The Distributed File System

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003

- We only cover GFS aspects that are most relevant for MapReduce

1.1 Motivation

- The abstraction of a single global file system greatly simplifies programming for warehouse-scale computers (i.e., clusters of commodity machines). A process can access a file that is possibly distributed over many machines as if it was a regular file stored locally.
- This frees the programmer from having to worry about messy details such as:
  - Into how many chunks should a large file be split and on which machines in the cluster should these chunks be stored?
  - Keeping track of the chunks and which chunk(s) to modify when the user tries to change a sequence of bytes somewhere in the file.
  - Management of chunk replicas: Replicas are needed to provision against failures. (Recall that in a large cluster of commodity machines failures are common.) These replicas have to be kept in sync with each other. What should be done when a machine fails? Which other machine should receive a new copy of a file chunk that was stored on the failed machine? When the failed machine comes back online, should the additional copy be removed?
  - Coordinating concurrent file access: What if two processes are trying to update the same file chunk? What if the processes access different chunks of the same file? How do we make sure that file creation is atomic, i.e., no two processes can create the same file concurrently?

1.2 Design Goals

- The distributed file system should be able to handle a modest number of large files. Google aimed for a few million files, most of them 100 MB or larger in size. In particular, the Google File System (GFS) was designed to handle multi-gigabyte and larger files efficiently.
- High sustained bandwidth is more important than low latency. This is based on the assumption that the file system would have to deal mostly with large bulk accesses.
  - Small reads and writes of a few bytes at random locations in a file would be costly in a distributed file system due to the high access latency.
- GFS should support a typical file system interface:
  - Files are organized in directories.
  - The file system supports operations such as create, delete, open, close, read, and write with syntax similar to a local file system.

2.1 GFS Architecture

- GFS consists of a single master, multiple chunkservers, and many clients. All machines are commodity Linux machines.
- Files are divided into fixed-size chunks, typically 64 MB or 128 MB in size. (LINK1: More info about chunk size) These chunks are stored on the chunkservers' local disks as Linux files. Chunks are typically replicated on multiple chunkservers, e.g., three times. (LINK2: More about replica placement)
- The master maintains all file system metadata such as namespace, access control information, the mapping from files to chunks, and chunk locations.

Why a Single Master?

- Having a single master simplifies the design, because the master can make decisions with global knowledge.
- There are two potential drawbacks with having a single master:
  - It could become a bottleneck. To avoid this problem, the GFS master does not perform any file transfers or read and write operations.
  - It is a single point of failure. Fortunately, the probability of a single machine failing is comparably low. Furthermore, GFS is designed such that the master can recover quickly, minimizing downtime.

Chunk Size

- Choosing a larger chunk size introduces system tradeoffs. In general, the number of chunks is limited by the master's memory size. (For fast accesses, the master manages all metadata in memory.)
  - In GFS, there are only 64 bytes of metadata per 64 MB chunk. Note that most chunks are "full", because all but the last chunk are completely filled and in large files the number of chunks is large.
  - GFS uses less than 64 bytes of namespace data per file.
- Advantages of large chunk size:
  - There will be fewer interactions with the master. Recall that GFS is designed to support large sequential reads and writes. Reading a block of 128 MB would involve 2-3 chunks of size 64 MB, but 128-129 chunks of size 1 MB.
  - For the same file size, less chunk location information is needed, reducing the amount of information managed by the master. Similarly, for large chunk sizes, metadata even for terabyte-sized working sets could be cached in the clients' memory.
- Disadvantage of large chunk size:
  - Fewer chunks result in fewer options for load balancing. For example, consider clients reading at offsets 1 million, 3 million, and 5 million. Using a 1 MB chunk size, each client would access a different chunk. Using a 64 MB chunk size, all three would access the same one. This can result in hotspots (for reading and writing) or concurrency-related delays (for writing).
    - Reading hotspots can be addressed by using a higher replication factor, distributing the load over more replicas.
    - Reading hotspots can also be addressed by letting clients read from other clients who already got the chunk.

