What can we learn from GFS about distributed Systems?

Brendan Good

2018-05-03 Thu 11:31

The Google File System was developed in the early 2000's for general data processing within Google. It is an interesting example of designing something specifically for your use-case and making trade-offs both within the system itself and how developers write application code around it. While Google ended up moving to Colossus, a different storage system, several years later, I think this paper is still worth studying for a few reasons:

- Most distributed systems aren't at the scale of Google's storage so there are still valuable things to learn
- The architecture and the main protocols follow from the assumptions in a natural way, making it a useful case-study for distributed systems design
- There doesn't seem to be a paper describing Colossus, so we can't study it in any meaningful depth from outside Google, so GFS it is.

This post has a completely different flow from the paper; while both start by addressing the constraints, this post focuses on how the core architecture and the protocols defined for writes and appends follow naturally from the constraints.

In particular, there are a few things in the paper not covered in this post, such as:

- Features not explicitly related to writing new data or reading data (e.g., snapshots)
- Things that are awkward to include in the post without ruining the flow (such as their locking system and some of the finer points about metadata).

- Some error handling cases. In a distributed system, these are undoubtedly crucial, but the sheer number of things that can go wrong are difficult to list meaningfully in this format.

Problem Parameters/Constraints

Before we go any further let's outline the constraints that shaped the design of GFS:

- Hardware failures are common: GFS was designed to anticipate many hardware failures; how do we ensure reliability in such a system?
- Files are relatively few but large: Typical files are on the order of a gigabyte in size.
- I/O is almost exclusively sequential: It's known that hard drives (even SSD's!) perform better with sequential I/O than random reads and writes. Given that performance is a big concern here, we can assume most writes are done sequentially and optimize for those with little regard for random write performance. However, random reads are still common enough that they should not be completely ignored.
- Multi-client writes should be supported: There should be a way for multiple clients to append to a single file without getting garbage data.
- High bandwidth is more important than low-latency: The latency of an individual read or write is unimportant; bulk data processing is more important.

There are also a few unwritten rules that are implied:

- Some choices, while perhaps theoretically less suited to the design goals, are deemed appropriate because they work well and are easier to implement.
- As a slight corollary, whenever an operation cannot be completed (either because of an error or some other reason[1]), it is up to the client to handle that case accordingly.

[1] An example of "some other reason" will be given when we talk about appends.

Intuition for hypothetical architectures

Let's start by thinking about how the constraints inform the potential architecture:

Representation of Files

We'll start by noting that the goal of GFS is to behave like a file system, which stores files as blocks. Blocks are used for looking up and operating on data (for GFS, a given operation can only occur on a single block, we'll explain more about how that works later on). So although conceptually we're working with files, the servers which hold the data think about the corresponding blocks, which this paper calls chunks.

Hardware Failures

The abundance of hardware failures (presumably at the level of whole machines and not just disks) pretty much implies a distributed system right off the bat. If the system were monolithic, a single component failure could mean the entire system goes down, yikes.

Of course, disks aren't exactly paragons of reliability either; they can also crash, or worse, read or write the wrong data and think everything's fine! Given that storage is at the very heart of this system, we should think about how to protect against this particular failure mode as well.

Efficiently detecting differences in data is exactly what a checksum is designed to do, so we'll want to store a checksum of the data. Ideally, the checksum should be stored independently of the data itself in order to avoid corruption of both user data and the checksum all in one go. Whenever data is written, the checksum can be updated or created and stored. Whenever data is read, it can then be checksummed and compared to the stored checksum. If a mismatch occurs, some action can be taken to restore that data. We'll go into more detail about how checksums interact with reads and writes once the rest of the architecture/fundamental operations are fleshed out.

Given that most of the writes to the file system are appends, we'll want a checksum that performs incremental checksum updates efficiently (instead of needing to checksum a bunch of data beyond what was just written). Fortunately, the Fletcher checksum is a checksum with this property [2]. So

[2] The paper doesn't explicitly mention that the Fletcher checksum is used in GFS, but it appears to be the best known checksum with the desired property.

what happens when a server discovers data corruption? It can simply return an error to the client (who will then request to read from a different chunk server) and then it informs the master of the data corruption and the master can then coordinate the correct data to be sent to the chunk server.

