acmqueue

Case Study
GFS: Evolution on Fast-forward

A discussion between Kirk McKusick and Sean Quinlan about the origin and evolution of the Google File System.

During the early stages of development at Google, the initial thinking did not include plans for building a new file system. While work was still being done on one of the earliest versions of the company's crawl and indexing system, however, it became quite clear to the core engineers that they really had no other choice, and GFS (Google File System) was born.

First, given that Google's goal was to build a vast storage network out of inexpensive commodity hardware, it had to be assumed that component failures would be the norm—meaning that constant monitoring, error detection, fault tolerance, and automatic recovery would have to be an integral part of the file system. Also, even by Google's earliest estimates, the system's throughput requirements were going to be daunting by anybody's standards—featuring multi-gigabyte files and data sets containing terabytes of information and millions of objects. Clearly, this meant traditional assumptions about I/O operations and block sizes would have to be revisited. There was also the matter of scalability. This was a file system that would surely need to scale like no other. Of course, back in those earliest days, no one could have possibly imagined just how much scalability would be required. They would learn about that soon enough.

Still, nearly a decade later, most of Google's mind-boggling store of data and its ever-growing array of applications continue to rely upon GFS. Many adjustments have been made to the file system along the way, and—together with a fair number of accommodations implemented within the applications that use GFS—they have made the journey possible.

To explore the reasoning behind a few of the more crucial initial design decisions as well as some of the incremental adaptations that have been made since then, ACM asked Sean Quinlan to pull back the covers on the changing file-system requirements and the evolving thinking at Google. Since Quinlan served as the GFS tech leader for a couple of years and continues now as a principal engineer at Google, he's in a good position to offer that perspective. As a grounding point beyond the Googleplex, ACM asked Kirk McKusick to lead the discussion. He is best known for his work on BSD (Berkeley Software Distribution) Unix, including the original design of the Berkeley FFS (Fast File System).

The discussion starts, appropriately enough, at the beginning—with the unorthodox decision to base the initial GFS implementation on a single-master design. At first blush, the risk of a single centralized master becoming a bandwidth bottleneck—or, worse, a single point of failure—seems fairly obvious, but it turns out Google's engineers had their reasons for making this choice.

MCKUSICK One of the more interesting—and significant—aspects of the original GFS architecture was the decision to base it on a single master. Can you walk us through what led to that decision?

QUINLAN The decision to go with a single master was actually one of the very first decisions, mostly just to simplify the overall design problem. That is, building a distributed master right from the outset was deemed too difficult and would take too much time. Also, by going with the single-master

approach, the engineers were able to simplify a lot of problems. Having a central place to control replication and garbage collection and many other activities was definitely simpler than handling it all on a distributed basis. So the decision was made to centralize that in one machine.

MCKUSICK Was this mostly about being able to roll out something within a reasonably short time frame?

QUINLAN Yes. In fact, some of the engineers who were involved in that early effort later went on to build BigTable, a distributed storage system, but that effort took many years. The decision to build the original GFS around the single master really helped get something out into the hands of users much more rapidly than would have otherwise been possible.

Also, in sketching out the use cases they anticipated, it didn't seem the single-master design would cause much of a problem. The scale they were thinking about back then was framed in terms of hundreds of terabytes and a few million files. In fact, the system worked just fine to start with.

MCKUSICK But then what?

QUINLAN Problems started to occur once the size of the underlying storage increased. Going from a few hundred terabytes up to petabytes, and then up to tens of petabytes… that really required a proportionate increase in the amount of metadata the master had to maintain. Also, operations such as scanning the metadata to look for recoveries all scaled linearly with the volume of data. So the amount of work required of the master grew substantially. The amount of storage needed to retain all that information grew as well.

In addition, this proved to be a bottleneck for the clients, even though the clients issue few metadata operations themselves—for example, a client talks to the master whenever it does an open. When you have thousands of clients all talking to the master at the same time, given that the master is capable of doing only a few thousand operations a second, the average client isn't able to command all that many operations per second. Also bear in mind that there are applications such as MapReduce, where you might suddenly have a thousand tasks, each wanting to open a number of files. Obviously, it would take a long time to handle all those requests, and the master would be under a fair amount of duress.

