```
6.824 2020 Lecture 11: Chain Replication

Chain replication for supporting high throughput and availability
(OSDI 2004) by Renesse and Schneider

Context: replication state machines
  Two main approach:
  1. Run all operations through Paxos/Raft
    Relatively uncommon approach but see Spanner (and lab 3)
    Performance is challenging, especially if data is large
  2. Configuration server plus primary/backup replication
    For example: GFS, master plus P/B replication for chunks
    For example: VM-FT, test-and-set server plus P/B replication

Chain replication (CR)
  This paper about how to do P/B replication for approach 2
  - covers fail-over
  Influential: many build on CR
    Ceph, Parameter Server, COPS, FAWN.

What is Chain Replication (CR)?
  Goals: if client gets write reply, data is safe if even one server survives.
         and linearizable.
  S1, S2, S3, S4
  S1 is "head"
  S4 is "tail"
  Writes:
    Client sends to head
    Forwarded down the chain, in order
    Each server overwrites old data with new data
    Tail responds to client
  Reads:
    Client sends to tail
    Tail responds (no other nodes involved)

Configuration service's role
 The config service monitors the servers
 If a server is unreachable:
   config makes a new configuration and informs clients and servers
   no service during this time
 Config service avoids split brain

Would it be correct (linearizable) to let clients read any replica in CR?
  No.
  A read could see uncommitted data, might disappear due to a failure.
  A client could see a new value from one replica,
    and then an older value from a different (later) replica.
Thus, read only from the tail

Lecture question: Give an example scenario where a client could
  observe incorrect results (i.e., non-linearizable) if the head of
  the chain would return a response to the client as soon as it
  received an acknowledgment from the next server in the chain
  (instead of the tail responding).

Intuition for linearizability of CR?
  When no failures, almost as if the tail were the only server.
    Head picks an order for writes, replicas apply in that order,
      so they will stay in sync except for recent (uncommitted) writes.
    Tail exposes only committed writes to readers.

Failure recovery, briefly.
  Good news: every replica knows of every committed write.
  But need to push partial writes down the chain.
```

```
  If head fails, successor takes over as head, no commited writes lost.
  If tail fails, predecessor takes over as tail, no writes lost.
  If intermediate fails, drop from chain, predecessor may need to
    re-send recent writes.

Adding server
  Extend tail, briefly:
    copy tail's state to new server
      tail records updates while copy is in progress
    new server tells tail it isn't the tail anymore
      tail send all recorded updates to new server
      new server tells configuration server, it is tail
    new server starts serving requests
      old tail redirect client requests

Why is CR attractive (vs Raft)?
  Client RPCs split between head and tail, vs Raft's leader handles both.
  Head sends each write just once, vs Raft's leader sends to all.
  Reads involve just one server, not all as in Raft.
    Or majority with Raft read optimization
  Situation after failure simpler than in Raft (remember Figure 7).
    [Consider some scenarios]

Why is it attractive to let clients read any replica in CR?
  The opportunity is that the intermediate replicas may still
    have spare CPU cycles when the read load is high enough to saturate the tail.
  Moving read load from the tail to the intermediate nodes
    might thus yield higher read throughput on a saturated chain.
  The CR paper describes one way to achieve this goal:
    Split objects over many chains, each server participates in multiple chains.
    C1: S1 S2 S3
    C2: S2 S3 S1
    C3: S3 S1 S2
  This works if load is more or less evenly divided among chains.
  It often isn't.
    Maybe you could divide objects into even more chains.

Why can CR serve reads from replicas linearizably but Raft/ZooKeeper/&c cannot?
  Relies on being a chain, so that *all* nodes see each
    write before the write commits, so nodes know about
    all writes that might have committed, and thus know when
    to ask the tail.
  Raft/ZooKeeper can't do this because leader can proceed with a mere
    majority, so can commit without all followers seeing a write,
    so followers are not aware when they have missed a committed write.

Does that mean CR is strictly more powerful than Raft &c?
  No.
  All CR replicas have to participate for any write to commit.
  If a node isn't reachable, CR must wait.
  So not immediately fault-tolerant in the way that ZK and Raft are.

Equivalently, why can't 2nd node take over as head if it can't reach the head?
  Partition -- split brain -- the 2nd node must wait patiently.

How can we safely make use of a replication system that can't handle partition?
  A single "configuration manager" must choose head, chain, tail.
  Everyone (servers, clients) must obey or stop.
    Regardless of who they locally think is alive/dead.
  A configuration manager is a common and useful pattern.
    It's the essence of how GFS (master) and VMware-FT (test-and-set server) work.
    Usually Paxos/Raft/ZK for config service,
      data sharded over many replica groups,
      CR or something else fast for each replica group.
    Lab 4 works this way (though Raft for everything).
```