Google File System

The Google File System is one online tool developed by Google.

Google is a multi-billion-dollar company. It's one of the big power players on the World Wide Web and beyond. The company relies on a **distributed computing system** to provide users with the infrastructure they need to access, create and alter data. Surely Google buys state-of-the-art computers and servers to keep things running smoothly, right?

Wrong. The machines that power Google's operations aren't cutting-edge power computers with lots of bells and whistles. In fact, they're relatively inexpensive machines running on Linux operating systems. How can one of the most influential companies on the Web rely on cheap hardware? It's due to the **Google File System** (**GFS**), which capitalizes on the strengths of off-the-shelf servers while compensating for any hardware weaknesses. It's all in the design.

Google uses the GFS to organize and manipulate huge files and to allow application developers the research and development resources they require. The GFS is unique to Google and isn't for sale. But it could serve as a model for file systems for organizations with similar needs.

Some GFS details remain a mystery to anyone outside of Google. For example, Google doesn't reveal how many computers it uses to operate the GFS. In official Google papers, the company only says that there are "thousands" of computers in the system (source: Google). But despite this veil of secrecy, Google has made much of the GFS's structure and operation public knowledge.

Append vs. Rewrite

The GFS team optimized the system for appended files rather than rewrites. That's because clients within Google rarely need to overwrite files -- they add data onto the end of files instead. While it's still possible to overwrite data on a file in the GFS, the system doesn't handle those processes very efficiently

Contents:

1. Google File System Basics

Google developers routinely deal with large files that can be difficult to manipulate using a traditional computer file system. The size of the files drove many of the decisions programmers had to make for the GFS's design. Another big concern was **scalability**, which refers to the ease of adding capacity to the system. A system is scalable if it's easy to increase the system's capacity. The system's performance shouldn't suffer as it grows. Google requires a very large network of computers to handle all of its files, so scalability is a top concern.

Because the network is so huge, monitoring and maintaining it is a challenging task. While developing the GFS, programmers decided to automate as much of the administrative duties required to keep the system running as possible. This is a key principle of **autonomic computing**, a concept in which computers are able to diagnose problems and solve them in real time without the need for human intervention. The challenge for the GFS team was to not only create an automatic monitoring system, but also to design it so that it could work across a huge network of computers.

The key to the team's designs was the concept of simplification. They came to the conclusion that as systems grow more complex, problems arise more often. A simple approach is easier to control, even when the scale of the system is huge.

Based on that philosophy, the GFS team decided that users would have access to basic file commands. These include commands like **open**, **create**, **read**, **write** and **close** files. The team also included a couple of specialized commands: **append** and **snapshot**. They created the specialized commands based on Google's needs. Append allows clients to add information to an existing file without overwriting previously written data. Snapshot is a command that creates quick copy of a computer's contents.

Files on the GFS tend to be very large, usually in the multi-gigabyte (GB) range. Accessing and manipulating files that large would take up a lot of the network's **bandwidth**. Bandwidth is the capacity of a system to move data from one location to another. The GFS addresses this problem by breaking files up into chunks of 64 megabytes (MB) each. Every chunk receives a unique 64-bit identification number called a **chunk handle**. While the GFS can process smaller files, its developers didn't optimize the system for those kinds of tasks.

By requiring all the file chunks to be the same size, the GFS simplifies resource application. It's easy to see which computers in the system are near capacity and which are underused. It's also easy to port chunks from one resource to another to balance the workload across the system.

Sharing the Load

Distributed computing *is all about networking several computers together and taking advantage of their individual resources in a collective way. Each computer contributes some of its resources (such as memory, processing power and hard drive space) to the overall network. It turns the entire network into a massive computer, with each individual computer acting as a processor and data storage device.*

2. Google File System Architecture

Google organized the GFS into **clusters** of computers. A cluster is simply a network of computers. Each cluster might contain hundreds or even thousands of machines. Within GFS clusters there are three kinds of entities: **clients**, **master servers** and **chunkservers**.

In the world of GFS, the term "client" refers to any entity that makes a file request. Requests can range from retrieving and manipulating existing files to creating new files on the system. Clients can be other computers or computer applications. You can think of clients as the customers of the GFS.