MCKUSICK Now, under the current schema for GFS, you have one master per cell, right?

QUINLAN That's correct.

MCKUSICK And historically you've had one cell per data center, right?

QUINLAN That was initially the goal, but it didn't work out like that to a large extent—partly because of the limitations of the single-master design and partly because isolation proved to be difficult. As a consequence, people generally ended up with more than one cell per data center. We also ended up doing what we call a "multi-cell" approach, which basically made it possible to put multiple GFS masters on top of a pool of chunkservers. That way, the chunkservers could be configured to have, say, eight GFS masters assigned to them, and that would give you at least one pool of underlying storage—with multiple master heads on it, if you will. Then the application was responsible for partitioning data across those different cells.

MCKUSICK Presumably each application would then essentially have its own master that would be responsible for managing its own little file system. Was that basically the idea?

QUINLAN Well, yes and no. Applications would tend to use either one master or a small set of the masters. We also have something we called Name Spaces, which are just a very static way of partitioning a namespace that people can use to hide all of this from the actual application. The

Logs Processing System offers an example of this approach: once logs exhaust their ability to use just one cell, they move to multiple GFS cells; a namespace file describes how the log data is partitioned across those different cells and basically serves to hide the exact partitioning from the application. But this is all fairly static.

MCKUSICK What's the performance like, in light of all that?

QUINLAN We ended up putting a fair amount of effort into tuning master performance, and it's atypical of Google to put a lot of work into tuning any one particular binary. Generally, our approach is just to get things working reasonably well and then turn our focus to scalability—which usually works well in that you can generally get your performance back by scaling things. Because in this instance we had a single bottleneck that was starting to have an impact on operations, however, we felt that investing a bit of additional effort into making the master lighter weight would be really worthwhile. In the course of scaling from thousands of operations to tens of thousands and beyond, the single master had become somewhat less of a bottleneck. That was a case where paying more attention to the efficiency of that one binary definitely helped keep GFS going for quite a bit longer than would have otherwise been possible.

It could be argued that managing to get GFS ready for production in record time constituted a victory in its own right and that, by speeding Google to market, this ultimately contributed mightily to the company's success. A team of three was responsible for all of that—for the core of GFS—and for the system being readied for deployment in less than a year.

But then came the price that so often befalls any successful system—that is, once the scale and use cases have had time to expand far beyond what anyone could have possibly imagined. In Google's case, those pressures proved to be particularly intense.

Although organizations don't make a habit of exchanging file-system statistics, it's safe to assume that GFS is the largest file system in operation (in fact, that was probably true even before Google's acquisition of YouTube). Hence, even though the original architects of GFS felt they had provided adequately for at least a couple of orders of magnitude of growth, Google quickly zoomed right past that.

In addition, the number of applications GFS was called upon to support soon ballooned. In an interview with one of the original GFS architects, Howard Gobioff (conducted just prior to his surprising death in early 2008), he recalled, "The original consumer of all our earliest GFS versions was basically this tremendously large crawling and indexing system. The second wave came when our quality team and research groups started using GFS rather aggressively—and basically, they were all looking to use GFS to store large data sets. And then, before long, we had 50 users, all of whom required a little support from time to time so they'd all keep playing nicely with each other."

One thing that helped tremendously was that Google built not only the file system but also all of the applications running on top of it. While adjustments were continually made in GFS to make it more accommodating to all the new use cases, the applications themselves were also developed with the various strengths and weaknesses of GFS in mind. "Because we built everything, we were free to cheat whenever we wanted to," Gobioff neatly summarized. "We could push problems back and forth between the application space and the file-system space, and then work out accommodations between the two."

The matter of sheer scale, however, called for some more substantial adjustments. One coping strategy had to do with the use of multiple "cells" across the network, functioning essentially as related but distinct file systems. Besides helping to deal with the immediate problem of scale, this proved to be a more efficient arrangement for the operations of widely dispersed data centers.

