

Metadata

Thursday, July 4, 2013

Spanner: Google's Globally-Distributed Database

The [Spanner paper](#) by Google (appeared in OSDI'12) is cryptic and hard to understand. When I first read it, I thought I understood the main idea, and that the benefit of TrueTime was to enable lock-free read-only transactions in Spanner. Then, I slowly realized things didn't check; it was possible to achieve lock-free read-only transactions without TrueTime as well. I did another read, and thought for some time, and had a better understanding of how TrueTime benefits Spanner, and how to improve its shortcomings.

I will first provide a summary of the Spanner work (borrowing sentences and figures from the Spanner paper), and then talk about what TrueTime is actually good for.

In a nutshell

Spanner is Google's scalable, multi-version, globally-distributed, and synchronously-replicated database. Spanner supports non-blocking reads in the past, lock-free read-only transactions, and atomic schema changes. In order to support *externally-consistent* distributed transactions at global scale, it uses a novel **TrueTime** API that exposes clock uncertainty.

From NoSQL to NewSQL!

The need to support semi-relational tables and synchronous replication in Spanner has been motivated by the popularity of Megastore. Many applications at Google (e.g., Gmail, Picasa, Calendar, Android Market, and AppEngine) chose to use Megastore because of its semi-relational data model and synchronous replication, despite its poor write throughput.

Spanner evolved from a Bigtable-like versioned key-value store into a temporal multi-version database. Data is stored in semi-relational tables, and Spanner provides a SQL-based query language and supports general-purpose long-lived transactions (e.g. for report generation —on the order of minutes). The Spanner team believes it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.

Spanner distributed database

Data is versioned, and each version is automatically timestamped with its commit time by the TrueTime API. Spanner provides externally consistent reads and writes, and globally-consistent reads across the database at a timestamp. External consistency (or equivalently, linearizability) is defined as follows: if a transaction T1 commits before another transaction T2 starts, then T1's commit timestamp is smaller than T2's. Using TrueTime Spanner is able to assign globally-meaningful commit timestamps to transactions, which reflect the serialization order.

2 TrueTime API

| Method | Returns |
|---------------------|---|
| <i>TT.now()</i> | <i>TTinterval</i> : [earliest, latest] |
| <i>TT.after(t)</i> | true if <i>t</i> has definitely passed |
| <i>TT.before(t)</i> | true if <i>t</i> has definitely not arrived |

Table 1: TrueTime API. The argument *t* is of type *TTstamp*.

About Me



Murat

I am a computer science and engineering professor at SUNY Buffalo. I work on distributed and networked systems and fault-tolerance. [Here is my webpage](#). You can also follow me on [Twitter](#).

[View my complete profile](#)

Blog Archive

- ▼ 2013 (22)
 - ▶ August (2)
 - ▼ July (4)
 - [How I read a research paper](#)
 - [Apps are selfish parasites! How can we get truly C...](#)
 - [Ramblings on serializability](#)
 - [Spanner: Google's Globally-Distributed Database](#)
 - ▶ June (2)
 - ▶ May (5)
 - ▶ April (8)
 - ▶ January (1)
- ▶ 2012 (18)
- ▶ 2011 (38)
- ▶ 2010 (31)
- ▶ 2007 (1)

Popular Posts

[Ceph: A Scalable, High-Performance Distributed File System](#)

[Spanner: Google's Globally-Distributed Database](#)

[Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM](#)

[MDCC: Multi-Data-Center Consistency](#)

[Megastore: Providing scalable, highly available storage for interactive services](#)

[Finding a Needle in Haystack: Facebook's Photo Storage](#)

[Fault Tolerance via Idempotence \(paper summary\)](#)

[SEDA: An architecture for well-conditioned scalable internet services](#)

[The role of distributed state](#)

