6.824 2006 Lecture 15: Viewstamped Replication

From "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems," Oki and Liskov, PODC 1988.

Recall overall topic: replicated fault-tolerant services
- E.g. lock server that can continue even if some servers crash.
- Last lecture: Paxos view-change (we'll use it in this lecture)
- This lecture: reconstructing the data after a failures (i.e. during view change)
- Next class: complete system built with these tools (Harp)

Overall plan for fault-tolerant service: primary-copy
- Elect a primary.
- Clients send all operations through primary.
- Primary processes client operations in some order.
- Primary sends each operation to backups, waits for ACKs.
- If primary fails, clients start using a backup as the primary.
- Problem: status of the last operation if primary fails.
- Problem: network partition might lead to two primaries.
- Problem: net or backup failure may cause a replica to miss some operations.

Core idea: quorums
- majority -> only one view
- intersection -> can find latest data from old view

Quorums are not as simple as they may seem:
- A, B, and C are running the lock service
- C is separated, leaving A and B
- A crashes and loses memory, and B separates
- A restarts and C's network connection is fixed
- A and C form a majority -- but do they know that C's data is stale?

Solution: Viewstamped Replication
- Harp uses a variant of viewstamped replication.
- I'll present a simplified form.
- More suitable for e.g. lock server than for file server.

Properties:
- Assume there are 2b + 1 replicas.
- Can tolerate failure (node or net) of up to b replicas.
  - This includes primary failures.
  - Will operate if 1b+1 replicas are reachable.
- Handles partition correctly:
  - If one partition has b + 1 or more, that partition can proceed.
  - A partition with b or fewer replicas does nothing.
- Mimics a single reliable server.
  - Will only operate if it can guarantee linear operation history
  - I.e. every server's state reflects same op sequence (harder than FAB)
  - And guarantees not to forget an ACKed operation

Overview of Viewstamped Replication
- System goes through a series of views, one at a time.
- A "view" consists of a primary and the replicas it can talk to

- Start a new view if set of reachable nodes changes (e.g. primary crashes)
- In each view:
  - Agree on the primary and set of backups (much like Paxos)
  - Recover latest state from previous view
  - Primary accepts client operations, sends to other replicas

Why the system is going to be correct
- I.e. mimic a single reliable server
- A view must have a majority, so one view at a time
- Primary sequences client operations within each view
- Primary sends each operation to every server in the view
- Next view must also be a majority, thus include one replica from prev view
  - So all operations that completed in previous view known to next view
  - This part is harder than it seems and it's the core of the technique.

Data types:
- viewid: <viewcount, replica_id>
- viewstamp: <viewid, opcount>

State maintained by each replica:
- cur_viewid
- data
- last_viewstamp
- max_viewid
- crashed

cur_viewid is on disk, preserved even if there's a crash.
- Others (including data) are in memory, lost in a crash.

Operation within a view:
- [draw a timeline, to help understand last operation issues]
- View consists of primary and at least b other replicas.
- Clients send operations to the primary.
  - They know because non-primary replicas will redirect them.
- Primary picks the next client operation.
- Primary assigns that operation the next viewstamp.
- Primary sends the operation+viewstamp to all the replicas *in the view*.
  - Thus possible to proceed despite some failed replicas.
- Primary waits for all ACKs.
  - (otherwise different replicas apply different subsets of operations.)
- Primary sends ACK to client.

When does a View Change occur?
- Primary notices a backup in the view has died.
- Any node in the view notices the primary has died.
- Any node in the view notices a node outside the view has revived.

First step: pick a new primary and a new view #
- This is much like Paxos/viewchange from previous lecture
- One or more servers send invitations (maybe simultaneously)
- Each invite contains a new viewid: higher than max_viewid.

```
  We want highest viewid to win:
    higher viewcount always wins.
    If viewcounts the same, higher replica_id wins.
  If a node sees an invite with viewid > max_viewid,
    drop everything and participate in election.
  Reachable replicas will agree on a viewid if nothing fails/joins
    for long enough.
  Node that proposed the winning viewid is the new primary.

Primary needs to reconstruct state.
  Which server (if any) has up-to-date state, reflecting all commits?
  We don't want to forget about any ACKed operations.
  So primary needs to establish three properties:
  1. At most one view at a time, so well-defined "previous view".
  2. New view knows what the previous view and viewid was.
  3. New view knows last state of previous view.
     Up through the last operation that could have been ACKed to a
client.

Property 1 is satisfied if primary assembles a majority.

Property 2 is also satisfied with a majority.
  Any majority must include a server from the previous view.
  That server has the previous view's cur_viewid on disk.
    Even if it crashed.
  We're assuming disks don't fail.
    So we're really assuming a majority with intact disks.

Property 3 satisfied if we have one non-crashed server from the
previous view.
  Since all servers in previous view saw all committed operations.
  Old primary would not have committed w/o ACK from every replica in
old view.
  So if our one replica doesn't have the viewstamp, it didn't commit.
  Servers *know* if they crashed+rebooted since last view change.

New primary sends all servers in view the state from the server it has
chosen.
  All servers in view write cur_viewid to disk.
  New primary can now start processing client requests.

Are we guaranteed to include last operation if it was ACKed to client?
  Old primary only sends ACK after all replicas ACK
  So any non-crashed server from previous view has latest ACKed
operation
  So "yes"

What about operations that hadn't quite committed when the view change
started?
  Primary had sent operation to one server, but not all of them.
  Since primary had not gotten ACKs, it didn't send response to client.
  So we can commit it, or not commit it; either is legal.
  Depends on which server primary chooses as "most recent state".
  So we might accept a final operation that old primary never sent ACK
for
  Client may re-send, primary needs to know if a duplicate operation.
    So somehow transaction IDs or RPC seq #s must be in the state?
    Or operations must be safely repeatable.

When *can't* we form a new view?
  remember earlier example:
    A, B, and C are running the lock service
    C is separated, leaving A and B
    A crashes and loses memory, and B separates
    A restarts and C's network connection is fixed
    A and C form a majority -- but do they know that C's data is stale?
  How does VSR prevent formation of a new view?
    A kept last cur_viewid on disk.
    So it can tell C isn't up to date.
    And A knows it isn't up to date either, since it crashed.
  Must wait for B to rejoin; it has the latest state.

Note that A and B are enough by themselves
  Majority means they can figure out the latest viewid
  B not crashed means they have latest state

Similarly, B and C are enough

Are we ever forbidden to form a view when it would actually be correct?
  I.e. is this the best fault-tolerance possible?
  Suppose A has intact data from the previous view.
  But B and C have crashed and are still down.
  It would actually be correct for A to proceed alone.
  But it cannot know that, so it must wait.

Is this system perfect?
  Must copy full state around at each view change.
    OK if lock service, a disaster if NFS service.
  Vulnerable to power failures.
    Nodes lose state if power fails.
    May strike all nodes at the same time.
  Primary executes operations one at a time.
    This is probably slow, cannot overlap execute of one
      with send of the next.
    Would be even slower if every op had to be written to disk.

We'll see solutions to these problems in the Harp paper.
  Replicated NFS server that uses viewstamped replication.
```