```
6.824 2015 Lecture 21: Distributed Optimistic Concurency Control

Paper: Efficient Optimistic Concurrency Control using Loosely
Synchronized Clocks, by Adya, Gruber, Liskov and Maheshwari.

Why this paper?
  to look at optimistic concurrency control (OCC)
  OCC might help us get large scale, high speed, *and* good semantics

Thor overview
  [clients, client caches, servers A-M N-Z]
  data sharded over servers
  code runs in clients (not like Argus; not an RPC system)
  clients read/write DB records from servers
  clients cache data locally for fast access
    on client cache miss, fetch from server

Thor arrangement is fairly close to modern big web site habits
  clients, local fast cache, slower DB servers
  like Facebook/memcache paper
  but Thor has much better semantics

Thor programs use fully general transactions
  multi-operation
  serializable
  so can do bank xfers w/o losing money, &c

Client caching makes transactions tricky
  writes have to invalidatate cached copies
  how to cope with reads of stale cached data?
  how to cope with read-modify-write races?
  clients could lock before using each record
    but that's slow -- probably need to contact server
    wrecks the whole point of fast local caching in clients
    (though caching read locks might be OK, as in paper Eval)

Thor uses optimistic concurrency control (OCC)
  an idea from the early 1980s
  just read and write the local copy
    don't worry about other transactions until commit
  when transaction wants to commit:
    send read/write info to server for "validation"
    validation decides if OK to commit -- if serializable
    if yes, send invalidates to clients with cached copies of written records
    if no, abort, discard writes
  optimistic b/c hopes for no conflict
    if turns out to be true, fast!
    if false, validation can detect, but slow

What should validation do?
  it looks at what the executing transactions read and wrote
  decides if there's a serial execution order that would have gotten
    the same results as the actual concurrent execution
  there are many OCC validation algorithms!
    i will outline a few, leading up to Thor's

Validation scheme #1
  a single validation server
  clients tell validation server the read and write VALUES
    seen by each transaction that wants to commit
```

```
    "read set" and "write set"
  validation must decide:
    would the results be serializable if we let these
      transactions commit?
  scheme #1 shuffles the transactions, looking for a serial order
    in which each read sees the value written by the most
    recent write; if one exists, the execution was serializable.

Validation example 1:
  initially, x=0 y=0 z=0
  T1: Rx0 Wx1
  T2: Rz0 Wz9
  T3: Ry1 Rx1
  T4: Rx0 Wy1
  validation needs to decide if this execution (reads, writes)
    is equivalent to some serial order
  yes: one such order is T4, T1, T3, T2; says yes to all
    (really T2 can go anywhere)
  note this scheme is far more permissive than Thor's
    e.g. it allows transactions to see uncommitted writes

OCC is neat b/c transactions didn't need to lock!
  so they can run quickly from client caches
  just one msg exchange w/ validator per transaction
    not one locking exchange per record used
  OCC excellent for T2 which didn't conflict with anything
    we got lucky for T1 T3 T4, which do conflict

Validation example 2 -- sometimes must abort:
  initially, x=0 y=0
  T1: Rx0 Wx1
  T2: Rx0 Wy1
  T3: Ry0 Rx1
  values not consistent w/ any serial order!
    T1 -> T3 (via x)
    T3 -> T2 (via y)
    T2 -> T1 (via x)
    there's a cycle, so not the same as any serial execution
  perhaps T3 read a stale y=0 from cache
    or T2 read a style x=0 from cache
  in this case validation can abort one of them
    then others are OK to commit
  e.g. abort T2
    then T1, T3 is OK (but not T3, T1)

How should client handle abort?
  roll back the program (including writes to program variables)
  re-run from start of transaction
  hopefully won't be conflicts the second time
  OCC is best when conflicts are uncommon!

Do we need to validate read-only transactions?
  example:
    initially x=0 y=0
    T1: Wx1
    T2: Rx1 Wy2
    T3: Ry2 Rx0
  i.e. T3 read a stale x=0 from its cache, hadn't yet seen invalidate.
  need to validate in order to abort T3.
  other OCC schemes can avoid validating read-only transactions
    keep multiple versions -- but Thor and my schemes don't
```

```
Is OCC better than locking?
  yes, if few conflicts
    avoids lock msgs, clients don't have to wait for locks
  no, if many conflicts
    OCC aborts, must re-start, perhaps many times
    locking waits
  example: simultaneous increment
    locking:
      T1: Rx0 Wx1
      T2: -------Rx1  Wx2
    OCC:
      T1: Rx0 Wx1
      T2: Rx0 Wx1
      fast but wrong; must abort one

We really want *distributed* OCC validation
  split storage and validation load over servers
  each storage server sees only xactions that use its data
  each storage server validates just its part of the xaction
  two-phase commit (2PC) to check that they all say "yes"
    only really commit if all relevant servers say "yes"

Can we just distribute validation scheme #1?
  imagine server S1 knows about x, server S2 knows about y
  example 2 again
    T1: Rx0 Wx1
    T2: Rx0 Wy1
    T3: Ry0 Rx1
  S1 validates just x information:
    T1: Rx0 Wx1
    T2: Rx0
    T3: Rx1
    answer is "yes" (T2 T1 T3)
  S2 validates just y information:
    T2: Wy1
    T3: Ry0
    answer is "yes" (T3 T2)
  but we know the real answer is "no"

So simple distributed validation does not work
  the validators must choose consistent orders!

