How Google Serves Data from Multiple Datacenters

Monday, August 24, 2009 at 1:54PM

Todd Hoff in Example, Strategy, distributed systems

Update: Streamy Explains CAP and HBase's Approach to CAP. *We plan to employ inter-cluster replication, with each cluster located in a single DC. Remote replication will introduce some eventual consistency into the system, but each cluster will continue to be strongly consistent.*

Ryan Barrett, Google App Engine datastore lead, gave this talk Transactions Across Datacenters (and Other Weekend Projects) at the Google I/O 2009 conference.

While the talk doesn't necessarily break new technical ground, Ryan does an excellent job explaining and evaluating the different options you have when architecting a system to work across multiple datacenters. This is called **multihoming**, operating from multiple datacenters simultaneously.

As multihoming is one of the most challenging tasks in all computing, Ryan's clear and thoughtful style comfortably leads you through the various options. On the trip you learn:

- The different **multi-homing options** are: Backups, Master-Slave, Multi-Master, 2PC, and Paxos. You'll also learn how they each fair on support for consistency, transactions, latency, throughput, data loss, and failover.
- Google App Engine **uses master/slave replication** between datacenters. They chose this approach in order to provide:

- lowish latency writes
- datacenter failure survival
- strong consistency guarantees.

- **No solution is all win**, so a compromise must be made depending on what you think is important. A major Google App Engine goal was to provide a strong consistency model for programmers. They also wanted to be able to survive datacenter failures. And they wanted write performance that wasn't too far behind a typical relational database. These priorities guided their architectural choices.
- In the future they hope to **offer optional models** so you can select Paxos, 2PC, etc for your particular problem requirements (Yahoo's PNUTS does something like this).

There's still a lot more to learn. Here's my gloss on the talk:

Consistency - What happens happens after you read after a write?

Read/write data is one of the hardest kinds of data to run across datacenters. Users a expect a certain level of reliability and consistency.

- **Weak** - it might be there, might not. Best effort. Like memcached. It's OK to drop for some applications like Voip, live video, and multiplayer games. You care more about where things are now, not where they where. For data this is not good.
- **Eventual** - You eventually see the stuff you wrote, just not right away. Email is a good example. You send it but it doesn't arrive right away, but it gets there, eventually. DNS change propagation, SMTP, Amazon S3, SimpleDB, search engine indexing are all of this type. There's a delay after a write when a read won't see what was written, but the writes eventually push through. Still not ideal for data.
- **Strong** - The ideal solution for a structured data system. You get what you put it in. Simplest to program against and think about. Any read after a write will return what was written. AppEngine, file systems, Microsoft Azure, and RDBMSes work this way.
- Once we move data across datacenters what consistency guarantees do

we have? We can give up some guarantees, but we should know what we are getting.

Transactions - Extended form of consistency across multiple operations.

- Transaction Properties: Correctness, consistency, enforce variants, ACID.
- Example: bank transaction. Transfer money from A to B. Subtract money from A and add to B. These happen at different times. What happens if another transfer happens for A in-between? What happens if there's a failure? What happens of program reads from A or B? You want guarantees. On a crash will money added to B still be added to B? Will money taken from A still be taken from A? You don't want to lose or create money.
- When you start operating across datacenters it's even harder to enforce transactions because more things can go wrong and operations have high latency.

Why Operate in Multiple Datacenters?

- **Sh*t happens** - datacenters fail for any number of reasons.
- **Performance** - geolocality allows operations to be moved closer to the user. The speed of light limits limits how fast data can be transferred and becomes significant when operating across the world. Going through multiple router hops also slows traffic. So closer is better and you can only be closer if your data is near the user which requires operating in multiple datacenters. CDNs do this for you, especially for more static data. They put data everywhere.

Why Not Operate in Multiple Datacenters?

- Operating in a single datacenter is easy: Low cost bandwidth. Low

latency. High bandwidth. Easy operations. Easier code.

