Paper Trail

Computer Systems, Distributed Algorithms and Databases

The Google File System

It's been a little while since my last technically meaty update. One system that I've been looking at a fair bit recently is Hadoop, which is an open-source implementation of Google's MapReduce. For me, the interesting part is the large-scale distributed filesystem on which it runs called HDFS. It's well known that HDFS is based heavily on its Google equivalent.

In 2003 Google published a paper on their Google File System (GFS) at SOSP, the Symposium on Operating Systems Principles. This is the same venue at which Amazon published their Dynamo work, albeit four years earlier. One of the lecturers in my group tells me that SOSP is a venue where "interesting" is rated highly as a criterion for acceptance, over other more staid conferences. So what, if anything, was interesting about GFS? Read on for some details…

Filesystems

Filesystems are a integral component of most operating systems that mediate access to persistently stored data. When you save a file to a magnetic hard disk, it gets recorded as a logical series of 1s and 0s, amongst a sea of other files similarly represented. The filesystem is the part of the operating system that makes sense of these 1s and 0s and is able to recover the structure in terms of files and folders that was present when the data were written.

Of course this, as any other one paragraph introduction to such a broad subject must be, is a simplification. Filesystems don't tend to deal in 1s and 0s, that would be far too messy. Instead they are supported by a device driver that typically speaks in terms of *blocks* – large chunks of data – which takes care of actually writing the blocks to disk. But the filesystem is responsible for determining what those blocks 'mean' in the context of files and folders.

Different filesystems have different characteristics. Some filesystems are extremely simple, lightweight and fast. Others are more complex, but can recover from some disk corruption. Some are more appropriate to workstation usage patterns, whereas some are tailored towards, say, the requirements of a disk-backed SQL database.

GFS

GFS was designed to be a highly distributed filesystem with support targeted towards extremely large files. We can read between the lines and infer that GFS might be used to store the results of Google's web crawlers, and as a storage layer for MapReduce. However, that's all we can do, because the GFS paper, while very solidly written, has nearly no actual justification for its design beyond 'we needed it to do *x*, so we made this design decision'. Trade secrets are well and good, but they don't allow a proper evaluation of the file system because

the trade-offs that are being made aren't exposed to the reader. Still, it's easy enough to imagine 'being Google' and having this particular set of requirements, and it was with that suspension of disbelief in mind that I read this paper.

One of the particular assumptions that they make, and one which I think is terribly important for modern distributed systems, is that failure is the norm. They make use of sufficiently large numbers of cheap nodes that the expected failure rate per unit time is actually reasonably close to 1. This means that not only must the system be highly-fault tolerant from the ground up, the cost of dealing with faults must be kept reasonably low. This is in contrast to other distributed systems where the likelihood of failure is considered sufficiently low that the designer is willing to 'pay' a high cost for recovering from that failure (say, at worst, restarting the system and replaying the logs) in order to cut the cost of normal behaviour.

Another assumption – pulled from their own requirements which we're not allowed to see – is that the typical write case is an append to a file, rather than an overwrite. Therefore they go all out to optimise this case, to the point of compromising some more standard usual filesystem semantics. We can only assume that this really is the common case (again, think web crawlers and appends to the results of the most recent crawl and we can believe this).

Design Decisions

In order to meet these goals of high distribution, tolerance to high failure rates, fast-path appends and huge files the designers provided their own API to the filesystem. Why is this a surprising design decision? Because every filesystem is supposed to support the POSIX file and directory standard in order to ensure that different filesystems can be used by applications with one set of commands to open, close, write etc. This ensures that better, faster, stronger filesystems can be used by old applications with no need to port (you wouldn't want to have to rewrite "cp" everytime a new filesystem was released) and is a strong point in favour of abstraction.

However, the designers of GFS felt that their design would be hampered by strict conformance to both POSIX semantics and the API. Therefore they designed a (very similar) API that their applications would be custom written to take advantage of. This makes a good deal of sense; it's not highly likely that GFS would suddenly be swapped out for a completely different highly-distributed filesystem (they're not ten-a-penny) and strongly coupling Google's internal applications to their filesystem doesn't therefore lose them much flexibility.

The deviations from POSIX are not extreme. API calls to create, delete, open, close, read and write files are still present. The only difference in semantics comes when writing to a file. POSIX ensures that two writes to a file will be serialised, that is that they will be written one after the other. GFS however allows for the possibility that, under certain conditions, two writes will conflict and half of each will succeed. The probability of this is rather small, and relies on an unlikely concurrence of several conditions. Similarly, it is possible that a read call might return 'stale', or old, data. Again, the window for this is very small.

Two further calls are added: *snapshot* allows a copy of the current state of part of the filesystem directory hierarchy to be backed up, and *record append* which implements an atomic append-to-file operation. Append itself has weird semantics that ensure that the records are appended *at least once* to the file. Therefore the resulting file may have duplicates of some information, and the onus is on the application to sort out what is original and what is not. Readers following along at home might be reminded of Dynamo, which takes a similar approach to worst-case inconsistencies. However, append operations only add junk padding to files, and this

can often be detected through checksums and other consistency checks.

