```
6.824 2020 Lecture 17: Causal Consistency, COPS

Lloyd et al, Don't Settle for Eventual: Scalable Causal Consistency
for Wide-Area Storage with COPS, SOSP 2011.

the setting: geo-replication for big web sites
  [ 3 datacenters, users, web servers, sharded storage servers ]
  multiple datacenters
  each datacenter has a complete copy of all data
  reads are local -- fast common case
  what about writes?
  what about consistency?

we've seen two solutions for geo-replication
  Spanner
    writes involve Paxos and perhaps two-phase commit
    Paxos quorum for write must wait for some remote sites
    no one site can write on its own
    but has read transactions, consistent, fairly fast
  Facebook / Memcache
    writes must go to the primary site's MySQL
    again, non-primary sites cannot write on their own
    but reads are blindingly fast (1,000,000 per second per memcache server)

can we have a system that allows writes from any datacenter?
  so a write can proceed without talking to / waiting for any other datacenter?
  would help fault tolerance, performance, robustness vs slow WAN
  these local reads and writes are the real goal
    the consistency model is a secondary consideration
    we'll make one up that matches whatever our local read/write design has to do

straw man one
  three data centers
  set of shards in each datacenter
  client reads and writes just contact local shard
  each shard pushes writes to other datacenters, shard-to-shard, asynchronously
  lots of parallelism
  this design favors reads
    could instead have writes be purely local, and reads check other datacenters
    or quorum, with overlap, as in Dynamo/Cassandra

straw man one is an "eventually consistent" design
  1. clients may see updates in different orders
  2. if no writes for long enough, all clients see same data
  a pretty loose spec, many ways to implement, easy to get good performance
  used in deployed systems, e.g. Dynamo and Cassandra
  but can be tricky for app programmers

example app code -- a photo manager:
  C1 uploads photo, adds reference to public list:
    C1: put(photo) put(list)
  C2 reads:
    C2:                       get(list) get(photo)
  C3 also sees new photo, adds to their own list:
    C3: get(list) put(list2)
  C4 sees photo on C3's list:
    C4:                       get(list2) get(photo)

what can C2 see?

what can C4 see?

app code can see non-intuitive behavior -- "anomalies"
  not incorrect, since there was no promise of better
```

```
  and it is possible to program such a system
    perhaps include versions &c in values
    perhaps wait for expected data (photo) to appear
  but we could hope for more intuitive -- easy-to-program -- behavior

an important aside: how to decide which write is most recent?
  in case writes to same key arrive from multiple remote datacenters
  everyone has to choose the same final value, for eventual consistency

why not attach the current wall-clock time as version number on each put?
  local shard server assigns v#=time when it receives client put()
  remote datacenter receives put(k, -, v#)
    if v# is larger than version of currently stored value for k
      replace with new value / v#
    otherwise
      ignore new value
  (note "version" is not quite the right word; "timestamp" would be better.)
  wall-clock time almost works!
  what if two put(k) happen at exactly the same time at different datacenters?
    break tie with a unique ID in the low bits of v#
  what if one datacenter's (or server's) clock is fast by an hour
    will cause that datacenter's values to win
    worse: prevents any other update for an hour!

COPS uses Lamport clocks to assign v#
  each server implements a "Lamport clock" or "logical clock"
    Tmax = highest v# seen (from self and others)
    T = max(Tmax + 1, wall-clock time)
  v# for a new put() is current T
  so: if some server has a fast clock, everyone who sees a version
    from that server will advance their Lamport clock

if concurrent writes, is it OK to simply discard all but one?
  the paper's "last-writer-wins"
  sometimes that's OK:
    e.g. there's only a single possible writer, so the problem can't arise
    probably I'm the only person who can write my photo list or profile
  sometimes latest-write-wins is awkward:
    what if put()s are trying to increment a counter?
    or update a shopping cart to have a new item?
  the problem is "conflicting writes"
  we'd often like to have a more clever plan to detect and merge
    real transactions
    mini-transactions -- atomic increment operation, not just get()/put()
    custom conflict resolution for shopping cart (set union?)
  resolution of conflicting writes is a problem for eventual/causal consistency
    no single "serialization point" to implement atomic operations or transactions
  the paper mostly ignores write conflict resolution
    but it's a problem for real systems

back to eventual consistency and straw man one
  can the storage system provide more intuitive results?
  and still retain local reads, write from any datacenter?