Rapid growth also put pressure on another key parameter of the original GFS design: the choice to establish 64 MB as the standard chunk size. That, of course, was much larger than the typical file-system block size, but only because the files generated by Google's crawling and indexing system were unusually large. As the application mix changed over time, however, ways had to be found to let the system deal efficiently with large numbers of files requiring far less than 64 MB (think in terms of Gmail, for example). The problem was not so much with the number of files itself, but rather with the memory demands all of those files made on the centralized master, thus exposing one of the bottleneck risks inherent in the original GFS design.

MCKUSICK I gather from the original GFS paper [Ghemawat, S., Gobioff, H., Leung, S-T. 2003. The Google File System. SOSP (ACM Symposium on Operating Systems Principles)] that file counts have been a significant issue for you right along. Can you go into that a little bit?

QUINLAN The file-count issue came up fairly early because of the way people ended up designing their systems around GFS. Let me cite a specific example. Early in my time at Google, I was involved in the design of the Logs Processing system. We initially had a model where a front-end server would write a log, which we would then basically copy into GFS for processing and archival. That was fine to start with, but then the number of front-end servers increased, each rolling logs every day. At the same time, the number of log types was going up, and then you'd have front-end servers that would go through crash loops and generate lots more logs. So we ended up with a lot more files than we had anticipated based on our initial back-of-the-envelope estimates.

This became an area we really had to keep an eye on. Finally, we just had to concede there was no way we were going to survive a continuation of the sort of file-count growth we had been experiencing.

MCKUSICK Let me make sure I'm following this correctly: your issue with file-count growth is a result of your needing to have a piece of metadata on the master for each file, and that metadata has to fit in the master's memory.

QUINLAN That's correct.

MCKUSICK And there are only a finite number of files you can accommodate before the master runs out of memory?

QUINLAN Exactly. And there are two bits of metadata. One identifies the file, and the other points out the chunks that back that file. If you had a chunk that contained only 1 MB, it would take up only 1 MB of disk space, but it still would require those two bits of metadata on the master. If your average file size ends up dipping below 64 MB, the ratio of the number of objects on your master to what you have in storage starts to go down. That's where you run into problems.

Going back to that logs example, it quickly became apparent that the natural mapping we had thought of—and which seemed to make perfect sense back when we were doing our back-of-the-envelope estimates—turned out not to be acceptable at all. We needed to find a way to work around this by figuring out how we could combine some number of underlying objects into larger files. In the case of the logs, that wasn't exactly rocket science, but it did require a lot of effort.

MCKUSICK That sounds like the old days when IBM had only a minimum disk allocation, so it provided you with a utility that let you pack a bunch of files together and then create a table of contents for that.

QUINLAN Exactly. For us, each application essentially ended up doing that to varying degrees. That proved to be less burdensome for some applications than for others. In the case of our logs, we hadn't really been planning to delete individual log files. It was more likely that we would end up rewriting the logs to anonymize them or do something else along those lines. That way, you don't get the garbage-collection problems that can come up if you delete only some of the files within a bundle.

For some other applications, however, the file-count problem was more acute. Many times, the most natural design for some application just wouldn't fit into GFS—even though at first glance you would think the file count would be perfectly acceptable, it would turn out to be a problem. When we started using more shared cells, we put quotas on both file counts and storage space. The limit that people have ended up running into most has been, by far, the file-count quota. In comparison, the underlying storage quota rarely proves to be a problem.

MCKUSICK What longer-term strategy have you come up with for dealing with the file-count issue? Certainly, it doesn't seem that a distributed master is really going to help with that—not if the master still has to keep all the metadata in memory, that is.

QUINLAN The distributed master certainly allows you to grow file counts, in line with the number of machines you're willing to throw at it. That certainly helps.

One of the appeals of the distributed multimaster model is that if you scale everything up by two orders of magnitude, then getting down to a 1-MB average file size is going to be a lot different from having a 64-MB average file size. If you end up going below 1 MB, then you're also going to run into other issues that you really need to be careful about. For example, if you end up having to read 10,000 10-KB files, you're going to be doing a lot more seeking than if you're just reading 100 1-MB files.

