How Google Spanner Assigns Commit Timestamps — The Secret Sauce of Its Strong Consistency

Eileen Pangu Follow
Dec 27, 2020 · 7 min read ★

Photo by Lukas Blazek on Unsplash

Spanner is Google's global scale and synchronously replicated relational database. Its research paper was first published in 2012. Spanner received much acclaim because it was the first system to distribute data at global scale and support strongly consistent distributed transactions. That seems to break the CAP theorem. But in fact, Spanner chooses consistency over availability in face of network partition. It's just that Google's data center infrastructure is so reliable that external users typically don't worry about its outages. Google now offers Spanner as a service through its cloud platform.

There have been many blog posts that try to explain how Spanner works. But most of them just acknowledge that Spanner has accurate commit timestamps, and go on discussing other aspects of its functions or architectures. Few have really tried to provide a clear insight of how its accurate commit timestamps are chosen, which is the real secret sauce of its strong consistency. The answer to that is not just "yup, they have a TrueTime API". In this blog post, I'll explore its internal logic with you.

What's the Big Deal of Accurate Commit Timestamps

First and foremost, we need to understand why accurate commit timestamps are important. In a nutshell, they bring order to the chaotic distributed systems. The fundamental challenge of distributed systems resides in the lack of synchronization among all the moving pieces. The lack of synchronization is partly by nature because things can fail frequently and unpredictably, and partly by design because the whole system would otherwise be extremely slow if all parts need to move together. Accurate commit timestamps instill a sense of ordering to the system. When all events can be tagged with the true timestamps when they happened, the system respects both the commit order and the global wall-time order. A system with that capability preserves linearizability, aka, external consistency, aka strong consistency. Concretely, once a write is committed, all reads that come after will know to reflect that because their timestamps are larger than the write timestamp. We can support a multi-version database and allow clients to trace the history at any point in time. We can even offer non-blocking atomic schema updates by assigning the update a future timestamp. To scale out performance, we can serve data from a replica as long as it's sufficiently up-to-date, which again, can be determined from the accurate commit timestamps. By the way, Spanner does all those.

But globally accurate timestamps are incredibly hard. We can't rely on machines' local clocks because clock skew is real in distributed systems. Research has shown that a local clock could drift about a few milliseconds in merely 30 seconds. That would be catastrophic for a distributed system composed of tens of thousands of machines that want to use timestamps for event ordering. Spanner manages to overcome that by building a TrueTime API. The TrueTime API does not give you an absolute time — as its name may have suggested — instead, it exposes the time uncertainty in bounded intervals, which Spanner exploits to order transactions. We'll see more later.

What's a Commit Timestamp, Exactly

Before we go on further, we need to take a moment and think about what a commit timestamp really is. A common narrative would be that it's the moment when all the writes in a transaction are written. While that definition serves its purposes in some scenarios, I prefer a different explanation in this discussion: the commit timestamp is a timestamp assigned to a transaction after it has acquired all the locks and before it releases any lock.

To appreciate its significance, see the following reasoning. Database transactions rely on two-phase locking to achieve isolation. A piece of data is protected by a read-write lock that allows shared reads but only one exclusive write at a time. If two parallel transactions don't contend on locks, that means that they won't mess up with each other, that they are truly concurrently, that they don't have causal dependence, all of which ultimately means that we don't have to stress too much about comparing their timestamps because their ordering is nondeterministic anyway. Clients would just accept that either one happens to complete faster than the other.

On the other hand, if two transactions contend on locks, one will acquire all the locks first and enter its transaction logic. The other will have to wait, and will be able to acquire locks when the first one starts releasing locks. Therefore, if we can assign a timestamp after the acquisition of all locks and before the release of any lock, the timestamp will reflect the execution order and their causal dependency.

The TrueTime API

Callers call `TT.now()`. The TrueTime API returns a `[earliest, latest]` interval. The absolute time falls in that interval. The essence of that is instead of telling us what time it is now, it tells us that the current time is between 10:30 and 10:35. That's extremely

useful, as we shall see later, because it bounds and exposes the clock uncertainty which the caller can in turn take advantage of. My “between 10:30 and 10:35” is just an illustrative exaggeration. In reality, the returned `earliest` and `latest` are only a few milliseconds apart. The interval is very narrow. Credit to the GPS and atomic clocks deployed in every Google data center. You may wonder what if the absolute time falls out of that interval due to errors. Google’s experiments showed that it’s six times less likely than a CPU failure. So in practice we don’t worry about it.