straw man two:
  provide a sync(k, v#) operation
  sync() does not return until:
    every datacenter has at least v# for k
  put(k) yields new v# so client can pass it to sync()
  you could call this "eventual plus barriers"
  note sync() is slow: requires talking to / waiting for all datacenters

straw-man-two clients call sync() to force order in which data appears to readers:
  C1: v# = put(photo), sync(photo, v#), put(list)
  C2:                                          get(list) get(photo)
```

```
    C2 may not see the new list, but if it does, it will see photo too

  straw-man-two applications must carefully order both put()s and get()s
    typically get() in the reverse order from put()s
    sync() guarantees "if you see new list, you'll *then* see new photo"
    sync() also forces freshness: reads will see our data once sync() returns

  straw man two may not be so bad
    it's a straightforward, efficient design
    if you don't need transactions, the semantics are pretty good
      it makes the photo list example work
      though requires some thought to get order and sync()s right
    read performance is excellent
    write performance is OK if you don't write much, or don't mind waiting
      after all, the Facebook / Memcache paper says all writes sent to primary datacenter
      and Spanner writes wait for majority of replica sites

  can we have the semantics of sync(), without the cost?
    can we tell remote data centers the right order, w/o sync-style waiting?

  a possibility: single write log per datacenter
    each datacenter has a single "log server"
    put() appends to the local server's log, but doesn't wait
    no sync()
    log server sends log, in order, to other datacenters
      remote sites apply log in order
      so put(photo), put(list) will appear in that order everywhere
    this is not a full solution, but it can be made to work
    but the log server might be a bottleneck if there are many shards
      so COPS does not follow this approach

  so:
    we want to forward puts asynchronously (no sync() or waiting)
    we want each shard to forward puts independently (no central log server)
    we want enough ordering to make example app code work

  each COPS client maintains a "context" to reflect order of client ops
    client adds an item to context after each get() and put()
    client tells COPS to order each put() after everything in its context
    get(X)->v2
      context: Xv2
    get(Y)->v4
      context: Xv2, Yv4
    put(Z, -)->v3
      client sends Xv2, Yv4 to shard server along with new Z
      context: Xv2, Yv4, Zv3
      (COPS optimizes this to just Zv3)

  COPS calls a relationship like "Zv3 comes after Yv4" a dependency
    Yv4 -> Zv3
    what does a dependency tell COPS to do?