My gut feeling is that if you design for an average 1-MB file size, then that should provide for a much larger class of things than does a design that assumes a 64-MB average file size. Ideally, you would like to imagine a system that goes all the way down to much smaller file sizes, but 1 MB seems a reasonable compromise in our environment.

MCKUSICK What have you been doing to design GFS to work with 1-MB files?

QUINLAN We haven't been doing anything with the existing GFS design. Our distributed master system that will provide for 1-MB files is essentially a whole new design. That way, we can aim for something on the order of 100 million files per master. You can also have hundreds of masters.

MCKUSICK So, essentially no single master would have all this data on it?

QUINLAN That's the idea.

With the recent emergence within Google of BigTable, a distributed storage system for managing structured data, one potential remedy for the file-count problem—albeit perhaps not the very best one—is now available.

The significance of BigTable goes far beyond file counts, however. Specifically, it was designed to scale into the petabyte range across hundreds or thousands of machines, as well as to make it easy to add more machines to the system and automatically start taking advantage of those resources

without reconfiguration. For a company predicated on the notion of employing the collective power, potential redundancy, and economies of scale inherent in a massive deployment of commodity hardware, these rate as significant advantages indeed.

Accordingly, BigTable is now used in conjunction with a growing number of Google applications. Although it represents a departure of sorts from the past, it also must be said that BigTable was built on GFS, runs on GFS, and was consciously designed to remain consistent with most GFS principles. Consider it, therefore, as one of the major adaptations made along the way to help keep GFS viable in the face of rapid and widespread change.

MCKUSICK You now have this thing called BigTable. Do you view that as an application in its own right?

QUINLAN From the GFS point of view, it's an application, but it's clearly more of an infrastructure piece.

MCKUSICK If I understand this correctly, BigTable is essentially a lightweight relational database.

QUINLAN It's not really a relational database. I mean, we're not doing SQL and it doesn't really support joins and such. But BigTable is a structured storage system that lets you have lots of key-value pairs and a schema.

MCKUSICK Who are the real clients of BigTable?

QUINLAN BigTable is increasingly being used within Google for crawling and indexing systems, and we use it a lot within many of our client-facing applications. The truth of the matter is that there are tons of BigTable clients. Basically, any app with lots of small data items tends to use BigTable. That's especially true wherever there's fairly structured data.

MCKUSICK I guess the question I'm really trying to pose here is: Did BigTable just get stuck into a lot of these applications as an attempt to deal with the small-file problem, basically by taking a whole bunch of small things and then aggregating them together?

QUINLAN That has certainly been one use case for BigTable, but it was actually intended for a much more general sort of problem. If you're using BigTable in that way—that is, as a way of fighting the file-count problem where you might have otherwise used a file system to handle that—then you would not end up employing all of BigTable's functionality by any means. BigTable isn't really ideal for that purpose in that it requires resources for its own operations that are nontrivial. Also, it has a garbage-collection policy that's not super-aggressive, so that might not be the most efficient way to use your space. I'd say that the people who have been using BigTable purely to deal with the file-count problem probably haven't been terribly happy, but there's no question that it is one way for people to handle that problem.

MCKUSICK What I've read about GFS seems to suggest that the idea was to have only two basic data structures: logs and SSTables (Sorted String Tables). Since I'm guessing the SSTables must be used to handle key-value pairs and that sort of thing, how is that different from BigTable?

QUINLAN The main difference is that SSTables are immutable, while BigTable provides mutable key value storage, and a whole lot more. BigTable itself is actually built on top of logs and SSTables. Initially, it stores incoming data into transaction log files. Then it gets *compacted*—as we call it—into a series of SSTables, which in turn get compacted together over time. In some respects, it's reminiscent of a log-structure file system. Anyway, as you've observed, logs and SSTables do seem to be the two data structures underlying the way we structure most of our data. We have log files for

mutable stuff as it's being recorded. Then, once you have enough of that, you sort it and put it into this structure that has an index.