How to Assign Commit Timestamps

Spanner’s data is partitioned by table key for scalability. Each partition is then replicated for redundancy and performance. Within each partition, a leader is elected among the replicas. All writes go through the leader. Spanner uses multi-paxos to implement a time-based leader lease, and thus a partition (the leader + other replicas) is called a paxos group. The group will elect a new leader when the leader lease expires. The leader can extend its lease by sending “heartbeats”. A transaction only interacts with one partition if all its reads and writes concern data in that single partition.

Interjection: it’s OK if you don’t understand Paxos. It won’t affect your understanding of this blog post. In fact, interestingly, research has shown that learning Paxos prematurely actually hurt your chance of comprehending distributed consensus . I’ve written a blog post about Raft [https://levelup.gitconnected.com/raft-consensus-protocol-made-simpler-922c38675181], which is better suited for new learners who want to grasp distributed consensus and log replication.

When a transaction touches multiple partitions, a distributed transaction takes place. The spanner client library selects a coordinator out of the participating partitions, and starts a two-phase commit. Eventually, once the coordinator collects all the necessary information, it will assign the timestamp and inform the participants to commit the data.

In the single partition case, the partition leader needs to make sure the commit timestamp is larger than all of the previous commit timestamps it has assigned. This can be achieved through a commit wait. See figure-1 for an illustration.

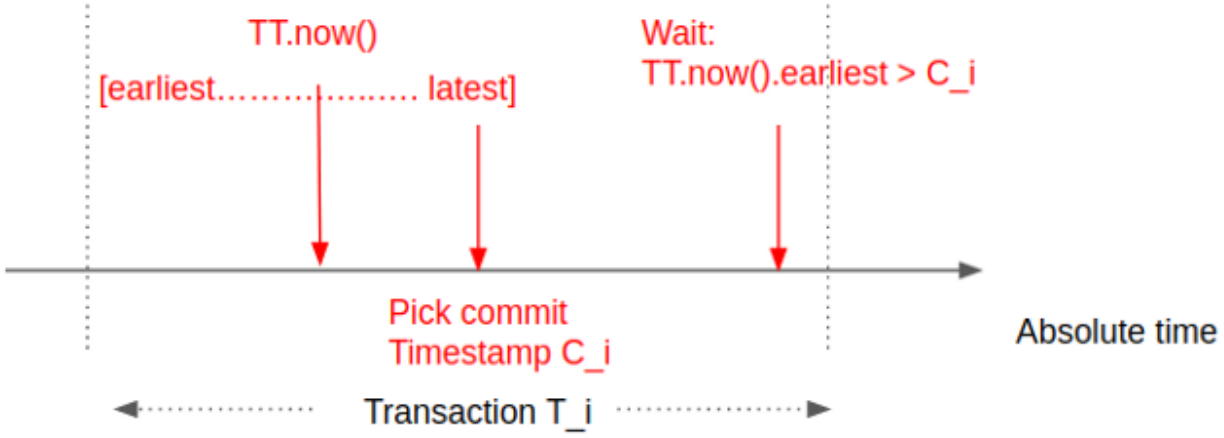
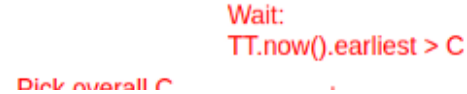


Figure 1

For a transaction `T_i`, we call `TT.now()` after acquisition of all locks, and pick the timestamp `C_i` to be the returned `latest`. Then we spin until the next `TT.now().earliest` is larger than `C_i`. This guarantees that the commit timestamp falls in the exclusive interval of transaction `T_i`. A conflicting transaction `T_(i+1)` that happens after `T_i` commits will therefore have a commit timestamp `C_(i+1)` larger than `C_i`. It may seem that the wait is causing a noticeable delay on the transaction commit. But in practice, Spanner does other work in parallel to that wait, including replicating data across continents, by the end of which `TT.now().earliest` is long after `C_i`.