Blocks, or 'chunks' as they're called in the paper, are set at 64Mb size. This is not a trivial design decision: large blocks mean less metadata, fewer lookups for streaming reads and reduced need to spend time opening TCP connections with several different replicas. However, a large chunk size can mean wasted storage for small files (although this is mitigated by having each replica grow its chunk to the correct size lazily), and the possibility of 'hot spots' occurring if many clients are accessing the same small file, since they will all be accessing the same chunk. In practice the authors report that this is uncommon, although they did have to deal with one instance of an executable file stored in GFS being accessed by hundreds of machines that were all starting the executable at the same time. Manually upping the replication factor fixed this.

Architecture

Fundamentally, GFS is quite simply put together. Nodes can play one of two roles in a GFS cluster: they can be *data nodes*, which are actually responsible for physically storing file chunks, or they can be the *master node* which keeps track of all the metadata for the filesystem, including records of which node has which block, mappings from files to blocks and a variety of other bookkeeping data. The master node decides where to put chunks, so a file may be split across many different machines.

There is only one master node per GFS instance. The paper suggests that this is for simplicity, and also for flexibility in the sense that a single master node must have global knowledge of the entire filesystem, and can use this to make, for example, sophisticated choices about chunk placement. The obvious problem is that the master then represents a single point of failure: if it goes down, the entire cluster is unavailable. The paper describes in detail how the master's state is meticulously recorded so that another machine may take over in the case of a failure. What isn't explained is why running the master as a simple distributed state machine wasn't an appropriate choice. We know that Google run Paxos for Chubby, their lock manager (although perhaps this paper predates Chubby), which does similar metadata management. Perhaps the cost of synchronising several nodes in a state machine was prohibitive to latency requirements.

Chunks are replicated on some configurable number of data nodes. The paper uses 3 replicas, which is presumably a value drawn from whatever failure model they have for their datacentre. Clients are pointed to a data node by the master node, which then steps out of the critical path for reads and writes, as it's important to keep load on the master to a minimum for availability reasons. A client only directly interacts with one data node per chunk – any changes to that chunk are pushed to the other replicas without the client's knowledge.

Reads and Writes (and Appends)

A typical (simplified) read interaction with a GFS cluster by a client application goes something like this:

1. The client contacts the master with a file path and name, plus an offset in the file for reading
2. The master responds with a data node address, picked from the set of live data nodes that it knows to have replicated the chunk that contains the required offset
3. The client issues a read directly to the data node.

Writes are conducted in much the same way, except that there is a bit of extra scaffolding required to make sure that writes aren't issued concurrently at two different replicas. This is achieved through *leases*, which are granted

to a data node – called the *primary* – by the master when it is being written to. There is only one lease per chunk, so that if two write requests go to the master both see the same lease denoting the same node. The responsibility then falls on the primary to ensure that writes are serialised correctly.

As a write takes place, the primary pushes the data to another replica, which then pushes it another replica and so on. When all live replicas have responded, the primary issues a 'write' message (which is akin to a commit in this two-phase commit protocol). Any errors at any stage in this process are met with retries and eventual failure. It's not clear if GFS detects the failure of a replica during a write operation and allows the write to succeed by going only to the remaining live nodes, or if a write only succeeds if it is committed at every replica that was live when the write started.

The problem of inconsistency rears its head when clients try and write across chunk boundaries. Because there is no mechanism to lease multiple chunks at once, there is no communication between primaries and the possibility is that two concurrent writes may be serialised differently by the different primaries. This leaves data *consistent* – all replicas will have the same chunks – but *undefined*, as the resultant state can not be the result of any legitimate serialisation of writes.

One way you could solve this is by assigning *writeids* to writes at one primary, ordered by the serialisation order. These could be passed to the second chunk primary, effectively saying "I have previously been writing at chunk x", and allowing the second primary to effect the same write order. Since the bounds of a write are known at time of issuance, clients can go to the chunk containing the lowest offset first to obtain this ordering. If the writes to the next chunk arrive out of order, the primary for that chunk can simply discard any conflicting writes that arrive late.

There is the possibility for stale data to exist on a replica that failed before a write was issued, but came back up straight afterwards. When the replica restarts, it checks with the master that its version numbers for each chunk are up to date. The master, knowing that the replica is stale, will refuse to point clients to that replica until it's been resynchronised. However, if a client has cached the address of the recovered replica, it might read from a chunk before it's resynchronised. It's not clear why the replica itself would allow this, knowing that it's out of sync, but again it seems that for Google's application domain responsiveness and simplicity of protocol is more important than consistency. This seems like a perfectly reasonable trade-off, but it would still have been good to find out why it was made.

Append operations follow almost exactly the same logic. The only difference is that if an append operation fails at one replica, it is retried at every replica with the possible consequence that the appended data are written twice at some chunk. A subtle point to note is that is the *client* that issues the retry, not the primary. This is because if the primary should fail, it is unavailable to orchestrate the retry. If a replica does fail an append, it pads the chunk with useless data so that the retried append is written at the same offset at all replicas.