Even though GFS does not provide a Posix interface, it still has a pretty generic file-system interface, so people are essentially free to write any sort of data they like. It's just that, over time, the majority of our users have ended up using these two data structures. We also have something called *protocol buffers*, which is our data description language. The majority of data ends up being protocol buffers in these two structures.

Both provide for compression and checksums. Even though there are some people internally who end up reinventing these things, most people are content just to use those two basic building blocks.

Because GFS was designed initially to enable a crawling and indexing system, throughput was everything. In fact, the original paper written about the system makes this quite explicit: "High sustained bandwidth is more important than low latency. Most of our target applications place a premium on processing data in bulk at a high rate, while few have stringent response-time requirements for an individual read and write."

But then Google either developed or embraced many user-facing Internet services for which this is most definitely not the case.

One GFS shortcoming that this immediately exposed had to do with the original single-master design. A single point of failure may not have been a disaster for batch-oriented applications, but it was certainly unacceptable for latency-sensitive applications, such as video serving. The later addition of automated failover capabilities helped, but even then service could be out for up to a minute.

The other major challenge for GFS, of course, has revolved around finding ways to build latency-sensitive applications on top of a file system designed around an entirely different set of priorities.

MCKUSICK It's well documented that the initial emphasis in designing GFS was on batch efficiency as opposed to low latency. Now that has come back to cause you trouble, particularly in terms of handling things such as videos. How are you handling that?

QUINLAN The GFS design model from the get-go was all about achieving throughput, not about the latency at which that might be achieved. To give you a concrete example, if you're writing a file, it will typically be written in triplicate—meaning you'll actually be writing to three chunkservers. Should one of those chunkservers die or hiccup for a long period of time, the GFS master will notice the problem and schedule what we call a *pullchunk*, which means it will basically replicate one of those chunks. That will get you back up to three copies, and then the system will pass control back to the client, which will continue writing.

When we do a pullchunk we limit it to something on the order of 5-10 MB a second. So, for 64 MB, you're talking about 10 seconds for this recovery to take place. There are lots of other things like this that might take 10 seconds to a minute, which works just fine for batch-type operations. If you're doing a large MapReduce operation, you're OK just so long as one of the items is not a real straggler, in which case you've got yourself a different sort of problem. Still, generally speaking, a hiccup on the order of a minute over the course of an hour-long batch job doesn't really show up. If you are working on Gmail, however, and you're trying to write a mutation that represents some user action, then getting stuck for a minute is really going to mess you up.

We've had similar issues with our master failover. Initially, GFS had no provision for automatic master failover. It was a manual process. Although it didn't happen a lot, whenever it did, the cell might be down for an hour. Even our initial master-failover implementation required on the order of minutes. Over the past year, however, we've taken that down to something on the order of tens of seconds.

MCKUSICK Still, for user-facing applications, that's not acceptable.

QUINLAN Right. While these instances—where you have to provide for failover and error recovery—may have been acceptable in the batch situation, they're definitely not OK from a latency point of view for a user-facing application. Another issue here is that there are places in the design where we've tried to optimize for throughput by dumping thousands of operations into a queue and then just processing through them. That leads to fine throughput, but it's not great for latency. You can easily get into situations where you might be stuck for seconds at a time in a queue just waiting to get to the head of the queue.

Our user base has definitely migrated from being a MapReduce-based world to more of an interactive world that relies on things such as BigTable. Gmail is an obvious example of that. Videos aren't quite as bad where GFS is concerned because you get to stream data, meaning you can buffer. Still, trying to build an interactive database on top of a file system that was designed from the start to support more batch-oriented operations has certainly proved to be a pain point.

MCKUSICK How exactly have you managed to deal with that?

