```
6.824 2017 Lecture 10: FaRM

course note: final project proposals due this Friday!
  can do final project or lab 4
  form group of 2-3 students
  please talk to us/email us about project ideas!

why are we reading this paper?
  many people want distributed transactions
  but they are thought to be slow
  this paper suggests that needn't be true -- very surprising performance!

big performance picture
  90 million *replicated* *persistent* *transactions* per second (Figure 7)
    1 million transactions/second per machine
    each with a few messages, for replication and commit
    very impressive
  a few other systems get 1 million ops/second per machine, e.g. memcached
    but not transactions + replicated + persistent (often not any of these!)
  perspective on 90 million:
    10,000 Tweets per second
    2,000,000 e-mails per second

how do they get high performance?
  data must fit in total RAM (so no disk reads)
  non-volatile RAM (so no disk writes)
  one-sided RDMA (fast cross-network access to RAM)
  fast user-level access to NIC
  transaction+replication protocol that exploits one-sided RDMA

NVRAM
  FaRM writes go to RAM, not disk -- eliminates a huge bottleneck
  can write RAM in 200 ns, but takes 10 ms to write hard drive, 100 us for SSD
    ns = nanosecond, ms = millisecond, us = microsecond
  but RAM loses content in power failure! not persistent by itself.
  why not just write to RAM of f+1 machines, to tolerate f failures?
    might be enough if failures were always independent
    but power failure is not independent -- may strike 100% of machines!
  so:
    batteries in every rack, can run machines for a few minutes
    power h/w notifies s/w when main power fails
    s/w halts all transaction processing
    s/w writes FaRM's RAM to SSD; may take a few minutes
    then machine shuts down
    on re-start, FaRM reads saved memory image from SSD
    "non-volatile RAM"
  what if crash prevents s/w from writing SSD?
    e.g bug in FaRM or kernel, or cpu/memory/hardware error
    FaRM copes with single-machine crashes by copying data
      from RAM of machines' replicas to other machines
      to ensure always f+1 copies
    crashes (other than power failure) must be independent!

why is the network often a performance bottleneck?
  the usual setup:
    app                       app
    ---                       ---
    socket buffers            buffers
    TCP                       TCP
    NIC driver                driver
    NIC  -------------------- NIC
  lots of expensive CPU operations:
    system calls
    copy messages
    interrupts
    and all twice if RPC
  slow:
    hard to build RPC than can deliver more than a few 100,000 / second
    wire b/w (e.g. 10 gigabits/second) is rarely the limit for short RPC
    these per-packet CPU costs are the limiting factor for small messages

Kernel bypass
  application access to NIC h/w is streamlined
  application directly interacts with NIC -- no system calls, no kernel
  shared memory mapping between app and NIC
  sender gives NIC an RDMA command
```

```
  for RPC, receiver s/w polls memory which RDMA writes

FaRM's network setup
  [hosts, 56 gbit NICs, expensive switch]
  NIC does "one-sided RDMA": memory read/write, not packet delivery
  sender says "write this data at this address", or "read this address"
    NIC *hardware* executes at the far end
    returns a "hardware acknowledgement"
  no interrupt, kernel, copy, read(), &c at the far end
  one server's throughput: 10+ million/second (Figure 2)
  latency: 5 microseconds (from their NSDI 2014 paper)
  FaRM uses RDMA in three ways:
    one-sided read of objects during transaction execution (also VALIDATE)
    RPC composed of one-sided writes to primary's logs or message queues
    one-sided write into backup's log

big challenge:
  how to use one-sided read/write for transactions and replication?
  protocols we've seen require receiver CPU to actively process messages
    e.g. Raft and two-phase-commit

let's review distributed transactions

remember this example:
  x and y are bank balances, maybe on different servers
  T1:             T2:
    add(x, 1)       tmp1 = get(x)
    add(y, -1)      tmp2 = get(y)
                    print tmp1, tmp2
  x and y start at $10
  we want serializability:
    results should be as if transactions ran one at a time in some order
  only two orders are possible
    T1 then T2 yields 11, 9
    T2 then T1 yields 10, 10
    serializability allows no other result

what if T1 runs entirely between T2's two get()s?
  would print 10,9 if the transaction protocol allowed it
  but it's not allowed!
what if T2 runs entirely between T1's two adds()s?
  would print 11,10 if the transaction protocol allowed it
  but it's not allowed!

