Spanner, TrueTime & The CAP Theorem

Eric Brewer
VP, Infrastructure, Google

February 14, 2017

Spanner is Google's highly available global SQL database [CDE+12]. It manages replicated data at great scale, both in terms of size of data and volume of transactions. It assigns globally consistent real-time timestamps to every datum written to it, and clients can do globally consistent reads across the entire database without locking.

The CAP theorem [Bre12] says that you can only have two of the three desirable properties of:

- C: Consistency, which we can think of as serializability for this discussion;
- A: 100% availability, for both reads and updates;
- P: tolerance to network partitions.

This leads to three kinds of systems: CA, CP and AP, based on what letter you leave out. Note that you are not entitled to 2 of 3, and many systems have zero or one of the properties.

For distributed systems over a "wide area", it is generally viewed that partitions are inevitable, although not necessarily common [BK14]. Once you believe that partitions are inevitable, any distributed system must be prepared to forfeit either consistency (AP) or availability (CP), which is not a choice anyone wants to make. In fact, the original point of the CAP theorem was to get designers to take this tradeoff seriously. But there are two important caveats: first, you only need forfeit something during an actual partition, and even then there are many mitigations (see the "12 years" paper [Bre12]). Second, the actual theorem is about 100% availability, while the interesting discussion here is about the tradeoffs involved for realistic high availability.

Spanner claims to be consistent and available

Despite being a global distributed system, Spanner claims to be consistent and highly available, which implies there are no partitions and thus many are skeptical.[1] Does this mean that Spanner is a CA system as defined by CAP? The short answer is "no" technically, but "yes" in effect and its users can and do assume CA.

The purist answer is "no" because partitions can happen and in fact have happened at Google, and during (some) partitions, Spanner chooses C and forfeits A. It is technically a CP system. We explore the impact of partitions below.

Given that Spanner always provides consistency, the real question for a claim of CA is whether or not Spanner's serious users assume its availability. If its actual availability is so high that users can ignore outages, then Spanner can justify an "effectively CA" claim. This does not imply 100% availability (and Spanner does not and will not provide it), but rather something like 5 or more "9s" (1 failure in 10^5 or less). In turn, the real litmus test is whether or not users (that want their own service to be highly available)

[1] Although I work for Google, I have not worked on Spanner or TrueTime, other than to push to make both available to our Cloud customers. My intent is to provide an objective outsider's view of Spanner, but I admit bias in that I like the system in addition to working for Google.

write the code to handle outage exceptions: if they haven't written that code, then they are assuming high availability. Based on a large number of internal users of Spanner, we know that they assume Spanner is highly available.

A second refinement is that there are many other sources of outages, some of which take out the users in addition to Spanner ("fate sharing"). We actually care about the differential availability, in which the user is up (and making a request) to notice that Spanner is down. This number is strictly higher (more available) than Spanner's actual availability — that is, you have to hear the tree fall to count it as a problem.

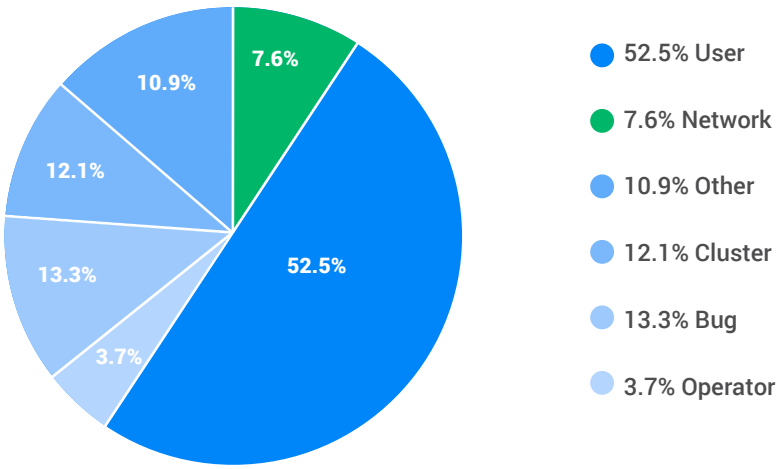
A third issue is whether or not outages are due to partitions. If the primary causes of Spanner outages are not partitions, then CA is in some sense more accurate. For example, any database cannot provide availability if all of its replicas are offline, which has nothing to do with partitions. Such a multi-replica outage should be very rare, but if partitions are significantly more rare, then you can effectively ignore partitions as a factor in availability. For Spanner, this means that when there is an availability outage, it is not in practice due to a partition, but rather some other set of multiple faults (as no single fault will forfeit availability).