Validation scheme #2
  Idea: client (or TC) chooses timestamp for committing xaction
    from loosely synchronized clocks, as in Thor
  validation checks that reads and writes are consistent with TS order
  solves distrib validation problem:
    timestamps tell the validators the order to check
    so "yes" votes will refer to the same order

Example 2 again, with timestamps:
  T1@100: Rx0 Wx1
  T2@110: Rx0 Wy1
  T3@105: Ry0 Rx1
  S1 validates just x information:
    T1@100: Rx0 Wx1
    T2@110: Rx0
    T3@105: Rx1
    timestamps say order must be T1, T3, T2
    does not validate! T2 should have seen x=1
```

```
  S2 validates just y information:
    T2@110: Wy1
    T3@105: Ry0
    timstamps say order must be T3, T2
    validates!
  S1 says no, S2 says yes, two-phase commit coordinator will abort

What have we given up by requiring timestamp order?
  example:
    T1@100: Rx0 Wx1
    T2@50: Rx1 Wx2
  T2 follows T1 in real time, and sees T1's write
  but T2 will abort, since TS says T2 comes first, so T1 should have seen x=2
    could have committed, since T1 then T2 works
  this will happen if client clocks are too far off
    if T1's client clock is ahead, or T2's behind
  so: requiring TS order can abort unnecessarily
    b/c validation no longer *searching* for an order that works
    instead merely *checking* that TS order consistent w/ reads, writes
    we've given up some optimism by requiring TS order
  maybe not a problem if clocks closely synched
  maybe not a problem if conflicts are rare

Problem with schemes so far:
  commit messages contained *values*, which can be big
  could instead use version numbers to check whether
    later xaction read earlier xaction's write
  let's use writing xaction's TS as record version number

Validation scheme #4
  tag each DB record (and cached record) with TS of xation that last wrote it
  validation requests carry TS of each record read

Our example for scheme #4:
  all values start with timestamp 0
  T1@100: Rx@0 Wx
  T2@110: Rx@0 Wy
  T3@105: Ry@0 Rx@100
  note:
    reads have timestamp that was in read record, not value
    writes don't include either value or timestamp
  S1 validates just x information:
    orders the transactions by timestamp:
    T1@100: Rx@0 Wx
    T3@105: Rx@100
    T2@110: Rx@0
    the question: does each read see the most recent write?
      T3 is ok, but T2 is not
  S2 validates just y information:
    again, sort by TS, check each read saw latest write:
    T3@105: Ry@0
    T2@110: Wy
    this does validate
  so scheme #4 abort, correctly, reasoning only about version TSs

what have we give up by thinking about version #s rather than values?
  maybe version numbers are different but values are the same
  e.g.
    T1@100: Wx1
    T2@110: Wx2
    T3@120: Wx1
```

```
    T4@130: Rx1@100
  timestamps say we should abort T4 b/c read a stale version
    should have read T3's write
    so scheme #4 will abort
  but T4 read the correct value -- x=1
    so abort wasn't necessary

Problem: per-record timestamp might use too much storage space
  Thor wants to avoid space overhead
  maybe important, maybe not

Validation scheme #5
  Thor's invalidation scheme: no timestamps on records
  how can validation detect that a transaction read stale data?
  it read stale data b/c earlier xaction's invalidation hadn't yet arrived!
  so server can track invalidation msgs that might not have arrived yet
    "invalid set" -- one per client
    delete invalid set entry when client ACKs invalidation msg
    server aborts committing xaction if it read record in client's invalid set
    client aborts running xaction if it read record mentioned in invalidation

Example use of invalid set
  [timeline]
  Client C1:
    T2@105 ... Rx ... 2PC commit point
    imagine that client acts as 2PC coordinator
  Server:
    VQ: T1@100 Wx
    T1 committed, x in C1's invalid set
      server has sent invalidation message to C1

Three cases:
  1. invalidation arrives before T2 reads
     Rx will miss in client cache, read from data from server
     client will (probably) return ACK before T2 commits
     server won't abort T2
  2. invalidation arrives after T2 reads, before commit point
     client will abort T2 in response to invalidation
  3. invalidation arrives after 2PC commit point
     i.e. after all servers replied to prepare
     this means the client was still in the invalid set when
       the server tried to validate the transaction
     so the server aborted, so the client will abort too
  so: Thor's validation detects stale reads w/o timestamp on each record

Performance

Look at Figure 5
  AOCC is Thor
  comparing to ACBL: client talks to srvr to get write-locks,
   and to commit non-r/o xactions, but can cache read locks along with data
  why does Thor (AOCC) have higher throughput?
    fewer msgs; commit only, no lock msgs
  why does Thor throughput go up for a while w/ more clients?
    apparently a single client can't keep all resources busy
    maybe due to network RTT?
    maybe due to client processing time? or think time?
    more clients -> more parallel xactions -> more completed
  why does Thor throughput level off?
    maybe 15 clients is enough to saturate server disk or CPU
    abt 100 xactions/second, about right for writing disk
```

```
    why does Thor throughput *drop* with many clients?
      more clients means more concurrent xactions at any given time
      more concurrency means more chance of conflict
      for OCC, more conflict means more aborts, so more wasted CPU

  Conclusions
    fast client caching + transactions would be excellent
    distributed OCC very interesting, still an open research area
    avoiding per-record version #s doesn't seem compelling
    Thor's use of time was influential, e.g. Spanner
```