```
6.824 2013 Lecture 7: Spanner

Spanner: Google's Globally-Distributed Datase
Corbett et al, OSDI 2012

why this paper?
  modern, high performance, driven by real-world needs
  sophisticated use of paxos
  tackles consistency + performance (will be a big theme)
  Lab 4 a (hugely) simplified version of Spanner

what are the big ideas?
  shard management w/ paxos replication
  high performance despite synchronous WAN replication
  fast reads by asking only the nearest replica
  consistency despite sharding (this is the real focus)
  clever use of time for consistency
  distributed transactions

this is a dense paper!
  i've tried to boil down some of the ideas to simpler form

idea: sharding
  we've seen this before in FDS
  the real problem is managing configuration changes
  Spanner has a more convincing design for this than FDS

simplified sharding outline (lab 4):
  replica groups, paxos-replicated
    paxos log in each replica group
  master, paxos-replicated
    assigns shards to groups
    numbered configurations
  if master moves a shard, groups eventually see new config
  "start handoff Num=7" op in both groups' paxos logs
    though perhaps not at the same time
  dst can't finish handoff until it has copies of shard data at majority
    and can't wait long for possibly-dead minority
    minority must catch up, so perhaps put shard data in paxos log (!)
  "end handoff Num=7" op in both groups' logs

Q: what if a Put is concurrent w/ handoff?
   client sees new config, sends Put to new group before handoff starts?
   client has stale view and sends it to old group after handoff?
   arrives at either during handoff?

Q: what if a failure during handoff?
   e.g. old group thinks shard is handed off
        but new group fails before it thinks so

Q: can *two* groups think they are serving a shard?

Q: could old group still serve shard if can't hear master?

idea: wide-area synchronous replication
  goal: survive single-site disasters
  goal: replica near customers
  goal: don't lose any updates

considered impractical until a few years ago
```

```
  paxos too expensive, so maybe primary/backup?
  if primary waits for ACK from backup
    50ms network will limit throughput and cause palpable delay
    esp if app has to do multiple reads at 50ms each
  if primary does not wait, it will reply to client before durable
  danger of split brain; can't make network reliable

what's changed?
  other site may be only 5 ms away -- San Francisco / Los Angeles
  faster/cheaper WAN
  apps written to tolerate delays
    may make many slow read requests
    but issue them in parallel
    maybe time out quickly and try elsewhere, or redundant gets
  huge # of concurrent clients lets you get hi thruput despite high delay
    run their requests in parallel
  people appreciate paxos more and have streamlined variants
    fewer msgs
      page 9 of paxos paper: 1 round per op w/ leader + bulk preprepare
      paper's scheme a little more involved b/c they must ensure
        there's at most one leader
    read at any replica

actual performance?
  Table 3
    pretend just measuring paxos for writes, read at any replica for reads
    latency
      why doesn't write latency go up w/ more replicas?
      why does std dev of latency go down w/ more replicas?
      r/o a *lot* faster since not a paxos agreement + use closest replica
    throughput
      why does read throughput go up w/ # replicas?
      why doesn't write throughput go up?
      does write thruput seem to be going down?
    what can we conclude from Table 3?
      is the system fast? slow?
    how fast do your paxoses run?
      mine takes 10 ms per agreement
      with purely local communication and no disk
      Spanner paxos might wait for disk write
  Figure 5
    npaxos=5, all leaders in same zone
    why does killing a non-leader in each group have no effect?
    for killing all the leaders ("leader-hard")
      why flat for a few seconds?
      what causes it to start going up?
      why does it take 5 to 10 seconds to recover?
      why is slope *higher* until it rejoins?

spanner reads from any paxos replica
  read does *not* involve a paxos agreement
  just reads the data directly from replica's k/v DB
  maybe 100x faster -- same room rather than cross-country

Q: could we *write* to just one replica?

Q: is reading from any replica correct?

example of problem:
  photo sharing site
  i have photos
```

```
    i have an ACL (access control list) saying who can see my photos
    i take my mom out of my ACL, then upload new photo
    really it's web front ends doing these client reads/writes
    1. W1: I write ACL on group G1 (bare majority), then
    2. W2: I add image on G2 (bare majority), then
    3. mom reads image -- may get old data from lagging G2 replica
    4. mom reads ACL -- may get new data from G1

  this system is not acting like a single server!
    there was not really any point at which the image was
      present but the ACL hadn't been updated

  this problem is caused by a combination of
    * partitioning -- replica groups operate independently
    * cutting corners for performance -- read from any replica

  how can we fix this?
    A. make reads see latest data
       e.g. full paxos for reads
       expensive!
    B. make reads see *consistent* data
       data as it existed at *some* previous point in time
       i.e. before #1, between #1 and #2, or after #2
       this turns out to be much cheaper
       spanner does this

  here's a super-simplification of spanner's consistency story for r/o clients
    "snapshot" or "lock-free" reads
    assume for now that all the clocks agree
    server (paxos leader) tags each write with the time at which it occurred
    k/v DB stores *multiple* values for each key,
      each with a different time
    reading client picks a time t
      for each read
        ask relevant replica to do the read at time t
    how does a replica read a key at time t?
      return the stored value with highest time <= t
    but wait, the replica may be behind
      that is, there may be a write at time < t, but replica hasn't seen it
      so replica must somehow be sure it has seen all writes <= t
      idea: has it seen *any* operation from time > t?
        if yes, and paxos group always agrees on ops in time order,
          it's enough to check/wait for an op with time > t
        that is what spanner does on reads (4.1.3)
    what time should a reading client pick?
      using current time may force lagging replicas to wait
      so perhaps a little in the past
      client may miss latest updates
      but at least it will see consistent snapshot
      in our example, won't see new image w/o also seeing ACL update

  how does that fix our ACL/image example?