Availability data

Before we get to Spanner, it is worth taking a look at the evolution of Chubby, another wide-area system that provides both consistency and availability. The original Chubby paper [Bur06] mentioned nine outages of 30 seconds or more in 700 days, and six of those were network related (as discussed in [BK14]). This corresponds to an availability worse than 5 9s (at best), to a more realistic 4 9s if we assume an average of 10 minutes per outage, and potentially even 3 9s at hours per outage.

For locking and consistent read/write operations, **modern geographically distributed Chubby cells provide an average availability of 99.99958%** (for 30s+ outages) due to various network, architectural and operational improvements. Starting in 2009, due to "excess" availability, Chubby's Site Reliability Engineers (SREs) started forcing periodic outages to ensure we continue to understand dependencies and the impact of Chubby failures.

Internally, Spanner provides a similar level of reliability to Chubby; that is, better than 5 9s. The Cloud version has the same foundation, but adds some new pieces, so it may be a little lower in practice for a while.



The pie chart above reveals the causes of Spanner incidents internally. An incident is an unexpected event, but not all incidents are outages; some can be masked easily. The chart is weighted by frequency not by impact. The bulk of the incidents **(User)** are due to user errors such as overload or misconfiguration and mostly affect that user, whereas the remaining categories could affect all users in an area. **Cluster** incidents reflect non-network problems in the underlying infrastructure, including problems with servers and power. Spanner automatically works around these incidents by using other replicas; however, SRE involvement is sometimes required to fix a broken replica. **Operator** incidents are accidents induced by SREs, such as a misconfiguration. **Bug** implies a software error that caused some problem; these can lead to large or small outages. The two biggest outages were both due to software bugs that affected all replicas of a particular database at the same time. **Other** is grab bag of various problems, most of which occurred only once.

The **Network** category, under 8%, is where partitions and networking configuration problems appear. There were no events in which a large set of clusters were partitioned from another large set of clusters. Nor was a Spanner quorum ever on the minority side of a partition. We did see individual data centers or regions get cut off from the rest of the network. We also had some misconfigurations that under-provisioned bandwidth temporarily, and we saw some temporary periods of bad latency related to hardware failures. We saw one issue in which one direction of traffic failed, causing a weird partition that had to be resolved by bringing down some nodes. So far, no large outages were due to networking incidents.

Summarizing, to claim "effectively CA" a system must be in this state of relative probabilities: 1) At a minimum it must have very high availability in practice (so that users can ignore exceptions), and 2) as this is about partitions it should also have a low fraction of those outages due to partitions. Spanner meets both.

It's the network

Many assume that Spanner somehow gets around CAP via its use of TrueTime, which is a service that enables the use of globally synchronized clocks. Although remarkable, TrueTime does not significantly help achieve CA; its actual value is covered below. To the extent there is anything special, it is really Google's wide-area network, plus many years of operational improvements, that greatly limit partitions in practice, and thus enable high availability.

First, Google runs its own private global network. Spanner is not running over the public Internet — in fact, every Spanner packet flows only over Google-controlled routers and links (excluding any edge links to remote clients). Furthermore, each data center typically has at least three independent fibers connecting it to the private global network, thus ensuring path diversity for every pair of data centers.[2] Similarly, there is redundancy of equipment and paths within a datacenter. Thus normally catastrophic events, such as cut fiber lines, do not lead to partitions or to outages.

The real risk for a partition is thus not a cut path, but rather some kind of broad config or software upgrade that breaks multiple paths simultaneously. This is a real risk and something that Google continues to work to prevent and mitigate. The general strategy is to limit the impact (or "blast radius") of any particular update, so that when we inevitably push a bad change, it only takes out some paths or some replicas. We then fix those before attempting any other changes.