Pageviews



122,792

Subscribe To

Posts

Comments

Search This Blog

Follow by Email

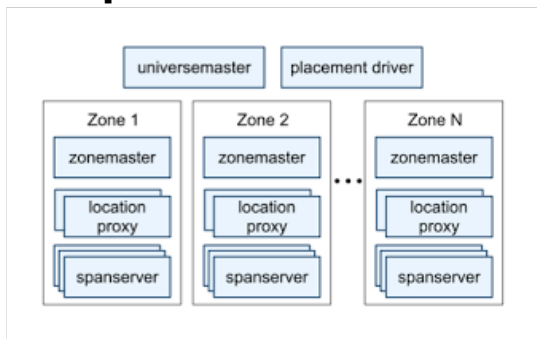
TT.now() was invoked. In other words, TrueTime guarantees that for an invocation $tt = TT.now()$, $tt.earliest < t_{abs}(e_{now}) < tt.latest$, where e_{now} is the invocation event and $t_{abs}(e_{now})$ is the absolute time of event e_{now} . The instantaneous error bound is denoted as ϵ , which is half of the interval's width.

Google keeps uncertainty small (bounded by around 6ms) by using multiple modern clock references (GPS and atomic clocks). TrueTime is implemented by a set of time master machines per datacenter and a time slave daemon per machine. The majority of masters have GPS receivers with dedicated antennas. The remaining masters (Armageddon masters) are equipped with atomic clocks.

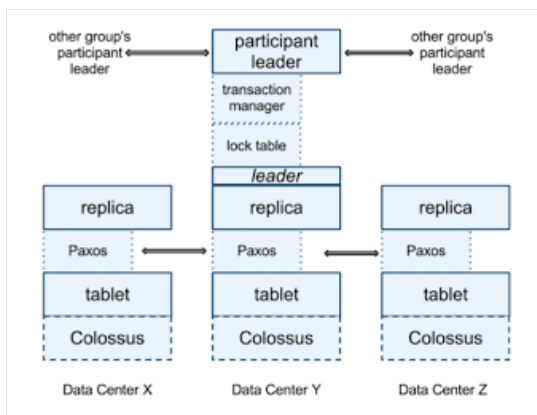
Every daemon polls a variety of masters to reduce vulnerability to errors from any one master. Between synchronizations, a daemon advertises a slowly increasing time uncertainty. ϵ is derived from conservatively applied worst-case local clock drift. The daemon's poll interval is currently 30 seconds, and the current applied drift rate is set at $200 \mu \text{ sec/second}$, which together account for the bound on uncertainty at 6ms.

The TrueTime API directly exposes clock uncertainty, and the guarantees on Spanner's timestamps depend on the bounds that the implementation provides. If the uncertainty is large, Spanner slows down to wait out that uncertainty.

3 Spanner Implementation



A zone has 1 zonemaster and 100 to 1000s of spanservers. Zonemaster assigns data to spanservers; spanserver serves data to clients. Location proxies help clients to locate the spanservers assigned to serve their data. The universe master displays status information about all the zones for interactive debugging. The placement driver handles automated movement of data across zones on the timescale of minutes.



Each spanserver is responsible for 100 to 1000 tablets. A tablet implements a bag of the following mappings: $(\text{key}:\text{string}, \text{timestamp}:\text{int64}) \rightarrow \text{string}$. To support replication, each spanserver implements a single Paxos state machine on top of each tablet.

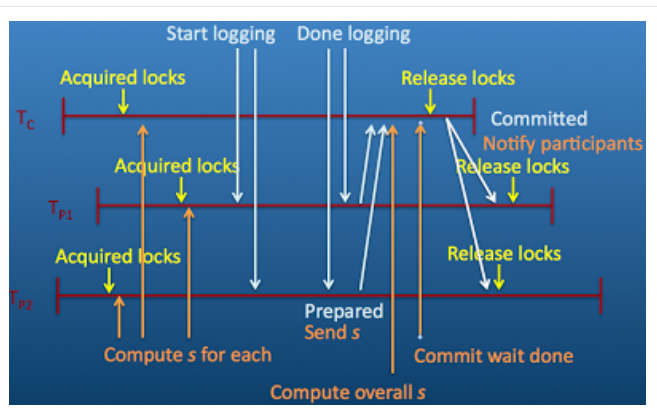
At every replica that is a leader, each spanserver implements: a lock table (mapping ranges of keys to lock states) to implement concurrency control, and a transaction manager to support distributed transactions. If a transaction involves only one Paxos group (as is the case for most transactions), it can bypass the transaction manager, since