two classes of concurrency control for transactions:
  pessimistic:
    wait for lock on first use of object; hold until commit/abort
    called two-phase locking
    conflicts cause delays
  optimistic:
    access object without locking; commit "validates" to see if OK
      valid: do the writes
      invalid: abort
    called Optimistic Concurrency Control (OCC)

FaRM uses OCC
  the reason: OCC lets FaRM read using one-sided RDMA reads
    server storing the object does not need to set a lock, due to OCC
  how does FaRM validate? we'll look at Figure 4 in a minute.

every FaRM server runs application transactions and stores objects
  an application transaction is its own transaction coordinator (TC)

FaRM transaction API (simplified):
  txCreate()
  o = txRead(oid)  -- RDMA
  o.f += 1
  txWrite(oid, o)  -- purely local
  ok = txCommit()  -- Figure 4

txRead
  one-sided RDMA to fetch object direct from primary's memory -- fast!
  also fetches object's version number, to detect concurrent writes

txWrite
  must be preceded by txRead
  just writes local copy; no communication
```

```
what's in an oid?
  <region #, address>
  region # indexes a mapping to [ primary, backup1, ... ]
  target NIC can use address directly to read or write RAM
    so target CPU doesn't have to be involved

server memory layout
  regions, each an array of objects
  object layout
    header with version # and lock
  for each other server
    (written by RDMA, read by polling)
    incoming log
    incoming message queue
  all this in non-volatile RAM (i.e. written to SSD on power failure)

every region replicated on one primary, f backups -- f+1 replicas
  [diagram of a few regions, primary/backup]
  only the primary serves reads; all f+1 see commits+writes
  replication yields availability if <= f failures
    i.e. available as long as one replica stays alive; better than Raft

transaction execution / commit protocol w/o failure -- Figure 4
  let's consider steps in Figure 4 one by one
  thinking about concurrency control for now (not replication)

LOCK (first message in commit protocol)
  TC sends to primary of each written object
  TC uses RDMA to append to its log at each primary
  LOCK record contains oid, version # xaction read, new value
  primary s/w polls log, sees LOCK, validates, sends "yes" or "no" reply message
  note LOCK is both logged in primary's NVRAM *and* an RPC exchange

what does primary CPU do on receipt of LOCK?
  (for each object)
  if object locked, or version != what xaction read, reply "no"
    implemented with atomic compare-and-swap
    "locked" flag is high-order bit in version number
  otherwise set the lock flag and return "yes"
  note: does *not* block if object is already locked

TC waits for all LOCK reply messages
  if any "no", abort
    send ABORT to primaries so they can release locks
    returns "no" from txCommit()

let's ignore VALIDATE and COMMIT BACKUP for now

TC sends COMMIT-PRIMARY to primary of each written object
  uses RDMA to append to primary's log
  TC only waits for hardware ack -- does not wait for primary to process log entry
  TC returns "yes" from txCommit()

what does primary do when it processes the COMMIT-PRIMARY in its log?
  copy new value over object's memory
  increment object's version #
  clear object's lock flag

example:
  T1 and T2 both want to increment x
  both say
    tmp = txRead(x)
    tmp += 1
    txWrite(x)
    ok = txCommit()
  x should end up with 0, 1, or 2, consistent with how many successfully committed

what if T1 and T2 are exactly in step?
  T1: Rx0  Lx  Cx
  T2: Rx0  Lx  Cx
  what will happen?

