```
6.824 2021 Lecture 15: FaRM, Optimistic Concurrency Control

why are we reading about FaRM?
  another take on transactions+replication+sharding
    this is still an open research area!
  motivated by huge performance potential of RDMA NICs

how does FaRM differ from Spanner?
  both replicate and use two-phase commit (2pc) for transactions
  Spanner:
    a deployed system
    focuses on geographic replication
      e.g. copies on East and West coasts, in case data centers fail
    is most innovative for read-only transactions -- TrueTime
    performance: r/w xaction takes 10 to 100 ms (Tables 3 and 6)
  FaRM
    a research prototype, to explore potential of RDMA
    all replicas are in same data center (wouldn't make sense otherwise)
    RDMA restricts design options: thus Optimistic Concurrency Control (OCC)
    performance: 58 microseconds for simple transactions (6.3, Figure 7)
      i.e. 100 times faster than Spanner
    performance: throughput of 100 million/second on 90 machines (Figure 7)
      extremely impressive, particularly for transactions+replication
  They target different bottlenecks:
    Spanner: speed of light and network delays
    FaRM: CPU time on servers

the overall setup
  all in one data center
  configuration manager, using ZooKeeper, chooses primaries/backups
  sharded w/ primary/backup replication
    P1 B1
    P2 B2
    ...
    can recover as long as at least one replica of each shard
    i.e. f+1 replicas tolerate f failures
  transaction clients (which they run in the servers)
  transaction code acts as two-phase-commit Transaction Coordinator (TC)

how do they get high performance?
  sharding over many servers (90 in the evaluation)
  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 (non-volatile RAM)
  FaRM writes go to RAM, not disk -- eliminates a huge bottleneck
  RAM write takes 200 ns, hard drive write takes 10 ms, SSD write 100 us
    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!
  summary:
    NVRAM eliminates persistence write bottleneck
    leaving network and CPU as remaining bottlenecks

why is the network often a performance bottleneck?
  the usual setup for RPC over TCP over LAN:
    app                       app
    ---                       ---
    socket buffers            buffers
    TCP                       TCP
    NIC driver                driver
    NIC  -------------------- NIC
  lots of expensive CPU operations:
    system calls
    copy messages
    interrupts
  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
    per-packet CPU costs are the limiting factor for small messages

FaRM uses two networking ideas:
  Kernel bypass
  RDMA

Kernel bypass
  [diagram: FaRM user program, CPU cores, DMA queues, NIC]
  application directly interacts with NIC -- no system calls, no kernel
  NIC DMAs into/out of user RAM
  FaRM s/w polls DMA areas to check for new messages

RDMA (remote direct memory access)
  [src host, NIC, switch, NIC, target memory, target CPU]
  remote NIC directly reads/writes memory
    Sender provides memory address
    Remote CPU is not involved!
    This is "one-sided RDMA"
    Reads an entire cache line, atomically
    (Not sure about writes)
  RDMA NICs use reliable protocol, with ACKs
  one server's throughput: 10+ million/second (Figure 2)
  latency: 5 microseconds (from their NSDI 2014 paper)

Performance would be amazing if clients could directly access
  DB records on servers via RDMA!

Q: Can transactions just directly read/write with one-sided RDMA?

How to combine RDMA with replication and transactions?
  The protocols we've seen so far require active server participation.
  e.g. is that record locked?
       which is the latest version?
       is that write committed yet?
  Not immediately compatible with one-sided RDMA.

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:
    read objects without locking
    don't install writes until commit
    commit "validates" to see if other xactions conflicted
    valid: commit the writes
    invalid: abort
    called Optimistic Concurrency Control (OCC)

FaRM uses OCC
  the reason:
    OCC lets FaRM read using one-sided RDMA reads
    server needn't actively participate (no lock, due to OCC)
  how does FaRM validate? we'll look at Figure 4 in a minute.

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

what's an oid?
  <region #, address>
  region # indexes a mapping to [ primary, backup1, ... ]
  target RDMA NIC uses address directly to read or write RAM

server memory layout
  regions, each an array of objects
  object layout
    header with version #, and lock flag in high bit of version #
  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)

Figure 4: transaction execution / commit protocol
  let's consider steps in Figure 4 one by one
  focus on concurrency control (not fault tolerance)

Execute phase
  TC (the client) reads the objects it needs from servers
    including records that it will write
    using one-sided RDMA reads
    without locking
    this is the optimism in Optimistic Concurrency Control
  TC remembers the version numbers
  TC buffers writes

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
  LOCK is now logged in primary's NVRAM, in case power fails

what does primary do on receipt of LOCK?
  it polls incoming logs in RAM, sees our LOCK
  if object locked, or version != what xaction read, reply "no"
  otherwise set the lock flag and return "yes"
  lock check, version check, and lock set are atomic
    using atomic compare-and-swap instructuion
    "locked" flag is high-order bit in version number
    in case other CPU also processing a LOCK, or a client is reading w/ RDMA
  if object already locked, does not block, just replies "no"
    which will cause the TC to abort the xaction
```

