Megastore notes

Large-scale database/storage service for google app engine; to support a wide variety of applications, such as email, docs, blogs, facebook.

Multiple data center support – so can hide data center outages from end users.

Some aspects will seem very familiar: they do testing using a framework like ours, that supports automated randomized message ordering, with reproducible behavior once you have found a bug, and replay once you have fixed the bug (so you can verify it stays fixed). System is very complex, so this kind of testing is essential.

Also, optimistic concurrency control – application transactions are checked at the point when they are applied, and aborted if they conflict with an earlier transaction.

ACID semantics within a specific name space ("entity group") – you can think of this as providing the ability to create your own transactional file (or object store) system, where transactions within the file system are ACID (that is, linearizable), but operations across file systems might be looser (eventually consistent, or might be ACID if you pay the performance penalty of running 2pc). Exactly once message delivery for asynch communication between these different name systems.

Builds on top of BigTable, for storage within a data center. Each entity group/name space stored as a BigTable row, including the recovery log! So they automatically get replicated, highly available storage within a data center.

Paxos for providing the ACID semantics across data centers. This would normally be slow, but they adapt Paxos to allow for fast reads and writes in the normal case of no failure.

This is where it gets kind of interesting.

Most operations are reads; most operations are to a nearby data center. So how do they keep reads local? Mostly by making the write operations slower.

First, rotating leader. Each transaction elects the next leader, in rotating fashion. So if you have a local operation to do, you just wait for your data center to get its turn.

If the leader fails, fall back on multiple proposers, but to keep it simple, they just do exponential backoff in delay between proposers.

Leader always gets to use proposal #0, and so it can skip the prepare step!

Operation is committed when a majority of data centers accept, but they don't make it visible until every participant either accepts or is invalidated. In the meantime, read-only transactions (such as "display current email state") can continue without waiting for the decision as to what the next item in the write log is, since it is serializable to apply them to the state of the log before the write.

If a replica is unresponsive, need to invalidate it before we can declare the write completed; this can delay writes in the event of a data center failure or network disconnection (hopefully rare!), but their backstop is that data centers hold Chubby leases, and so will eventually timeout and can be declared dead by chubby, at which point everyone else can proceed.

In that case then, local reads can go ahead, as long as the local replica is up to date and hasn't been invalidated (didn't miss any updates).