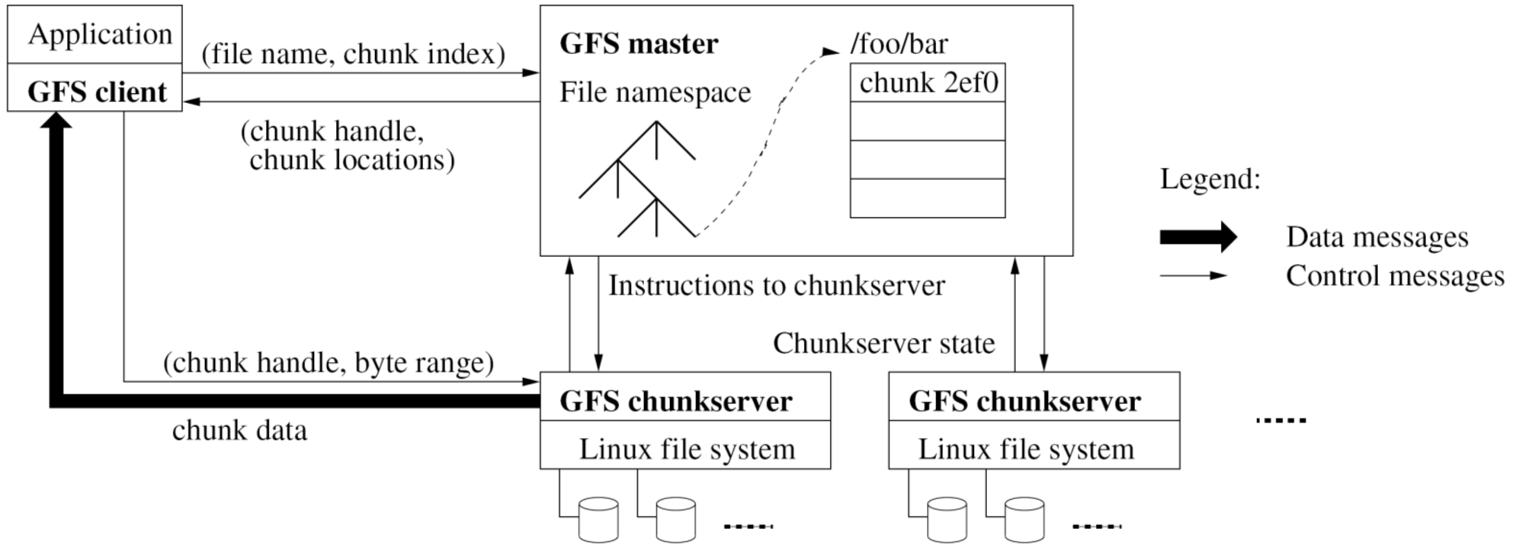
Replica Placement

- The goals of choosing machines to host the replicas of a chunk are scalability, reliability, and availability. Achieving this is difficult, because in a typical GFS setting there are hundreds of chunkservers spread across many machine racks, accessed from hundreds of clients in the same or different racks. Communication may cross multiple network switches. And bandwidth into or out of a rack may be less than the aggregate bandwidth of all the machines within the same rack.
- Spreading replicas across different racks is good for fault tolerance. Furthermore, read operations benefit from the aggregate bandwidth of multiple racks. On the other hand, it is bad for writes as the updates now flow through multiple racks.
- The master can move replicas or create/delete them to react to system changes and failures.

2.2 High-Level Functionality

- The master controls all system-wide activities such as chunk lease management, garbage collection, and chunk migration.
- The master communicates with the chunkservers through HeartBeat messages to give instructions and collect their state.
- The client who tries to access a file gets metadata (in particular the locations of relevant file chunks) from the master, but then accesses files directly through the chunkservers. (LINK1, LINK2)
- There is no GFS-level file caching for several reasons:
  - Caching offers little benefit for streaming access (i.e., reading a file "in one direction") or for large working sets.
  - This avoids having to deal with cache coherence issues in the distributed setting.
  - On each chunkserver, standard (local) Linux file caching is sufficient for good performance.