The master server acts as the coordinator for the cluster. The master's duties include maintaining an **operation log**, which keeps track of the activities of the master's cluster. The operation log helps keep service interruptions to a minimum -- if the master server crashes, a replacement server that has monitored the operation log can take its place. The master server also keeps track of **metadata**, which is the information that describes chunks. The metadata tells the master server to which files the chunks belong and where they fit within the overall file. Upon startup, the master **polls** all the chunkservers in its cluster. The chunkservers respond by telling the master server the contents of their inventories. From that moment on, the master server keeps track of the location of chunks within the cluster.

There's only one active master server per cluster at any one time (though each cluster has multiple copies of the master server in case of a hardware failure). That might sound like a good recipe for a bottleneck -- after all, if there's only one machine coordinating a cluster of thousands of computers, wouldn't that cause data traffic jams? The GFS gets around this sticky situation by keeping the messages the master server sends and receives very small. The master server doesn't actually handle file data at all. It leaves that up to the chunkservers.

Chunkservers are the workhorses of the GFS. They're responsible for storing the 64-MB file chunks. The chunkservers don't send chunks to the master server. Instead, they send requested chunks directly to the client. The GFS copies every chunk multiple times and stores it on different chunkservers. Each copy is called a **replica**. By default, the GFS makes three replicas per chunk, but users can change the setting and make more or fewer replicas if desired.

Which Replica Does GFS use?

The GFS separates replicas into two categories: ***primary replicas*** *and* ***secondary replicas****. A primary replica is the chunk that a chunkserver sends to a client. Secondary replicas serve as backups on other chunkservers. The master server decides which chunks will act as primary or secondary. If the client makes changes to the data in the chunk, then the master server lets the chunkservers with secondary replicas know they have to copy the new chunk off the primary chunkserver to stay current.*

3. Using the Google File System

File requests follow a standard work flow. A read request is simple -- the client sends a request to the master server to find out where the client can find a particular file on the system. The server responds with the location for the primary replica of the respective chunk. The primary replica holds a **lease** from the master server for the chunk in question.

If no replica currently holds a lease, the master server designates a chunk as the primary. It does this by comparing the IP address of the client to the addresses of the chunkservers containing the replicas. The master server chooses the chunkserver closest to the client. That chunkserver's chunk becomes the primary. The client then contacts the appropriate chunkserver directly, which sends the replica to the client.

Write requests are a little more complicated. The client still sends a request to the master server, which replies with the location of the primary and secondary replicas. The client stores this information in a memory cache. That way, if the client needs to refer to the same replica later on, it can bypass the master server. If the primary replica becomes unavailable or the replica changes, the client will have to consult the master server again before contacting a chunkserver.

The client then sends the write data to all the replicas, starting with the closest replica and ending with the furthest one. It doesn't matter if the closest replica is a primary or secondary. Google compares this data delivery method to a **pipeline**.

Once the replicas receive the data, the primary replica begins to assign consecutive serial numbers to each change to the file. Changes are called **mutations**. The serial numbers instruct the replicas on how to order each mutation. The primary then applies the mutations in sequential order to its own data. Then it sends a write request to the secondary replicas, which follow the same application process. If everything works as it should, all the replicas across the cluster incorporate the new data. The secondary replicas report back to the primary once the application process is over.

At that time, the primary replica reports back to the client. If the process was successful, it ends here. If not, the primary replica tells the client what happened. For example, if one secondary replica failed to update with a particular mutation, the primary replica notifies the client and retries the mutation application several more times. If the secondary replica doesn't update correctly, the primary replica tells the secondary replica to start over from the beginning of the write process. If that doesn't work, the master server will identify the affected replica as **garbage**.

What About Big Files?

If a client creates a write request that affects multiple chunks of a particularly large file, the GFS breaks the overall write request up into an individual request for each chunk. The rest of the process is the same as a normal write request.

4. Other Google File System Functions

Apart from the basic services the GFS provides, there are a few special functions that help keep the system running smoothly. While designing the system, the GFS developers knew that certain issues were bound to pop up based upon the system's architecture. They chose to use cheap hardware, which made building a large system a cost-effective process. It also meant that the individual computers in the system wouldn't always be reliable. The cheap price tag went hand-in-hand with computers that have a tendency to fail.