      if C2 sees Zv3, and then asks for Y, it should see at least Yv4
    this notion of dependency is meant to match programmer intuition
      about what it means to get(Y) AND THEN put(Z)
      or put(Z) AND THEN put(Q)

  each COPS shard server
    when it receives a put(Z, -, Yv4) from a client,
      picks a new v# = 3 for Z,
      stores Z, -, v3
      sends Z/-/v3/Yv4 to corresponding shard server in each datacenter
        but does not wait for reply
    remote shard server receives Z/-/v3/Yv4
      talks to local shard server for Y
        waits for Yv4 to arrive
```

```
    then updates DB to hold Z/-/v3

the point:
  if a reader at a remote data center sees Zv3,
    and then reads Y,
    it will see a version no later than Yv4

this makes the photo list example work
  put(list) will arrive with photo as a dependency
  remote servers will wait for photo to appear before installing updated list

these consistency semantics are called "causal consistency"
  [ Figure 2 ]
  a client establishes dependencies between versions in two ways:
    1. its own sequence of puts and gets ("Execution Thread" in Section 3)
    2. reading data written by another client
  dependencies are transitive (if A -> B, and B -> C, then A -> C)
  the system guarantees that if A -> B, and a client reads B,
    then the client must subsequently see A (or a later version)

nice: when updates are causally related, readers see updates
  in the order in which the writer saw them

nice: when updates are NOT causally related, COPS has no order obligations
  example:
    C1: put(X)                  put(Z)
    C2:           put(Y)
  X must appear before Z (and this requires effort from COPS)
  Y does NOT have to appear before Z
  this freedom is the basis of the paper's claim of scalability

note: readers may see newer data then required by causal dependencies
  so we're not getting transactions or snapshots

note: COPS sees only certain causal relationships
  ones that COPS can observe from client get()s and put()s
  if there are other communication channels, it is only eventually consistent
  e.g. I put(k), tell you by phone, you do get(k), maybe you won't see my data
  COPS isn't externally consistent

optimizations avoid ever-growing client contexts
  * put(K)->vN sends context, then clears context, replaces with KvN
    so next put(), e.g. put(L), depends only on KvN
    so remote sites will wait for arrival of KvN before writing L
    KvN itself was waiting for (cleared) context
    so L effectively also waits for (cleared) context
  * garbage collection sees when all datacenters have a certain version
    that version never needs to be remembered in context
    since it's already visible to everyone

are there situations where ordered puts/gets aren't sufficient?
  paper's example: a photo list with an ACL (Access Control List)
  get(ACL), then get(list)?
    what if someone deletes you from ACL, then adds a photo?
      between the two get()s
  get(list), then get(ACL)?
    what if someone deletes photo, then adds you to ACL?
  need a multi-key get that returns mutually consistent versions

COPS-GT get_trans() approach
  servers store full set of dependencies for each value
  servers store a few old versions
  get_trans(k1,k2,...)
    client library does independent get()s
    get()s return dependencies as well as value/v#
```

```
  client checks dependencies
    for each get() result R,
      for each other get result S mentioned in R's dependencies,
        is Sv#  >= version mentioned in R's dependency?
  if yes for all, can use results
  if no for any, need a second round of get()s for values that were too old
    each fetches the version mentioned in dependencies
    may be old: to avoid cascading dependencies

for ACL / list example:
  C1: get_trans(ACL, list)
  C1: get(ACL) -> v1, no deps
      C2: put(ACL, v2)
      C2: put(list, v2, deps=ACL/v2)
  C1: get(list) -> v2, deps: ACL/v2
  (C1 checks dependencies against value versions)
  C1: get(ACL) -> v2
  (now C1 has a causally consistent pair of get() results)

why are only two phases needed for COPS-GET get transactions?
  a new value won't be installed until all its dependencies are installed
  so if a get() returns a dependency, it must already be locally installed

performance?
  around 50,000 ops/second
  that is OK, like a conventional DB
  no comparisons with other systems
    not for performance
    not for ease of programming
  too bad, since central thesis is that COPS has a better tradeoff between
    ease of programming and performance

limitations / drawbacks, for both COPS and causal consistency
  conflicting writes are a serious difficulty
  awkward for clients to track causality
    e.g. user and browser, multiple page views, multiple servers
  COPS doesn't see external causal dependencies
    s/w and people really do communicate outside of the COPS world
  limited notion of "transaction"
    only for reads (though later work generalized a bit)
    definition is more subtle than serializable transactions
  significant overhead to track, communicate, obey causal dependencies
    remote servers must check and delay updates
    update delays may cascade

impact?
  causal consistency is a popular research idea
    with good reason: promises both performance and useful consistency
    much work before COPS -- and after (Eiger, SNOW, Occult)
  causal consistency is rarely used in deployed storage systems
  what is actually used?
    no geographic replication at all, just local
    primary-site (PNUTS, Facebook/Memcache)
    eventual consistency (Dynamo, Cassandra)
    strongly consistent (Spanner)
```