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

let's ignore VALIDATE and COMMIT BACKUP for now

at this point primaries need to know TC's decision

TC appends COMMIT-PRIMARY to primaries' logs
  TC only waits for RDMA hardware acknowledgement (ack)
    does not wait for primary to process log entry
    hardware ack means safe in primary's NVRAM
  TC returns "yes" from txCommit()

when primary processes 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
    x = x + 1
  what results does serializability allow?
    i.e. what outcomes are possible if run one at a time?
    x = 2, both clients told "success"
    x = 1, one client told "success", other "aborted"
    x = 0, both clients told "aborted"

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 FaRM's OCC provides serializability:
  i.e. checks "was execution same as one at a time?"
  if there was no conflicting transaction:
    the versions won't have changed
  if there was a conflicting transaction:
    one or the other will see a 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 re-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 strong 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
  what will happen?
  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
  T1 commits
  T2 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: one-sided VALIDATE read rather than LOCK+COMMIT writes

a purely read-only FaRM transaction uses only one-sided RDMA reads
  no writes, no log records
  very fast!

what about fault tolerance?
  some computers crash and don't reboot
  most interesting if TC and some primaries crash
  but we assume one backup from each shard survives

the critical issue:
  if a transaction was interrupted by a failure,
    and a client could have been told a transaction committed,
    or a committed value could have been read by another xaction,
  then the transaction must be preserved and completed during recovery.

look at Figure 4.
a committed write might be revealed as soon the
  first COMMIT-PRIMARY is sent (since primary writes and unlocks).
so by then, all of the transaction's writes must be on all
  f+1 replicas of all relevant shards.
the good news: LOCK and COMMIT-BACKUP achieve this.
  LOCK tells all primaries the new value(s).
  COMMIT-BACKUP tells all backups the new value(s).
  TC doesn't send COMMIT-PRIMARY until all LOCKs and COMMIT-BACKUPS complete.
  backups may not have processed COMMIT-BACKUPs, but in NVRAM logs.

similarly, TC doesn't return to client until at least one
 COMMIT-PRIMARY is safe in primary log.
  without the COMMIT-PRIMARY, the risky case is:
    all backups with COMMIT-BACKUP for a shard fail after the TC returns client
    now there is not evidence to conclude that transaction committed
      the only evidence left for the transaction is the LOCK record,
      which was written before validation so we don't if trans committed/aborted
  writing the COMMIT-PRIMARY handles the risk, because the TC's
     decision will survive f failures of any shard.
  since there's one shard with a full set of COMMIT-BACKUP and COMMIT-PRIMARY.
  any of which is evidence that the primary decided to commit.

FaRM is very impressive; does it fall short of perfection?
  * works best if few conflicts, due to OCC.
  * data must fit in total RAM.
  * replication only within a datacenter (no geographic distribution).
  * 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?
    * requires somewhat unusual RDMA and NVRAM hardware.

  summary
    super high speed distributed transactions
    hardware is exotic (NVRAM and RDMA) but may be common soon
    use of OCC for speed and to allow fast one-sided RDMA reads
```