[2]The actual requirement is a target availability, not a number of connections per se.

Although the network can greatly reduce partitions, it cannot improve the speed of light. Consistent operations that span a wide area have a significant minimum round trip time, which can be tens of milliseconds or more across continents. (A distance of 1000 miles is about 5 million feet, so at ½ foot per nanosecond, the minimum would be 10 ms.) Google defines "regions" to have a 2ms round-trip time, so that regional offerings provide a balance between latency and disaster tolerance. Spanner mitigates latency via extensive pipelining of transactions, but that does not help single-transaction latency. For reads, latency is typically low, due to global timestamps and the ability to use a local replica (covered below).

A model with weaker consistency could have lower update latency. However, without the long round trip it would also have a window of lower durability, since a disaster could take out the local site and delete all copies before the data is replicated to another region.

What happens during a Partition

To understand partitions, we need to know a little bit more about how Spanner works. As with most ACID databases, Spanner uses two-phase commit (2PC) and strict two-phase locking to ensure isolation and strong consistency. 2PC has been called the "anti-availability" protocol [Hel16] because all members must be up for it to work. Spanner mitigates this by having each member be a Paxos group, thus ensuring each 2PC "member" is highly available even if some of its Paxos participants are down. Data is divided into groups that form the basic unit of placement and replication.

As mentioned above, in general Spanner chooses C over A when a partition occurs. In practice, this is due to a few specific choices:

- Use of Paxos groups to achieve consensus on an update; if the leader cannot maintain a quorum due to a partition, updates are stalled and the system is not available (by the CAP definition). Eventually a new leader may emerge, but that also requires a majority.
- Use of two-phase commit for cross-group transactions also means that a partition of the members can prevent commits.

The most likely outcome of a partition in practice is that one side has a quorum and will continue on just fine, perhaps after electing some new leaders. Thus the service continues to be available, but users on the minority side have no access. But this is a case where differential availability matters: those users are likely to have other significant problems, such as no connectivity, and are probably also down. This means that multi-region services built on top of Spanner tend to work relatively well even during a partition. It is possible, but less likely, that some groups will not be available at all.

Transactions in Spanner will work as long as all of the touched groups have a quorum-elected leader and are on one side of the partition. This means that some transactions work perfectly and some will time out, but they are always consistent. An implementation property of Spanner is that any reads that return are consistent, even if the transaction later aborts (for any reason, including time outs).

In addition to normal transactions, Spanner supports snapshot reads, which are read at a particular time in the past. Spanner maintains multiple versions over time, each with a timestamp, and thus can precisely answer snapshot reads with the correct version. In particular, each replica knows the time for which it is caught up (for sure), and any replica can unilaterally answer a read before that time (unless it is way too old and has been garbage collected). Similarly, it is easy to read (asynchronously) at the same time across many groups. Snapshot reads do not need locks at all. In fact, read-only transactions are implemented as a snapshot read at the current time (at any up-to-date replica).

Snapshot reads are thus a little more robust to partitions. In particular, a snapshot read will work if:

1. There is at least one replica for each group on the initiating side of the partition, and
2. The timestamp is in the past for those replicas.

The latter might not be true if the leader is stalled due to a partition, and that could last as long as the partition lasts, since it might not be possible to elect a new leader on this side of the partition. During a partition, it is likely that reads at timestamps prior to the start of the partition will succeed on both sides of the partition, as any reachable replica that has the data suffices.

What about TrueTime?

In general, synchronized clocks can be used to avoid communication in a distributed system. Barbara Liskov provides a fine overview with many examples [Lis91].[3] For our purposes, TrueTime is a global synchronized clock with bounded non-zero error: it returns a time interval that is guaranteed to contain the clock's actual time for some time during the call's execution. Thus, if two intervals do not overlap, then we know calls were definitely ordered in real time. If the intervals overlap, we do not know the actual order.