LINK1: Reading from a Chunk

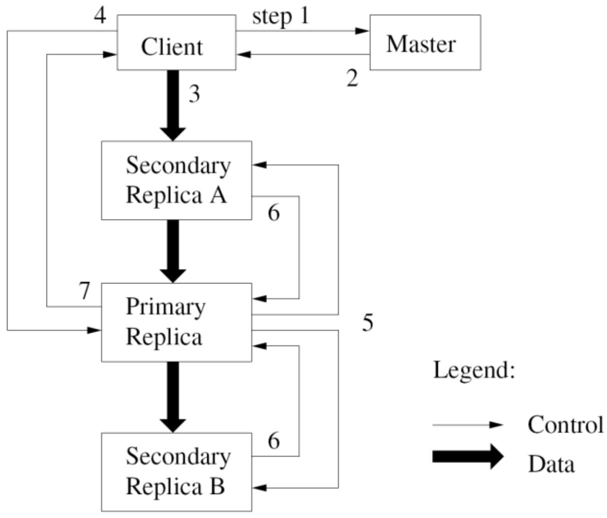


- Assume the client wants to read data at some offset in some GFS file. From the offset, it can easily compute the index of the chunk it needs to access (LINK: HOW?). It then asks the master for the locations of this chunk.
  - Since chunk locations rarely change, the client typically buffers this location info for some time. Then it can avoid contacting the master for future accesses to the same chunk.
- Knowing the locations of the replicas of the requested chunk, the client requests the data from a nearby chunkserver. This data transfer does not involve the master at all.
- Many clients can read from different chunkservers in parallel, enabling GFS to support a high aggregate data transfer rate.

LINK2: Updating a Chunk

- Writing to a file is more challenging than reading, because all chunk replicas have to be in sync. This means that they have to apply the same updates in the same order. GFS achieves this by designating one replica of a chunk as the primary. It determines update order and notifies the secondaries to follow its lead.
  - Large updates are broken into chunk-wise updates.

1. The client contacts the master to find the locations of the primary and secondary replicas of the chunk it is trying to update.
2. The master responds with the requested information.
3. The client starts pushing the updates to all replicas. (LINK1: More details about this process)
4. After receiving all acks that the update arrived, the client sends a write request to the primary who assigns it a serial number.
5. The primary forwards the write request, including its serial number, to all other replicas.
6. After completing the write request, the secondaries acknowledge success to the primary.
7. The primary replies to the client about the outcome of the update operation. If the operation failed, the client would re-try steps 3-7. (LINK2: Is data consistency guaranteed?)



Data Flow

- In GFS, data flow is decoupled from control flow for efficient network use. Instead of the client multi-casting the chunk update to each replica, the data is pipelined linearly along a chain of chunkservers. There are several reasons for this design decision:
  - Pipelining makes use of the full outbound bandwidth for the fastest possible transfer, instead of dividing it in a non-linear topology.
  - It avoids network bottlenecks by forwarding to the "next closest" destination machine.
  - It minimizes latency: once a chunkserver receives data, it starts forwarding to the next one in the pipeline immediately. Notice that the machines are typically connected by a switched network with full-duplex links. This intuitively means that sending out data does not affect the bandwidth for receiving data (and vice versa). According to Google's measurements in 2003, pipelining enabled them to distribute 1 MB of data in about 80 milliseconds.

Problems With Updates

- It is challenging to ensure consistent update order across replicas when machines can fail or be slow to respond.
- GFS has mechanisms to deal with these problems, but for general updates (in contrast to append-only updates) it relies on a relaxed consistency model. This means that under certain circumstances, in particular when a client uses cached chunk location information, it might read from a stale chunk replica.
- Life would be easier without updates. There would be no need to keep replicas in sync and no risk of reading stale replicas.
- The good news is that MapReduce does not need to perform any updates in the distributed file system. As we will discuss later, Map only reads data from GFS. Reduce writes data to GFS, but does so by creating a new file and never modifying it. (The reduce task creates a temp file in the local file system. It then atomically copies and renames it to a file in the global file system.)

2.2.3 Achieving High Availability

- Master and chunkservers can restore their state and start up in seconds.
- Chunk replication ensures data availability in the presence of failures.
- Master data, in particular operation log and checkpoints, can be replicated to enable quick failover. When the master fails permanently, monitoring infrastructure outside GFS will start a new master with the replicated operation log. Since clients use a DNS alias, they are automatically re-routed to the new master.

Summary

- GFS supports large-scale data processing workloads on commodity hardware.
- Component failures are treated as the norm, not the exception. GFS deals with failures through multiple mechanisms:
  - Constant monitoring of all participants.
  - Replication of crucial data.
  - A relaxed consistency model for updates, which does not negatively affect data consistency for MapReduce.
  - Fast, automatic recovery.
- GFS is optimized for huge files, append writes, and large sequential reads.
- It achieves high aggregate throughput for concurrent readers and writers through separation of file system control (through master) from data transfer (directly between chunkservers and clients).