QUINLAN Within GFS, we've managed to improve things to a certain degree, mostly by designing the applications to deal with the problems that come up. Take BigTable as a good concrete example. The BigTable transaction log is actually the biggest bottleneck for getting a transaction logged. In effect, we decided, "Well, we're going to see hiccups in these writes, so what we'll do is to have two logs open at any one time. Then we'll just basically merge the two. We'll write to one and if that gets stuck, we'll write to the other. We'll merge those logs once we do a replay—if we need to do a replay, that is." We tended to design our applications to function like that—which is to say they basically try to hide that latency since they know the system underneath isn't really all that great.

The guys who built Gmail went to a multihomed model, so if one instance of your Gmail account got stuck, you would basically just get moved to another data center. Actually, that capability was needed anyway just to ensure availability. Still, part of the motivation was that they wanted to hide the GFS problems.

MCKUSICK I think it's fair to say that, by moving to a distributed-master file system, you're definitely going to be able to attack some of those latency issues.

QUINLAN That was certainly one of our design goals. Also, BigTable itself is a very failure-aware system that tries to respond to failures far more rapidly than we were able to before. Using that as our metadata storage helps with some of those latency issues as well.

The engineers who worked on the earliest versions of GFS weren't particularly shy about departing from traditional choices in file-system design whenever they felt the need to do so. It just so happens that the approach taken to consistency is one of the aspects of the system where this is particularly evident.

Part of this, of course, was driven by necessity. Since Google's plans rested largely on massive deployments of commodity hardware, failures and hardware-related faults were a given. Beyond

that, according to the original GFS paper, there were a few compatibility issues. "Many of our disks claimed to the Linux driver that they supported a range of IDE protocol versions but in fact responded reliably only to the more recent ones. Since the protocol versions are very similar, these drives mostly worked but occasionally the mismatches would cause the drive and the kernel to disagree about the drive's state. This would corrupt data silently due to problems in the kernel. This problem motivated our use of checksums to detect data corruption."

That didn't mean just any checksumming, however, but instead rigorous end-to-end checksumming, with an eye to everything from disk corruption to TCP/IP corruption to machine backplane corruption.

Interestingly, for all that checksumming vigilance, the GFS engineering team also opted for an approach to consistency that's relatively loose by file-system standards. Basically, GFS simply accepts that there will be times when people will end up reading slightly stale data. Since GFS is used mostly as an append-only system as opposed to an overwriting system, this generally means those people might end up missing something that was appended to the end of the file after they'd already opened it. To the GFS designers, this seemed an acceptable cost (although it turns out that there are applications for which this proves problematic).

Also, as Gobioff explained, "The risk of stale data in certain circumstances is just inherent to a highly distributed architecture that doesn't ask the master to maintain all that much information. We definitely could have made things a lot tighter if we were willing to dump a lot more data into the master and then have it maintain more state. But that just really wasn't all that critical to us."

Perhaps an even more important issue here is that the engineers making this decision owned not just the file system but also the applications intended to run on the file system. According to Gobioff, "The thing is that we controlled both the horizontal and the vertical—the file system and the application. So we could be sure our applications would know what to expect from the file system. And we just decided to push some of the complexity out to the applications to let them deal with it."

Still, there are some at Google who wonder whether that was the right call if only because people can sometimes obtain different data in the course of reading a given file multiple times, which tends to be so strongly at odds with their whole notion of how data storage is supposed to work.

MCKUSICK Let's talk about consistency. The issue seems to be that it presumably takes some amount of time to get everything fully written to all the replicas. I think you said something earlier to the effect that GFS essentially requires that this all be fully written before you can continue.

QUINLAN That's correct.

MCKUSICK If that's the case, then how can you possibly end up with things that aren't consistent?

QUINLAN Client failures have a way of fouling things up. Basically, the model in GFS is that the client just continues to push the write until it succeeds. If the client ends up crashing in the middle of an operation, things are left in a bit of an indeterminate state.

Early on, that was sort of considered to be OK, but over time, we tightened the window for how long that inconsistency could be tolerated, and then we slowly continued to reduce that. Otherwise, whenever the data is in that inconsistent state, you may get different lengths for the file. That can lead to some confusion. We had to have some backdoor interfaces for checking the consistency of the file data in those instances. We also have something called RecordAppend, which is an interface

designed for multiple writers to append to a log concurrently. There the consistency was designed to be very loose. In retrospect, that turned out to be a lot more painful than anyone expected.