- Operating in multiple datacenters is hard: high cost, high latency, low latency, difficult operations, harder code.
- It's especially hard if you have a read/write structured data system where you accept writes from more than one location. You have consistency problems. Maintaining consistency in the face of the distances and failures is non-trivial.

Your Different Architecture Options

- **Single Datacenter**. Don't bother operating in mutiple datacenters. This is the easiest option and is what most people do. But datacenters fail, you could lose data, and your site could go down.
- **Bunkerize**. Create a Maginot Line for the Ultimate Datacenter. Make sure your datacenter doesn't ever go down. SimpleDB and Azure use this strategy.
- **Single Master**. Pick a master datacenter that writes go to and other sites replicate to. The replicates sites off read-only services.

- Better, but not great.
- Data are usually replicated asynchronously so there's a window of vulnerability for loss.
- Data in your other datacenters may not be consistent on failure.
- Popular with financial institutions.
- You get geolocation to serve reads. Consistency depends on the technique. Writes are still limited to one datacenter.

- **Multi-Master**. True multihoming. The Holy Grail. All datacenters are serving reads and writes. All data is consistent. Transactions just work. This is really hard.

- So some choose to do it with just two datacenters. NASDAQ has two datacenters close together (low latency) and perform a two-phase commit on every transaction, but they have very strict latency requirements.
- Using more than two datacenters is fundamentally harder. You pay for it

with queuing delays, routing delays, speed of light. You have to talk between datacenters. Just fundamentally slower with a smaller pipe. You may pay for with capacity and throughput, but you'll definitely pay in latency.

How Do You Actually Do This?

What are the techniques and tradeoffs of different approaches? Here's the evaluation matrix:

	Backups	M/S	MM	2PC	Paxos
Consistency	Weak	Eventual	Eventual	Strong	Strong
Transactions	No	Full	Local	Full	Full
Latency	Low	Low	Low	High	High
Throughput	High	High	High	Low	Medium
Data loss	Lots	Some	Some	None	None
Failover	Down	Read-only	Read/Write	Read/Write	Read/Write

- M/S = master/slave, MM - multi-master, 2PC - 2 Phase Commit
- What kind of consistency, transactions, latency throughput do we get for a particular approach? Will we lose data on failure? How much will we lose? When we failover for maintenance or we want to move things, say decommissioning a datacenter, how well do we do that, how well do the techniques support it?

- **Backups** - Make a copy of your data that's secret and safe. Generally weak consistency. Usually no transactions. Used for the first internal datastore launch. Not good enough for a production system. Lose data since last backup. You are down while restoring a backup to another datacenter.

- **Master/Slave Replication** - Writes to a master are also written to one or more slaves.
- Replication is asynchronous so good for latency and throughput.
- Weak/eventual consistency unless you are very careful.
- You have multiple copies in the datacenters, so you'll lose a little data on failure, but not much. Failover can go read-only until the master has been moved to another datacenter.
- Datastore currently uses this mechanism. Truly multihoming adds latency because you have to add the extra hop between datacenters. App Engine is already slow on writes so this extra hit would be painful. M/S gives you most of the benefits of better forms while still offering lower latency writes.

- **Multi-Master Replication** - support writes from multiple datacenters simultaneously.
- You figure out how to merge all the writes later when there's a conflict. It's like asynchronous replication, but you are serving writes from multiple locations.
- Best you can do is Eventual Consistency. Writes don't immediately go everywhere. This is a paradigm shift here. We've assumed with a strongly consistent system that backup and M/S that they don't change anything. They are just techniques to help us multihome. Here it literally changes how the system runs because the multiple writes must be merged.
- To do the merging you must find away to serialize, impose an ordering on all your writes. There is no global clock. Things happen in parallel. You can't ever know what happens first. So you make it up using timestamps, local timetamps + skew, local version numbers, distributed consensus protocol. This is the magic and there are a number of ways to do it.
- There's no way to do a global transaction. With multiple simultaneous writes you can't guarantee transactions. So you have to figure out what to

do afterward.
- AppEngine wants strong consistency to make building applications easier, so they didn't consider this option.
- Failover is easy because each datacenter can handle writes.