The GFS developers-built functions into the system to compensate for the inherent unreliability of individual components. Those functions include master and chunk replication, a streamlined recovery process, rebalancing, stale replica detection, garbage removal and **checksumming**.

While there's only one active master server per GFS cluster, copies of the master server exist on other machines. Some copies, called **shadow masters**, provide limited services even when the primary master server is active. Those services are limited to read requests, since those requests don't alter data in any way. The shadow master servers always lag a little behind the primary master server, but it's usually only a matter of fractions of a second. The master server replicas maintain contact with the primary master server, monitoring the operation log and polling chunkservers to keep track of data. If the primary master server fails and cannot restart, a secondary master server can take its place.

The GFS replicates chunks to ensure that data is available even if hardware fails. It stores replicas on different machines across different **racks**. That way, if an entire rack were to fail, the data would still exist in an accessible format on another machine. The GFS uses the unique chunk identifier to verify that each replica is valid. If one of the replica's handles doesn't match the chunk handle, the master server creates a new replica and assigns it to a chunkserver.

The master server also monitors the cluster as a whole and periodically rebalances the workload by shifting chunks from one chunkserver to another. All chunkservers run at near capacity, but never at full capacity. The master server also monitors chunks and verifies that each replica is current. If a replica doesn't match the chunk's identification number, the master server designates it as a stale replica. The stale replica becomes garbage. After three days, the master server can delete a garbage chunk. This is a safety measure -- users can check on a garbage chunk before it is deleted permanently and prevent unwanted deletions.

To prevent data corruption, the GFS uses a system called checksumming. The system breaks each 64 MB chunk into blocks of 64 kilobytes (KB). Each block within a chunk has its own 32-bit checksum, which is sort of like a fingerprint. The master server monitors chunks by looking at the checksums. If the checksum of a replica doesn't match the checksum in the master server's memory, the master server deletes the replica and creates a new one to replace it.

Heartbeats and Handshakes

The GFS components give system updates through electronic messages called ***heartbeats*** *and* ***handshakes****. These short messages allow the master server to stay current with each chunkserver's status.*

5. Google File System Hardware

Google says little about the hardware it currently uses to run the GFS other than it's a collection of off-the-shelf, cheap Linux servers. But in an official GFS report, Google revealed the specifications of the equipment it used to run some benchmarking tests on GFS performance. While the test equipment might not be a true representation of the current GFS hardware, it gives you an idea of the sort of computers Google uses to handle the massive amounts of data it stores and manipulates.

The test equipment included one master server, two master replicas, 16 clients and 16 chunkservers. All of them used the same hardware with the same specifications, and they all ran on Linux operating systems. Each had dual 1.4 gigahertz Pentium III processors, 2 GB of memory and two 80 GB hard drives. In comparison, several vendors currently offer consumer PCs that are more than twice as powerful as the servers Google used in its tests. Google developers proved that the GFS could work efficiently using modest equipment.

The network connecting the machines together consisted of a 100 megabytes-per-second (Mbps) full-duplex Ethernet connection and two Hewlett Packard 2524 network switches. The GFS developers connected the 16 client machines to one switch and the other 19 machines to another switch. They linked the two switches together with a one gigabyte-per-second (Gbps) connection.

By lagging behind the leading edge of hardware technology, Google can purchase equipment

and components at bargain prices. The structure of the GFS is such that it's easy to add more machines at any time. If a cluster begins to approach full capacity, Google can add more cheap hardware to the system and rebalance the workload. If a master server's memory is overtaxed, Google can upgrade the master server with more memory. The system is truly scalable.

How did Google decide to use this system? Some credit Google's hiring policy. Google has a reputation for hiring computer science majors right out of graduate school and giving them the resources and space, they need to experiment with systems like the GFS. Others say it comes from a "do what you can with what you have" mentality that many computer system developers (including Google's founders) seem to possess. In the end, Google probably chose the GFS because it's geared to handle the kinds of processes that help the company pursue its stated goal of organizing the world's information.

Bandwidth vs. Latency

*While **bandwidth** refers to a system's capacity for moving data from one location to another, **latency** refers to the delay between a system command and the corresponding response. In general, most system administrators strive for high bandwidth and low latency. Google developers are more concerned with bandwidth because Google applications manipulate very large files.*