    1. W1: I write ACL, G1 assigns it time=10, then
    2. W2: I add image, G2 assigns it time=15 (> 10 since clocks agree)
    3. mom picks a time, for example t=14
    4. mom reads ACL t=14 from lagging G1 replica
       if it hasn't seen paxos agreements up through t=14, it knows to wait
       so it will return G1
    5. mom reads image from G2 at t=14
       image may have been written on that replica
       but it will know to *not* return it since image's time is 15
```

```
  other choices of t work too

Q: is it reasonable to assume that different computers' clocks agree?
   why might they not agree?

Q: what may go wrong if servers' clocks don't agree?

a performance problem: reading client may pick time in the
  future, forcing reading replicas to wait to "catch up"

a correctness problem:
  again, the ACL/image example
  G1 and G2 disagree about what time it is
  1. W1: I write ACL on G1 -- stamped with time=15
  2. W2: I add image on G2 -- stamped with time=10
  now a client read at t=14 will see image but not ACL update

Q: why doesn't spanner just ensure that the clocks are all correct?
   after all, it has all those master GPS / atomic clocks

TrueTime (section 3)
  there is an actual "absolute" time t_abs
    but server clocks are typically off by some unknown amount
    TrueTime can bound the error
  so now() yields an interval: [earliest,latest]
  earliest and latest are ordinary scalar times
    perhaps microseconds since Jan 1 1970
  t_abs is highly likely to be between earliest and latest

Q: how does TrueTime choose the interval?

Q: why are GPS time receivers able to avoid this problem?
   do they actually avoid it?
   what about the "atomic clocks"?

spanner assigns each write a scalar time
  might not be the actual absolute time
  but is chosen to ensure consistency

the danger:
  W1 at G1, G1's interval is [20,30]
    is any time in that interval OK?
  then W2 at G2, G2's interval is [11,21]
    is any time in that interval OK?
  if they are not careful, might get s1=25 s2=15

so what we want is:
  if W2 starts after W1 finishes, then s2 > s1
  simplified "external consistency invariant" from 4.1.2
  causes snapshot reads to see data consistent w/ true order of W1, W2

how does spanner assign times to writes?
  (again, this is much simplified, see 4.1.2)
  a write request arrives at paxos leader
  s will be the write's time-stamp
  leader sets s to TrueTime now().latest
    this is "Start" in 4.1.2
  then leader *delays* until s < now().earliest
    i.e. until s is guaranteed to be in the past (compared to absolute time)
    this is "commit wait" in 4.1.2
  then leader runs paxos to cause the write to happen
```

```
    then leader replies to client

  does this work for our example?
    W1 at G1, TrueTime says [20,30]
      s1 = 30
      commit wait until TrueTime says [31,41]
      reply to client
    W2 at G2, TrueTime *must* now say >= [21,31]
      (otherwise TrueTime is broken)
      s2 = 31
      commit wait until TrueTime says [32,43]
      reply to client
    it does work for this example:
      the client observed that W1 finished before S2 started,
      and indeed s2 > s1
      even though G2's TrueTime clock was slow by the most it could be
      so if my mom sees S2, she is guaranteed to also see W1

  why the "Start" rule?
    i.e. why choose the time at the end of the TrueTime interval?
    previous writers waited only until their timestamps were barely < t_abs
    new writer must choose s greater than any completed write
    t_abs might be as high as now().latest
    so s = now().latest

  why the "Commit Wait" rule?
    ensures that s < t_abs
    otherwise write might complete with an s in the future
      and would let Start rule give too low an s to a subsequent write

  Q: why commit *wait*; why not immediately write value with chosen time?
    indirectly forces subsequent write to have high enough s
      the system has no other way to communicate minimum acceptable next s
      for writes in different replica groups
    waiting forces writes that some external agent is serializing
      to have monotonically increasing timestamps
    w/o wait, our example goes back to s1=30 s2=21
    you could imagine explicit schemes to communicate last write's TS
      to the next write

  Q: how long is the commit wait?

  this answers today's Question
    a large TrueTime uncertainty requires a long commit wait
    so Spanner authors are interested in accurate low-uncertainty time

  let's step back
    why did we get into all this timestamp stuff?
      our replicas were 100s or 1000s of miles apart (for locality/fault tol)
      we wanted fast reads from a local replica (no full paxos)
      our data was partitioned over many replica groups w/ separate clocks
      we wanted consistency for reads:
        if W1 then W2, reads don't see W2 but not W1
    it's complex but it makes sense as a
      high-performance evolution of Lab 3 / Lab 4

  why is this timestamp technique interesting?
    we want to enforce order -- things that happened in some
      order in real time are ordered the same way by the
      distributed system -- "external consistency"
    the naive approach requires a central agent, or lots of communication
```

```
    Spanner does the synchronization implicitly via time
      time can be a form of communication
      e.g. we agree in advance to meet for dinner at 6:00pm

  there's a lot of additional complexity in the paper
    transactions, two phase commit, two phase locking,
      schema change, query language, &c
    some of this we'll see more of later
    in particular, the problem of ordering events in a
      distributed system will come up a lot, soon
```