In the multiple partitions case, the coordinator needs to make sure the commit timestamp is larger than not only its own previous commit timestamps but also the commit timestamps other participants have previously assigned. The coordinator collects that information during the process of the two-phase commit. If you know the two-phase commit, you'll know that there is a PREPARE phase in which the coordinator will receive responses from participants. Participants can send their previous commit timestamps in those responses. In the end, the coordinator will instruct other participants to commit, in which it can also communicate the chosen commit timestamp. But that detail is not important here. The takeaway is that there are pre-existing communication channels between the coordinate and other participants through which they can piggyback the commit timestamp information. See figure-2 for an illustration.



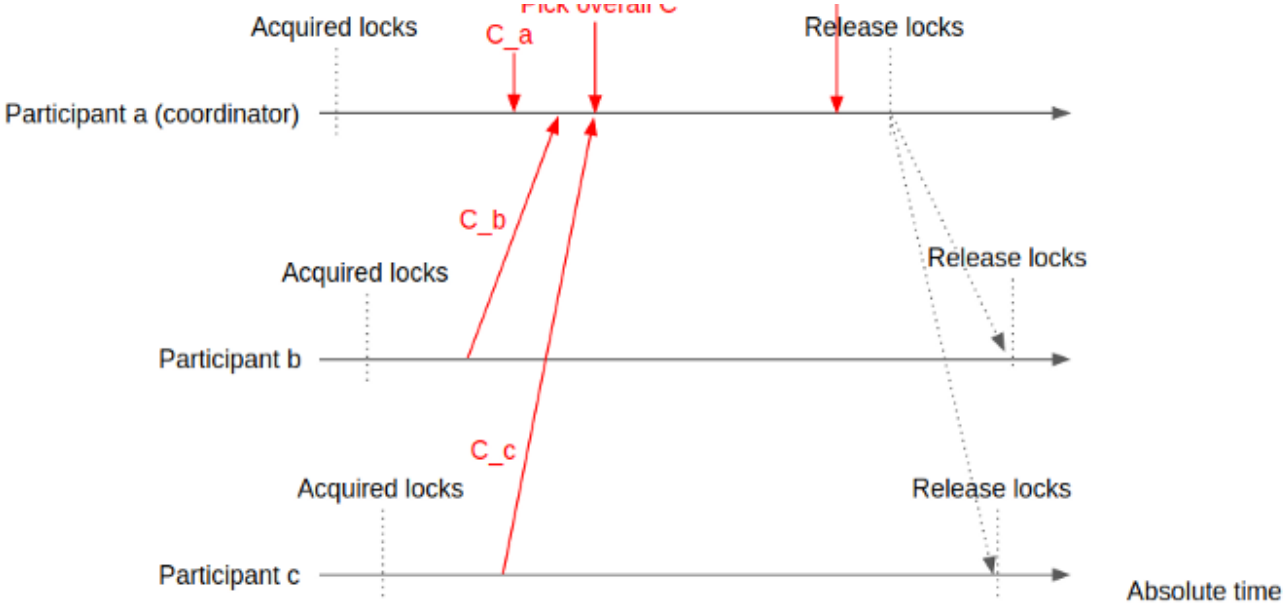


Figure-2

The coordinator acquires locks and picks its own commit timestamp `C_a`. It then waits for other participants to acquire their locks and send it their picks of commit timestamps (`C_b`, and `C_c`). The coordinator picks an overall commit timestamp `C` that's the max of all. The coordinator then does the commit wait the same way as in the single partition case. When it's safe to release locks, the coordinator does so and informs other participants. A new transaction may start in the coordinator right after it releases locks while other participants are still in locking. But that's safe. Because if the new transition is a single partition transaction confined within the coordinator, it doesn't concern data in other participants. But if it's a distributed transaction, its overall commit timestamp `C` will have to wait for the previous transaction to release locks in other participants first since other participants only send their picks of the new commit timestamps after new locks acquisition.

Closing

I hope this blog post has provided some insights to how Spanner guarantees strong consistency. Next time, when people ask you that, you will be able to say: "it's because Spanner waits out the clock uncertainty — which it obtains through the TrueTime API — and make sure the assigned commit timestamps fall in the exclusive interval of the transactions." That'll be all. See you next time.

Distributed Systems Cloud Computing Cloud Spanner Tech Software Engineering

About Help Legal