If a transaction involves more than one Paxos group, those groups' leaders coordinate to perform 2-phase commit. One of the participant groups is chosen as the coordinator: the participant leader of that group will be referred to as the coordinator leader, and the slaves of that group as coordinator slaves.

4 Concurrency control

The Spanner implementation supports read-write transactions, read-only transactions, and snapshot reads. Standalone writes are implemented as read-write transactions; non-snapshot standalone reads are implemented as read-only transactions. A snapshot read is a read in the past that executes without locking.

4.1 Read-Write transactions



Read-write transactions use 2-phase locking and 2-phase commit. First, the client issues reads to the leader replica of the appropriate group, which acquires read locks and then reads the most recent data. When a client has completed all reads and buffered all writes, it starts 2-phase commit. Read-write transactions can be assigned commit timestamps by the coordinator leader at any time when all locks have been acquired, but before any locks have been released. For a given transaction, Spanner assigns it the timestamp that the coordinating leader assigns to the Paxos write that represents the transaction commit. To wait out the uncertainty in TrueTime, there is a *Commit Wait*: The coordinator leader ensures that clients cannot see any data committed by T_i until $TT.after(s_i)$ is true.

4.2 Read-only transactions

Reads in a read-only transaction execute at a system-chosen timestamp without locking, so that incoming writes are not blocked. A read-only transaction executes in 2 phases: assign a timestamp s_{read} , and then execute the transaction's reads as snapshot reads at s_{read} . The snapshot reads can execute at any replicas that are sufficiently up-to-date.

Serving reads at a timestamp. Every replica tracks a value called safe time t_{safe} which is the maximum timestamp at which a replica is up-to-date. A replica can satisfy a read at a timestamp t , if $t \leq t_{safe}$. We define $t_{safe} = \min(t^{Paxos}, t^{TM})$, where each Paxos state machine has a safe time t^{Paxos} and each transaction manager has a safe time t^{TM} .

t^{Paxos} is the timestamp of the highest-applied Paxos write. Because timestamps increase monotonically and writes are applied in order, writes will no longer occur at or below t^{Paxos} with respect to Paxos.

t^{TM} is ∞ at a replica if there are zero prepared (but safe not committed) transactions—that is, transactions in between the two phases of 2-phase commit. Otherwise, for every participant group g , over all transactions T_i prepared at g , $t^{TM} = \min_i (s^{prepare}_{i,g}) - 1$. In other words, t^{TM} denotes the request timestamp of the earliest prepared but not committed transaction.

timestamp may require the execution of the data reads at s_{read} to block if t_{safe} has not advanced sufficiently. To reduce the chances of blocking, Spanner should assign the oldest timestamp that preserves external consistency. (External consistency constraint dictates that you cannot use an older version of variable to read, and you cannot assign a timestamp earlier than pending read-write transaction on any of the variables involved in the read-only transaction). Spanner implements a simpler choice when multiple Paxos groups are involved. The client avoids a negotiation round, and just has its reads execute at $s_{read} = TT.now().latest$, which may wait for safe time to advance.

4.3 Refinements

t^{TM} as defined above has a weakness, in that a single prepared transaction prevents t_{safe} from advancing. Such false conflicts can be removed by augmenting t^{TM} with a fine-grained mapping from key ranges to prepared-transaction timestamps. When a read arrives, it only needs to be checked against the fine-grained safe time for key ranges with which the read conflicts.

t^{Paxos} is also advanced by heartbeats to help t_{safe} advance at the replicas. (This does not require high precision clock synchronization, and NTP easily suffices for this.)