One subtle thing about Spanner is that it gets serializability from locks, but it gets external consistency (similar to linearizability) from TrueTime. Spanner's external consistency invariant is that for any two transactions, T_1 and T_2 (even if on opposite sides of the globe):

> *if T_2 starts to commit after T_1 finishes committing, then the timestamp for T_2 is greater than the timestamp for T_1.*

Quoting from Liskov [Lis91, section 7]:

> *Synchronized clocks can be used to reduce the probability of having a violation of external consistency. Essentially the primary holds leases, but the object in question is the entire replica group. Each message sent by a backup to the primary gives the primary a lease. The primary can do a read operation unilaterally if it holds unexpired leases from a sub-majority[4] of backups. …*
>
> *The invariant in this system is: whenever a primary performs a read it holds valid leases from a sub-majority of backups. This invariant will not be preserved if clocks get out of synch.*

Spanner's use of TrueTime as the clock ensures the invariant holds. In particular, during a commit, the leader may have to wait until it is sure the commit time is in the past (based on the error bounds). This "commit wait" is not a long wait in practice and it is done in parallel with (internal) transaction communication. In general, external consistency requires monotonically increasing timestamps, and "waiting out the uncertainty" is a common pattern.

Spanner aims to elect leaders for an extended time, typically 10 seconds, by using renewable leases for elected leaders. As discussed by Liskov, every time a quorum agrees on a decision the lease is extended, as the participants just verified that the leadership is effective. When a leader fails there are two options: 1) you can wait for the lease to expire and then elect a new leader, or 2) you can restart the old leader, which might be faster. For some failures, we can send out a "last gasp" UDP packet to release the lease, which is an optimization to speed up expiration. As unplanned failures are rare in a Google data center, the long lease makes sense. The lease also ensures monotonicity of time across leaders, and enables group participants to serve reads within the lease time even without a leader.

[3]There is a direct connection here: two of the Spanner paper authors, Wilson Hsieh and Sanjay Ghemawat, were grad students in Barbara Liskov's extended group in the early 1990s. I was as well.
[4]A "sub-majority" is majority minus one, implying a majority when counting the leader as well.

However, the real value of TrueTime is in what it enables in terms of consistent snapshots. Stepping back a bit, there is a long history of multi-version concurrency-control systems (MVCC) [Ree78] that separately keep old versions and thus allow reading past versions regardless of the current transactional activity. This is a remarkably useful and underrated property: in particular, in Spanner snapshots are consistent (for their time) and thus whatever invariants hold for your system, they will also hold for the snapshot. This is true even when you don't know what the invariants are! Essentially, snapshots are taken in between consecutive transactions and reflect everything up to the time of the snapshot, but nothing more. Without transactionally consistent snapshots, it is difficult to restart from a past time, as the contents may reflect a partially applied transaction that violates some invariant or integrity constraint. It is the lack of consistency that sometimes makes restoring from backups hard to do; in particular, this shows up as some corruption that needs to be fixed by hand.[5]

For example, consider using MapReduce to perform a large analytics query over a database. On Bigtable, which also stores past versions, the notion of time is "jagged" across the data shards, which makes the results unpredictable and sometimes inconsistent (especially for the very recent past). On Spanner, the same MapReduce can pick a precise timestamp and get repeatable and consistent results.

TrueTime also makes it possible to take snapshots across multiple independent systems, as long as they use (monotonically increasing) TrueTime timestamps for commit, agree on a snapshot time, and store multiple versions over time (typically in a log). This is not limited to Spanner: you can make your own transactional system and then ensure snapshots that are consistent across both systems (or even k systems). In general, you need a 2PC (while holding locks) across these systems to agree on the snapshot time and confirm success, but the systems need not agree on anything else, and can be wildly different.