or
  T1:    Rx0 Lx Cx
  T2: Rx0          Lx  Cx
```

```
or
  T1: Rx0  Lx  Cx
  T2:             Rx0  Lx  Cx

intuition for why validation checks serializability:
  i.e. checks "was execution one at a time?"
  if no conflict, versions don't change, and commit is allowed
  if conflict, one will see lock or changed version #

what about VALIDATE in Figure 4?
  it is an optimization for objects that are just read by a transaction
  VALIDATE = one-sided RDMA read to fetch object's version # and lock flag
  if lock set, or version # changed since read, TC aborts
  does not set the lock, thus faster than LOCK+COMMIT

VALIDATE example:
x and y initially zero
T1:
  if x == 0:
    y = 1
T2:
  if y == 0:
    x = 1
(this is a classic test example for consistency)
T1,T2 yields y=1,x=0
T2,T1 yields x=1,y=0
aborts could leave x=0,y=0
but serializability forbids x=1,y=1

suppose simultaneous:
  T1:  Rx  Ly  Vx  Cy
  T2:  Ry  Lx  Vy  Cx
  the LOCKs will both succeed
  the VALIDATEs will both fail, since lock bits are both set
  so both will abort -- which is OK

how about:
  T1:  Rx  Ly  Vx      Cy
  T2:  Ry          Lx  Vy  Cx
  then T1 commits, T2 still aborts since T2's Vy sees T1's lock or higher version
but we can't have *both* V's before the other L's
so VALIDATE seems correct in this example
  and fast: faster than LOCK, no COMMIT required

what about fault tolerance?
  defense against losing data?
    durable? available?
  integrity of underway transactions despite crashes?
  partitions?

high-level replication diagram
  o o region 1
  o o region 2
  o CM
  o o o ZK

f+1 copies of each region to tolerate <= f failures in each region
  TCs send all writes to all copies (TC's COMMIT-BACKUP)
  not immediately available if a server crashes
    transaction reads and commits will wait
  but CM will soon notice, make a new copy, recover transactions

reconfiguration
  one ZooKeeper cluster (a handful of replicas)
    stores just configuration #, set of servers in this config, and CM
    breaks ties if multiple servers try to become CM
    chooses the active partition if partitioned (majority partition)
  a Configuration Manager (CM) (not replicated)
    monitors liveness of all servers via rapid ping
    manages reconfiguration
      renews leases
        only activates if it gets a response from majority of machines
      checks that at least one copy of each region exists
      assigns regions to primary/backup sets
      tells servers to make new copies
      manages completion of interrupted transactions
```

```
let's look back the at the Figure 4 commit protocol to see how
  any xaction that might have committed will be visible despite failed servers.
  "might have committed":
    TC might have replied "yes" to client
    primary might have revealed update to a subsequent read

after TC sees "yes" from all LOCKs and VALIDATEs,
  TC appends COMMIT-BACKUP to each backup's log
  after all ack, appends COMMIT-PRIMARY to each primary's log
  after one ack, reports "commit" to application

note TC replicates to backups; primaries don't replicate
  COMMIT-BACKUP contains written value, enough to update backup's state

why TC sends COMMIT-PRIMARY only after acks for *all* COMMIT-BACKUPs?
  a primary may execute as soon as it sees COMMIT-PRIMARY
    and show the update to other transactions
  so by that point each object's new value must in f+1 logs (per region)
    so f can fail without losing the new value
  if there's even one backup that doesn't have the COMMIT-BACKUP
    that object's writes are in only f logs
    all f could fail along with TC
    then we'd have exposed commit but maybe permanently lost one write!

why TC waits for an ack from a COMMIT-PRIMARY?
  so that there is a complete f+1 region that's aware of the commit
  before then, only f backups per region knew (from COMMIT-BACKUPs)
  but we're assuming up to f per region to fail

the basic line of reasoning for why recovery is possible:
  if TC could have reported "commit", or a primary could have exposed value,
  then all f+1 in each region have LOCK or COMMIT-BACKUP in log,
  so f can fail from any/every region without losing writes.
  if recovery sees one or more COMMIT-*, and a COMMIT-* or LOCK
    from each region, it commits; otherwise aborts.
    i.e. evidence TC decided commit, plus each object's writes.
    (Section 5.3, Step 7)

FaRM is very impressive; does it fall short of perfection?
  * works best if few conflicts, due to OCC.
  * data must fit in total RAM.
  * the data model is low-level; would need e.g. SQL library.
  * details driven by specific NIC features; what if NIC had test-and-set?

summary
  distributed transactions have been viewed as too slow for serious use
  maybe FaRM demonstrates that needn't be true
```