Exactly how the replica that fails knows how to do this is not made clear. Presumably, the assumption is that the write has failed because the physical disk might be corrupt at that replica location, so despite the write failing the replica knows that it must act as though it has succeeded. There is a distinction here between replicas that fail – and therefore do not receive and act upon an append – and those that have difficulties in enacting the append. Replicas that have totally failed and recovered spend some time resynchronising their state to discover any operations that they might have missed. Replicas that simply fail a write do not resynchronise, rather they pad (if the write was an append) and then wait for the retry.

Replicas communicate regularly with the master through heartbeat messages that both indicate liveness and contain a digest of which chunks they are managing. The master doesn't maintain a persistent list of which chunks are on which data nodes – when a data node becomes live (or a master restarts) the data node reports all the chunks that it has to the master. The idea behind this is that only the data node has a good idea of which chunks it actually has – any definitive list the master has is guaranteed to get out of sync.

Checkpoints and Snapshots

The master itself keeps a transaction log which is replicated to several back-up masters every time it is written to. Only metadata is written to this log – the results of file creation or deletion operations. If the master should fail its operation can be recovered by a back-up master which can simply replay the log to get to the same state. However, this can be very slow, especially if the cluster has been alive for a long time and the log is very long. To help with this issue, the master's state is periodically serialized to disk and then replicated so that on recovery a master may load the checkpoint into memory, replay any subsequent operations in the log, and be available again very quickly. All metadata is held by the master in main memory – this avoids latency problems caused by disk writes, as well as making scanning the entire chunkspace (e.g. for garbage collection) very efficient. The paper quotes less than 64 bytes of metadata for a 64MB chunk, so the addressable range for even a small amount of main memory is in the terabyte range and above.

Checkpointing the master state is distinct from the second unusual operation that GFS supports, *snapshot*. A snapshot is a copy of some subtree of the global namespace as it exists at a given point in time. The paper suggests that this is used to efficiently branch two versions of the same data. Snapshots in GFS are initially zero-copy, that is copies are only made when copied chunks are dirtied by write operations. The master orchestrates this by revoking all leases for copied chunks, which ensures that the next write to these chunks goes through the master. The snapshot is then made by duplicating the appropriate metadata on the master, but the duplicated files still point to the original chunks. When a write is made to one of these chunks, the master detects that it is a copy-on-write chunk by examining its reference count (which will now be > 1), and telling data nodes to make a copy of it. These local copies are made to avoid copying the chunk over the network – it's possible that this can lead to a doubled chunk load in a pathological situation for a given replica. Once the copy is complete, the master issues a lease for the new copy as normal, and the write proceeds.

Conclusions

The GFS paper was great for perhaps two main reasons. Primarily, it describes some fairly interesting technology. GFS has a fairly clean and apparently efficient design. It would be nice to know a bit more about workloads. The evaluation section describes some micro-benchmarks and then describes typical workload, but it doesn't truly characterise the kind of use cases they are expecting. This leaves the reader a little unsure about the extent to which the tail is wagging the dog. I think that it might have been possible to clean up the possible inconsistencies that their model implies. Sadly, there doesn't seem to be any measurement of how often inconsistencies occur, so no understanding is attempted of how important the problem is is achieved. That said, the paper is admirably clear on most aspects of the design, and precise in the sense that after reading the educated reader might feel able to construct an implementation (and indeed, this is what the HDFS guys did!).

The second reason is that it was – I think – the first high profile systems paper published by Google (notwithstanding the original search engine paper from when the founders were at Stanford). Throwing the doors

open – or at least allowing us to peer through the windows – and demonstrating that non-trivial, interesting systems work was being done 'in the real world' not only encouraged other companies to do the same, it gave researchers who wanted it a new set of target applications and domains for their work.

It also gave a lot of desperate systems PhD students hope of finding a job.

Published: October 1, 2008
Filed Under: computer science, Distributed systems, Operating systems, Paper Walkthrough
Tags: Distributed systems : filesystem : gfs : google : paper review

3 Responses to "The Google File System"

1. *Paper Trail on GFS - A River of Words* says:
   October 7, 2008 at 11:14 pm

   [...] always entertaining Paper Trail recently did a write up deciphering the Google paper describing the Google File System – a great read if you're [...]

2. *Diagrams, and the state of the union at Paper Trail* says:
   January 21, 2009 at 5:29 pm

   [...] systems research, with a particular emphasis on real systems that exist and work. Hence the GFS, BigTable and PNUTS articles, and the recent series on OSDI papers. Part of my day job is being [...]

3. *What are some good resources for learning about distributed computing? - Quora* says:
   June 8, 2011 at 6:45 pm

   [...] Research papers?also http://www.columbia.edu/~ak2834/… , http://www.cs.rutgers.edu/~muthu… , http://the-paper-trail.org/blog/…)Setup account with Amazon AWS/EC2/S3/EBS and experiment with running Hadoop on a cluster with large [...]

Leave a Comment

Name: *Required* Email: *Required, not published*

Homepage:

Comment:

Post Comment

« Previous Post

Next Post »

Elsewhere

Search