You can also use timestamps as tokens passed through a workflow. For example, if you make an update to a system, you can pass the time of that update to the next stage of the workflow, so that it can tell if its system reflects time after that event. In the case of a partition, this may not be true, in which case the next stage should actually wait if it wants consistency (or proceed if it wants availability). Without the time token, it is hard to know that you need to wait. This isn't the only way to solve this problem, but it does so in a graceful robust way that also ensures eventual consistency. This is particularly useful when the different stages share no code and have different administrators — both can agree on time with no communication.[6]

Snapshots are about the past, but you can also agree on the future. A feature of Spanner is that you can agree on the time in the future for a schema change. This allows you to stage the changes for the new schema so that you are *able* to serve both versions. Once you are ready, you can pick a time to switch to the new schema atomically at all replicas. (You can also pick the time before you stage, but then you might not be ready by the target time.) In theory at least, you can also do future operations, such as a scheduled delete or a change in visibility.

TrueTime itself could be hindered by a partition. The underlying source of time is a combination of GPS receivers and atomic clocks, both of which can maintain accurate time with minuscule drift by themselves. As there are "time masters" in every datacenter (redundantly), it is likely that both sides of a partition would continue to enjoy accurate time. Individual nodes however need network connectivity to the masters, and without it their clocks will drift. Thus, during a partition their intervals slowly grow wider over time, based on bounds on the rate of local clock drift. Operations depending on TrueTime, such as Paxos leader election or transaction commits, thus have to wait a little longer, but the operation still

[5]For comparison, in ARIES [MHL+92], snapshots are intentionally fuzzy, but you can then replay the log on a per-page basis to bring that page up to the target cutoff transaction (and logical time). This works well for recovery, but not so well for running analytics on the snapshot (since it is fuzzy).
[6]Note there is still background communication involved in clock synchronization, including GPS itself and periodic correction traffic.

completes (assuming the 2PC and quorum communication are working).

Conclusion

Spanner reasonably claims to be an "effectively CA" system despite operating over a wide area, as it is always consistent and achieves greater than 5 9s availability. As with Chubby, this combination is possible in practice if you control the whole network, which is rare over the wide area. Even then, it requires significant redundancy of network paths, architectural planning to manage correlated failures, and very careful operations, especially for upgrades. Even then outages will occur, in which case Spanner chooses consistency over availability.

Spanner uses two-phase commit to achieve serializability, but it uses TrueTime for external consistency, consistent reads without locking, and consistent snapshots.

Acknowledgements:
Thanks in particular to Spanner and TrueTime experts Andrew Fikes, Wilson Hsieh, and Peter Hochschild. Additional thanks to Brian Cooper, Kurt Rosenfeld, Chris Taylor, Susan Shepard, Sunil Mushran, Steve Middlekauff, Cliff Frey, Cian Cullinan, Robert Kubis, Deepti Srivastava, Sean Quinlan, Mike Burrows, and Sebastian Kanthak.

References:

[BK14] P. Bailis and K. Kingsbury. The Network is Reliable, *Communications of the ACM*. Vol. 57 No. 9, Pages 48-55. September 2014. Also: https://aphyr.com/posts/288-the-network-is-reliable

[Bre12] E. Brewer. CAP Twelve Years Later: How the "Rules" Have Changed, *IEEE Computer*, Vol. 45, Issue 2, February 2012. pp. 23--29.

[Bur06] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. *Proceedings of OSDI '06: Fourth Symposium on Operating System Design and Implementation*, Seattle, WA, November 2006.

[CDE+12] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, JJ Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. *Proceedings of OSDI '12: Tenth Symposium on Operating System Design and Implementation*, Hollywood, CA, October, 2012

[Hel16] P. Helland. Standing on Giant Distributed Shoulders: Farsighted Physicists of Yore were Danged Smart! *ACM Queue*, Vol. 14, Issue 2, March-April 2016.

[Lis91] B. Liskov. Practical Uses of Synchronized Clocks in Distributed Systems. *ACM Principles of Distributed Computing (PODC)*. Montreal, Canada, August 1991.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh and P. Schwartz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems,* Vol. 17, No. 1, March 1992, pp. 94-162.

[Ree78] D. Reed. NAMING AND SYNCHRONIZATION IN A DECENTRALIZED COMPUTER SYSTEM, PhD Dissertation, MIT Laboratory for Computer Science, Technical Report MIT-LCS-TR-205. October 1978 [See Section 6.3 for list of versions with timestamps]