- **Two Phase Commit (2PC)** - protocol for setting up transactions between distributed systems.
- Semi-distributed because there's always a master coordinator for a given 2PC transaction. Because there are so few datacenters you tend to go through the same set of master coordinators.
- It's synchronous. All transactions are serialized through that master which kills your throughput and increases latency.
- Never serious considered this option because write throughput is very important to them. No single point of failure or serialization point would work for them. Latency is high because of the extra coordination. Writes can be in the 200msec area.
- This option does work though. You write to all datacenters or nothing. You get strong consistency and transactions.
- Need N+1 datacenters. If you take one down then you still have N to handle your load.

- **Paxos** - A consensus protocol where a group of independent nodes reach a majority consensus on a decision.
- Protocol: there's a propose step and then an agree step. You only need a majority of nodes to agree to say something is persisted for it to be considered persisted.
- Unlike 2PC it is fully distributed. There's no single master coordinator.
- Multiple transactions can be run in parallel. There's less serialization.
- Writes are high latency because of the 2 extra round coordination trips required in the protocol.
- Wanted to do this, but the they didn't want to pay the 150msec latency hit to writes, especially when competing against 5msec writes for

RDBMSes.
- They tried using physcially close datacenters but the built-in multi-datacenter overhead (routers, etc) was too high. Even in the same datacenter was too slow.
- Paxos is still used a ton within Google. Especially for lock servers. For coordinating anything they do across datacenters. Especially when state is moved between datacenters. If your app is serving data in one datacenter and it should be moved to another that coordination is done through Paxos. It's used also in managing memcache and offline processing.

Miscellaneous

- Entity Groups are the unit of consistency in AppEngine. Operations are serialized on Entity Groups. The log for each commit to an entity group is replicated. This maintains consistency and provides transactions. Entity Groups are essentially shards. Sharding enables scaling because it allows you to handle a lot of writes. Datastore shards in entity group size chunks. BuddyPoke has 40 million users, each of which has an entity group. That's 40 million different shards.
- Eating your own dog food is a strategy used a lot at Google. Iterate and make people use new features internally. Using a ton of stuff that's very early. You can iterated many many times so that improves it before you are ready to launch.
- They see relational databases in the datacenter as their competition as much as Azure and SimpleDB. Inserts into RDBMS are in low milliseconds. Writes into AppEngine are 30-40 msecs. Reads are fast. They like this trade-off because on the web reads vastly out number writes.

Discussion

A few things I wondered through the talk. Did they ever consider a distributed MVCC approach? That might be interesting and wasn't

addressed as an option. Clearly at Google scale an in-memory data grid isn't yet appropriate.

A preference for the strong consistency model was repeatedly specified as a major design goal because this makes the job of the programmer easier. A counter to this is that the programming model for Google App Engine is already very difficult. The limits and the lack of traditional relational database programming features put a lot of responsibility back on the programmer to write a scalable app. I wonder if giving up strong consistency would have been such a big deal in comparison?

I really appreciated the evaluation matrix and the discussion of why Google App Engine made the choices they did. Since writes are already slow on Google App Engine they didn't have a lot of headroom to absorb more layers of coordination. These are the kinds of things developers talk about in design meetings, but they usually don't make it outside the cubicle or conference room walls. I can just hear Ryan, with voiced raised, saying "Why can't we have it all!" But it never seems we can have everything. Many thanks to Ryan for sharing.

Related Articles

- Slides for the Talk
- ZooKeeper - A Reliable, Scalable Distributed Coordination System
- Yahoo!'s PNUTS Database: Too Hot, Too Cold or Just Right?
- Paper: Consensus Protocols: Paxos by Henry Robinson
- Paper: Consensus Protocols: Two-Phase Commit by Henry Robinson
- Paper: Dynamo: Amazon’s Highly Available Key-value Store
- Are Cloud Based Memory Architectures the Next Big Thing?