MCKUSICK What exactly was loose? If the primary replica picks what the offset is for each write and then makes sure that actually occurs, I don't see where the inconsistencies are going to come up.

QUINLAN What happens is that the primary will try. It will pick an offset, it will do the writes, but then one of them won't actually get written. Then the primary might change, at which point it can pick a different offset. RecordAppend does not offer any replay protection either. You could end up getting the data multiple times in the file.

There were even situations where you could get the data in a different order. It might appear multiple times in one chunk replica, but not necessarily in all of them. If you were reading the file, you could discover the data in different ways at different times. At the record level, you could discover the records in different orders depending on which chunks you happened to be reading.

MCKUSICK Was this done by design?

QUINLAN At the time, it must have seemed like a good idea, but in retrospect I think the consensus is that it proved to be more painful than it was worth. It just doesn't meet the expectations people have of a file system, so they end up getting surprised. Then they had to figure out work-arounds.

MCKUSICK In retrospect, how would you handle this differently?

QUINLAN I think it makes more sense to have a single writer per file.

MCKUSICK All right, but what happens when you have multiple people wanting to append to a log?

QUINLAN You serialize the writes through a single process that can ensure the replicas are consistent.

MCKUSICK There's also this business where you essentially snapshot a chunk. Presumably, that's something you use when you're essentially replacing a replica, or whenever some chunkserver goes down and you need to replace some of its files.

QUINLAN Actually, two things are going on there. One, as you suggest, is the recovery mechanism, which definitely involves copying around replicas of the file. The way that works in GFS is that we basically revoke the lock so that the client can't write it anymore, and this is part of that latency issue we were talking about.

There's also a separate issue, which is to support the snapshot feature of GFS. GFS has the most general-purpose snapshot capability you can imagine. You could snapshot any directory somewhere, and then both copies would be entirely equivalent. They would share the unchanged data. You could change either one and you could further snapshot either one. So it was really more of a clone than what most people think of as a snapshot. It's an interesting thing, but it makes for difficulties—especially as you try to build more distributed systems and you want potentially to snapshot larger chunks of the file tree.

I also think it's interesting that the snapshot feature hasn't been used more since it's actually a very powerful feature. That is, from a file-system point of view, it really offers a pretty nice piece of functionality. But putting snapshots into file systems, as I'm sure you know, is a real pain.

MCKUSICK: I know. I've done it. It's excruciating—especially in an overwriting file system.

QUINLAN Exactly. This is a case where we didn't cheat, but from an implementation perspective, it's hard to create true snapshots. Still, it seems that in this case, going the full deal was the right decision. Just the same, it's an interesting contrast to some of the other decisions that were made early on in terms of the semantics.

All in all, the report card on GFS nearly 10 years later seems positive. There have been problems and shortcomings, to be sure, but there's surely no arguing with Google's success and GFS has without a doubt played an important role in that. What's more, its staying power has been nothing short of remarkable given that Google's operations have scaled orders of magnitude beyond anything the system had been designed to handle, while the application mix Google currently supports is not one that anyone could have possibly imagined back in the late '90s.

Still, there's no question that GFS faces many challenges now. For one thing, the awkwardness of supporting an ever-growing fleet of user-facing, latency-sensitive applications on top of a system initially designed for batch-system throughput is something that's obvious to all.

The advent of BigTable has helped somewhat in this regard. As it turns out, however, BigTable isn't actually all that great a fit for GFS. In fact, it just makes the bottleneck limitations of the system's single-master design more apparent than would otherwise be the case.

For these and other reasons, engineers at Google have been working for much of the past two years on a new distributed master system designed to take full advantage of BigTable to attack some of those problems that have proved particularly difficult for GFS.

Accordingly, it now seems that beyond all the adjustments made to ensure the continued survival of GFS, the newest branch on the evolutionary tree will continue to grow in significance over the years to come. Q

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org