5 TrueTime, what is it good for?

The paper does not directly discuss what TrueTime buys for Spanner. It says this in the conclusions: "One aspect of our design stands out: the linchpin of Spanner's feature set is TrueTime. We have shown that reifying clock uncertainty in the time API makes it possible to build distributed systems with much stronger time semantics." What does this mean exactly? What TrueTime buys Spanner is left unclear in the paper.

After re-reading the paper only with this question in mind, I was left more puzzled. In my first read of the paper, I thought TrueTime enabled lock-free reads in Spanner. After the second reading, I realized that lock-free reads could be implemented without TrueTime, only by using version numbers, because read-only transactions were also serialized by coordinating leaders and Paxos groups. TrueTime wasn't speeding up read-only transactions either. Even using TrueTime, read-only transaction still needs to wait to hear/learn the commit information from any variable/tablet-overlapping pending/prepared read-write transaction.

Maybe TrueTime benefited by making Spanner implementation simple, but Section 4.2.4 lists several unavoidable implementation hardships even with TrueTime. It looks like using version numbers wouldn't be significantly more complicated. Also, for schema change and paxos leader replacement (which the paper claims TrueTime simplified a lot), the NTP synchronization (several tens of ms accuracy) easily suffices. We could have easily avoided the more precise TrueTime implementation with GPS and atomic clocks for these.

TrueTime and consistent-cuts in the past

After I was almost convinced that TrueTime was not good for anything, I realized this:

TrueTime benefits snapshot reads (reads in the past) the most! By just giving a time in the past, the snapshot read can get a consistent cut read of all the variables requested at that given time. This is not an easy feat to accomplish in a distributed system without using TrueTime and high-precision synchronized clocks, as it would require capturing and recording causality relationships across many different versions of the variables involved so that a consistent cut can be identified for all the variables requested in the snapshot read. That would certainly be highly prohibitive to store in the multiversion database and very hard to query as well. *TrueTime provides a convenient and succinct way of encoding and accessing past consistent-cuts of the Spanner multiversion database.*


But can we find any clever solutions to circumvent that problem without resorting to the high-precision clocks in TrueTime?

My close friend and frequent-collaborator Sandeep Kulkarni at Michigan State University had proposed [HybridClocks](#) in 2001. HybridClocks also exposed clock uncertainty ϵ in

Using HybridClocks it can be possible to relieve the atomic clock requirement in Spanner and use NTP instead. Even with ϵ at 100 msec, we can still track finer-grain dependencies between variables using HybridClocks. The good news is that the size of the HybridClocks can be kept very succinct and can be recorded in Spanner database as timestamp to enable snapshot reads easily.

HybridClocks idea may also help speed up Spanner's high-precision clock-based implementation. In Spanner ϵ determines the rate of read-write transactions on a tablet. This is because, the coordinating leaders delay read-write transactions to commit with at least ϵ time apart in order to ensure that there is no uncertainty in past snapshot reads. It is possible to avoid this wasteful waiting by adding some logical clocks information to TrueTime as prescribed in HybridClocks.

Sandeep and I are currently exploring these ideas.

Posted by [Murat](#) at 2:59 AM 

+27 [Recommend this on Google](#)

1 comment:

 [Evan Jones](#) said...

I wondered the same thing when I read the Spanner paper. My conclusion was that it simplifies externally consistent reads (which is related, now that I think about it). Consider Process 1 commits a transaction, sends a message to Process 2, which then reads the results of that transaction. The Spanner implementation guarantees that Process 2 will see the results of Process 1's write, without needing to do anything like communicate version vectors or other "state" like some protocols require.

However, its been a while since I read the paper, so this could be inaccurate. :) Thanks for the detailed summary.

[July 18, 2013 at 8:46 PM](#)

[Post a Comment](#)

Links to this post

[Create a Link](#)

[Newer Post](#)

[Home](#)

[Older Post](#)

Subscribe to: [Post Comments \(Atom\)](#)