GFS FAQ

Q: Why is atomic record append at-least-once, rather than exactly once?

Section 3.1, Step 7, says that if a write fails at one of the secondaries, the client re-tries the write. That will cause the data to be appended more than once at the non-failed replicas. A different design could probably detect duplicate client requests despite arbitrary failures (e.g. a primary failure between the original request and the client's retry). You'll implement such a design in Lab 3, at considerable expense in complexity and performance.

Q: How does an application know what sections of a chunk consist of padding and duplicate records?

A: To detect padding, applications can put a predictable magic number at the start of a valid record, or include a checksum that will likely only be valid if the record is valid. The application can detect duplicates by including unique IDs in records. Then, if it reads a record that has the same ID as an earlier record, it knows that they are duplicates of each other. GFS provides a library for applications that handles these cases.

Q: How can clients find their data given that atomic record append writes it at an unpredictable offset in the file?

A: Append (and GFS in general) is mostly intended for applications that sequentially read entire files. Such applications will scan the file looking for valid records (see the previous question), so they don't need to know the record locations in advance. For example, the file might contain the set of link URLs encountered by a set of concurrent web crawlers. The file offset of any given URL doesn't matter much; readers just want to be able to read the entire set of URLs.

Q: What's a checksum?

A: A checksum algorithm takes a block of bytes as input and returns a single number that's a function of all the input bytes. For example, a simple checksum might be the sum of all the bytes in the input (mod some big number). GFS stores the checksum of each chunk as well as the chunk. When a chunkserver writes a chunk on its disk, it first computes the checksum of the new chunk, and saves the checksum on disk as well as the chunk. When a chunkserver reads a chunk from disk, it also reads the previously-saved checksum, re-computes a checksum from the chunk read from disk, and checks that the two checksums match. If the data was corrupted by the disk, the checksums won't match, and the chunkserver will know to return an error. Separately, some GFS applications stored their own checksums, over application-defined records, inside GFS files, to distinguish between correct records and padding. CRC32 is an example of a checksum algorithm.

Q: The paper mentions reference counts -- what are they?

A: They are part of the implementation of copy-on-write for snapshots. When GFS creates a snapshot, it doesn't copy the chunks, but instead increases the reference counter of each chunk. This makes creating a snapshot inexpensive. If a client writes a chunk and the master notices the reference count is greater than one, the master first makes a copy so that the client can update the copy (instead of the chunk that is part of the snapshot). You can view this as delaying the copy until it is absolutely necessary. The hope is that not all chunks will be modified and one can avoid making some copies.

Q: If an application uses the standard POSIX file APIs, would it need to be modified in order to use GFS?

A: Yes, but GFS isn't intended for existing applications. It is designed for newly-written applications, such as MapReduce programs.

Q: How does GFS determine the location of the nearest replica?

A: The paper hints that GFS does this based on the IP addresses of the servers storing the available replicas. In 2003, Google must have assigned IP addresses in such a way that if two IP addresses are close to each other in IP address space, then they are also close together in the machine room.

Q: Suppose S1 is the primary for a chunk, and the network between the master and S1 fails. The master will notice and designate some other server as primary, say S2. Since S1 didn't actually fail, are there now two primaries for the same chunk?

A: That would be a disaster, since both primaries might apply different updates to the same chunk. Luckily GFS's lease mechanism prevents this scenario. The master granted S1 a 60-second lease to be primary. S1 knows to stop being primary when its lease expires. The master won't grant a lease to S2 until the previous lease to S1 expires. So S2 won't start acting as primary until after S1 stops.

Q: 64 megabytes sounds awkwardly large for the chunk size!

A: The 64 MB chunk size is the unit of book-keeping in the master, and the granularity at which files are sharded over chunkservers. Clients could issue smaller reads and writes -- they were not forced to deal in whole 64 MB chunks. The point of using such a big chunk size is to reduce the size of the meta-data tables in the master, and to avoid limiting clients that want to do huge transfers to reduce overhead. On the other hand, files less than 64 MB in size do not get much parallelism.

Q: Does Google still use GFS?

A: Rumor has it that GFS has been replaced by something called Colossus, with the same overall goals, but improvements in master performance and fault-tolerance.

Q: How acceptable is it that GFS trades correctness for performance and simplicity?

A: This a recurring theme in distributed systems. Strong consistency usually requires protocols that are complex and require chit-chat between machines (as we will see in the next few lectures). By exploiting ways that specific application classes can tolerate relaxed consistency, one can design systems that have good performance and sufficient consistency. For example, GFS optimizes for MapReduce applications, which need high read performance for large files and are OK with having holes in files, records showing up several times, and inconsistent reads. On the other hand, GFS would not be good for storing account balances at a bank.

Q: What if the master fails?

A: There are replica masters with a full copy of the master state; the paper's design requires human intervention to switch to one of the replicas after a master failure (Section 5.1.3). We will see later how to build replicated services with automatic cut-over to a backup, using Raft.

Q: Why 3 replicas?

A: This number is chosen to minimize the probability that a chunk will be lost. I imagine is calculated based on the reliability of disks and the time it takes to create a new replica. Here is a study on disk reliability from that era:
https://research.google.com/archive/disk_failures.pdf. If clients want a file to be replicated more than 3 times (e.g., to allow for more concurrent reads), then they can specify that.

Q: Did having a single master turn out to be a good idea?

A: That idea simplified initial deployment but was not so great in the long run. This article -- https://queue.acm.org/detail.cfm?id=1594206 -- says that as the years went by and GFS use grew, a few things went wrong. The number of files grew enough that it wasn't reasonable to store all files' metadata in the RAM of a single master. The number of clients grew enough that a single master didn't have enough CPU power to serve them. The fact that switching from a failed master to one of its backups required human intervention made recovery slow. Apparently Google's replacement for GFS, Colossus, splits the master over multiple servers, and has more automated master failure recovery.

Q: What is internal fragmentation? Why does lazy allocation help?

A: Internal fragmentation is the space wasted when a system uses an allocation unit larger than needed for the requested allocation. For example, in GFS the risk is an application creates a 1-byte file and 64M-1byte is wasted, because the allocation size is 64MB (a chunk). GFS avoids this potential problem, because the 64MB is lazy allocated. Every chunk is a Linux file, and so when an application creates 1-byte file, the on-disk representation of the chunk is 1-byte Linux file.