Bandwidth and Latency

High bandwidth requirements indicate that clients should not be communicating with any one machine; but, as alluded to above, we want this to be reasonable to implement. Because we can tolerate high latency operations and we care about reliability, data should be replicated to different machines.

I/O Patterns

The combination of big file sizes, mostly sequential I/O, and the typically large size of an I/O operation leads us to reconsider the block sizes. This is more of an implementation detail, and all of the benefits of adjusting it aren't clear at this stage, but it's worth pointing out that we can already see it emerge as a tunable parameter.

Starting to put it together

Now that we have identified how the individual parameters constrain the architecture, we can start putting together the architecture as a whole and see what new features emerge.

We'll start with the distributed aspect. As we know from Designing Data Intensive Applications, distributed systems can take on many forms and one which has a single "master" node that coordinates with other nodes is the easiest class to handle.

The first such system that may come to mind is "Have a master which keeps track of mappings from files to chunks and chunk locations (e.g., by assigning each chunk a unique ID). Whenever a client wants to read or write from/to a file, the client sends a request to the master which will then redirect the operation to the correct server holding the data". This seems like a pretty reasonable architecture, but there could be a problem with the master being a bottleneck. To elaborate, consider the process for a write; the master has to:

1. Accept each request
2. Look up the corresponding server and send a request to it

3. Wait for a response from each server
4. Finally respond to the client.

`https://b-t-g.github.io/assets/arc1.png`

Whew! That's a lot of work for one server for each individual operation.

Can we do better? The only thing that we really needed from the master in the above architecture is to find the corresponding chunk. So we change the system so that:

1. The client converts the desired byte offset to a chunk index[3] sends a request the master containing the file name and chunk offset
2. The master replies with the chunk identifier and the locations of the chunk servers
3. The client queries the server directly
4. The server performs any work that it needs to do (more details on that later) and replies (in the case on success or error).

`https://b-t-g.github.io/assets/arc2.png`

We'll expand on this protocol, especially step 4, in a few paragraphs (this is not the final "version" of the write protocol). In this protocol, the client can cache the chunk server information and send multiple operations directly to the server without needing to talk to the master. It does add some complexity around handling crashed servers or deleted chunks that the client wishes to read. The first problem can be solved by the master keeping up-to-date knowledge about which servers hold which chunk (e.g., by sending heartbeat messages to the chunk servers). The second problem can be mitigated by having the servers know explicitly what chunks it has and respond with an error message or the like.

While this system seems great for our needs, we do need to be a bit careful about a few things:

1. A single master system is, by default, a bit less reliable because precautions need to be taken to replicate the data on the master server in case it crashes. We can mitigate that by keeping a log of important events and replicating that to several different servers and, when the master crashes, one of the standby servers can pull up that operation log from that server and carry on as the new master

[3]This is calculated by using the byte offset and the fixed chunk size to find out which chunk contains that specific byte offset. The master could, in principle do this as well, but we want to make the master as little a bottleneck as possible.

2. To properly account for chunk servers going down (and coming back up), we should have some way of associating a version with each chunk server[4]

3. This protocol, as written, doesn't address data replication

How should we replicate in the face of wanting sane semantics for multiple clients writing to a single chunk? We value reliability and not so much the latency of individual operations, so we can propagate each write to another replica before acknowledging that the write was a success. Is there a way to accomplish this without the client having to coordinate with each chunk server (which is highly reminiscent of a "multi-master" system)? The master could nominate a "primary chunk server" which can coordinate all of the writes. For example, suppose we had chunk servers X, Y, and Z where X is the primary chunk server and Y and Z are secondaries. Suppose the client wants to write to chunk C. Let us further suppose that Y is the closest[5] chunk server to the client, so:

1. The client asks the master where it can find the desired chunk. If no primary chunk server has been nominated, the master will do so now

2. The master responds with all the chunk servers holding that chunk (primary and secondary). The client can cache this information for future file mutations

3. The data is pushed to the chunk servers[6]

4. When all of the servers have received the data, the client sends a request to X

5. X forwards the write requests to Y and Z and then the writes start (in order to maintain a consistent order of writes)

6. Y and Z inform X when their writes are done

7. When all of the writes are done, X informs the client that the write is complete

[4]When a new primary chunk server is nominated, the chunk version of the new primary chunk server is bumped, the new primary chunk server then tells each up-to-date chunk server to do the same.

