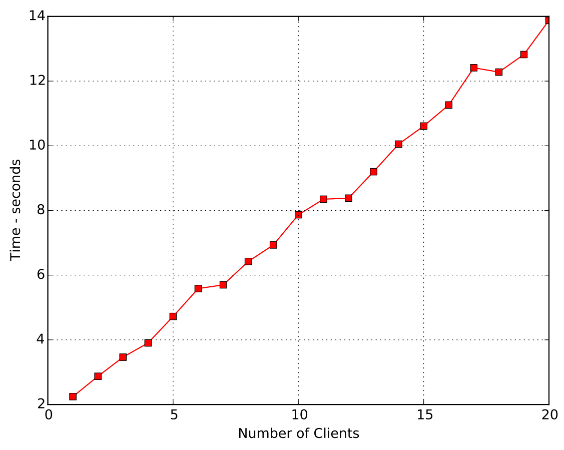

Figure 3: Concurrent Write Overhead (1)

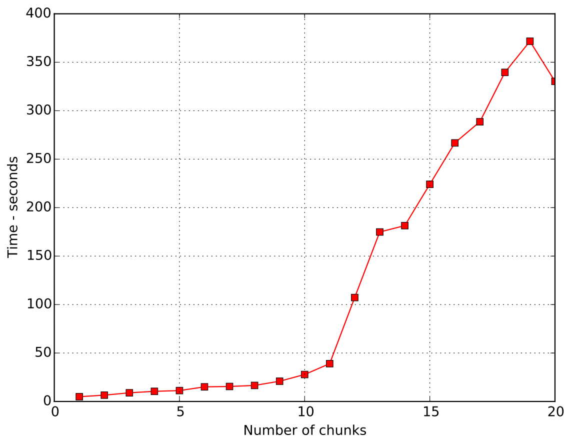

Figure 4: Concurrent Write Overhead (2)

5.2 Concurrent Write Overhead

In this section we will evaluate the overhead for concurrent write. All tests are done on the local machine with 500GB SSD and 16GB RAM.

Figure (3) is for N number of clients, ranging from 1 to 20, to concurrently write one chunk of data. We expect the graph to grow linearly because the more data you write the longer it takes. Concurrent overhead should not make the time it takes to write data grow exponentially.

Figure (4) is for 5 clients to concurrently write N number of chunks, ranging from 1 to 20. We also expected this to be linearly increasing but turns out that is not the case. The big jump happens from 12 chunks to 13 chunks. At that point, the memory limit on the local machine, which is 16GB is reached, the underlying operating system starts to swapping in and out memories on disk. We do see similar pattern on the original GFS paper where their aggregate write rate drops to 35MB/s when the number of clients increases to 16, but their limitation was introduced by network bandwidth, whereas ours is introduced by memory limit on local machine.

Currently each operations in a write process are fail stop but the client library are not handling failures and retries. We intend to add retry mechanisms as our future work.

5.3 Master Metadata

We have 6 chunk server process running on our local machine which has 16GB DRAM and 512GB SSD. We generate the namespace like this: each directory has five children directory (from a to e) and ten files under that directory (from 1 to 10). So the final namespace is like a tree and we limit the directory depth to be `dpeth` and `depth` should be smaller than 5. For depth equals 4 case, the total number of directories is around 800 and total number of files in the system is around 8000. (An example directory is /a/b/c/d An example file is /e/e/e/e/9). Each file contains 10 chunks so there are in total 240000 chunks in the system (with replication factor equals 3).

Depth	Chunk Manager	Namespace Manager	Bootstrap Time
1	12KB	4KB	1.18s
2	60KB	4KB	3.61s
3	340KB	20KB	12.87s
4	1.8MB	112KB	60.55s

Table 2: Master Metadata

As we can see from Table. (2), as the depth get bigger, the metadata that need to be persistently stored is increasing. The size is 5 times bigger for depth x compared with depth $x-1$. Bootstrap time monotonically increases as chunk servers have more chunks to report and master seems to be a bottleneck here. Here we add chunk server rate control to prevent chunk server overflowing master using a lot of report messages. And every chunk server also sleep for a while to reduce the probability of sending report messages together with another chunk server. Although there are a lot of chunks and namespace information, master's metadata size is still pretty small.

It takes our system a lot of time to bootstrap and partially it is because we don't send requests in a batch to reduce the total number of messages required.

6 Experience

Through the development of SGFS, we learned a lot about system design. Here are some of the lessons we learnt:

- Caching is really important for system performance. When we don't have client-side caching, everytime we want to read a file, the client will ask master for the location and file length. It often crashes the master with too many requests. After we add client caching, the whole system works like charm again.
- Message buffering and send as a batch is very important for a system too. Small writes to the same chunk should not be sent to chunk server immediately. Instead, we should buffer the write and send them as batches.
- We find test driven development extremely useful. We have a lot of unit tests and integration tests to ensure that our system is working.
- Client lease VS. Chunk server lease. Initially we implemented client lease instead of chunk server lease for implementation simplicity. Later we realized that increases significant amount of client-master communications, which violates one of the basic design principles of of SGFS, which is to limit the master's involvement in all operations.

7 Future Work

- Stale replica detection. We plan to store chunk version persistently in master and chunk servers to detect stale replica. Currently, we assume a fail-stop model. If a replica fails, it will not be back again.
- Garbage collection. We don't have we working garbage collection scheme yet. Chunk server will store a replica forever even though their metadata stored on the master is deleted. We should periodically scan the in-memory data structure to see what can be garbage collected. We haven't implemented it yet because it will lock the in-memory data structure during the scan which will introduce a lot of overhead.
- Performance. We can use fine-grained locking to avoid overhead. Also, `reportChunk` RPC does not support batch report yet which can put a lot of burden on the master server.