[5]Within Google's network, "closeness" is typically done by analyzing the IP address.

[6]The data flow for each write is done by sending data to the nearest server, the advantage for this is that it reduces network bottlenecks and the use of high-latency links.

`https://b-t-g.github.io/assets/arc3.png`

If there's an error at any replica, the client can retry the operation; this may result in duplicated data, either in full or in part, which may cause the replicas to not have identical data[7]. The client adapts to this situation by using the checksum to strip out padding and incomplete data. Detecting duplicates, however, is done on an application by application basis.

This also has the added benefit that error messages are from a single source (X in this case).

Even though this protocol is not entirely straight-forward, it follows naturally from the constraints of the problem; we need all of the servers to have the data before we can begin the writes because we need all of the servers to write data in the same order without synchronization. If we had writes and data propagation going on at the same time, there would need to be a lot more coordination to ensure writes occur in the same order.

Finally, there's chunk size; there are numerous benefits for this use case to having a large chunk size:

- If we had a small chunk size, then we would need to break up our operations into much smaller operations and the client would have to either query the master server for each one of those operations to find the chunk location or we would have to cache where each of those many chunks live! Additionally:
- With large chunk sizes, a client may only perform operations on a single chunk. With that in mind, network overhead can be reduced by keeping a persistent TCP connection to the chunk server.
- It reduces the metadata that the master needs to hold; perhaps by enough to allow the metadata to fit in memory.

Therefore, the chunk size should be as big as possible (and experimentally chosen)[8].

Multi-Client appends

Now that we've nailed down the core architecture, we can now talk about checksums and the protocol for multi-client atomic appends (record ap-

[7]It's worth explicitly pointing out that, because of this, that checksums should be unique to chunkservers.

[8]The chunk size chosen was 64 MB, which is huge by block size standards[9]! Internal fragmentation is reduced by only allocating more space when necessary.

[9]In ZFS, for example, the **maximum** block (record) size is 128 KB! In Ext4, the maximum is 64 KB!

pends).

With the procedure we described for writes, we get record appends[10] pretty much for free. In the case where we wish to append to a chunk that fits within the chunk size, then the procedure is exactly the write procedure described earlier. There's just one thing that we need to be careful about; remember that each operation can only work on one block so, if the append will cause the chunk to exceed the chunk size, the primary will pad out the rest of the chunk (and inform the secondary replicas to do so) and reply to the client that the write should be retried on the next chunk. The reason we pad out the chunk instead of extending it over two blocks is because each operation can only operate on a single block, so extending it over two blocks would cause two different appends, which may not be atomic.[11]

Lessons Learned

Going back to the title, what have we learned from GFS about distributed system design? Knowing your constraints and designing your architecture around them is ridiculously powerful. It can give you a simple, yet very efficient architecture and, more importantly, can sometimes practically design our protocols for us.

Of course, life is not always so kind as to grant us knowledge of what our needs are and how the system will evolve but, even in cases such as those, it is better to not prematurely commit to architectural decisions. If, for no other reason, that we cannot use the style of architecting described in this paper to make the architecture as suitable and simple as possible for the needs of the system.

[10]Record appends are differentiated from "regular" appends because a regular append is just a write at an offset that the caller/client believes to be the end of the file; in a record append, GFS chooses the offset that it knows to be the end of the file. Additionally, record appends are atomic; these properties are useful when multiple clients are trying to write to the same file.

[11]For normal writes that are either big or cross a chunk boundary, they are split into multiple writes. Appends are restricted to writing no more than 1/4 of the block size to avoid unnecessary allocations and internal fragmentation.