Primary-backup replication

Jinyang Li

Some slides adapted from 6.824 notes
Fault tolerance via replication

• To tolerate machine failure, one must replicate data on >1 servers.

• Particularly important at scale.
  – Suppose a typical server crashes every month
  – How often some server crashes in a 10,000-server cluster?
    • $30 \times 24 \times 60 / 10000 = 4.3$ minutes
Consistency: Correctness of replication

• How to replicate data “correctly”?
• Some informal notion of correctness:
  – copies of the same data should (eventually) be the same
  – replicated system should “behave similarly” to its un-replicated system
  – (we’ll discuss formal correctness notion in Lec 3)
Challenges in achieving correctness

1. Concurrency
2. Machine failure
3. Network failure (e.g. network partition)
   • Particularly tricky because
     – one might mistake “slowness” for “failure”
     – one cannot tell 2 from 3.
Replication: a strawman
Strawman fails under concurrency

1. Put(x, 100)
2. Put(x, 200)
3. x = 200
4. x = 100
5. x = 200
6. x = 100
Order updates via primary

• To keep replica in sync, writes must be done in the same order
• Idea: use a designated server (primary) to determine the order of updates, others follow order
Primary determines order of updates

backup applies writes according to seqno order
Challenges in handling failure

• What if a backup timed out in acknowledging the replication?
  – Primary re-tries (works only if backup failure is transient)
  – Ignore the fact that a backup might not have processed the op.
    • How many is it safe to ignore?
    • How to make a backup catch up?
Challenges in handling failure

• What if the primary fails? Switch to another primary?
  – Could there be accidentally two “valid” primaries?
  – If an op is done before the switch, how to ensure it’s not “lost” after the switch?

• What to re-integrate a recovered server?
Failure handling: the stone age

- For a long time, people do it manually (with no guaranteed correctness)
  - One primary, one backup. Primary ignores temporary replication failure of a backup.
  - If primary crashes, human operator re-configures the system to use the former backup as new primary
  - Some ops done by primary might be “lost” at new primary
Viewstamp replication

- Original paper Oki and Liskov, 1988
- Viewstamp revisited: Liskov and Cowling, 2012
- A landmark work: The first (together with Paxos) to handle failure correctly.
VR overview: state machine replication

- Servers replicate a log of operations (instead of directly modifying state in-place)
  - Efficient for: comparing state among servers, sync-ing out-of-date servers
  - General: A log of operations may be
    - [key=x, data="..."] [key=x, data="..."] ...
    - [create /jinyang/x] [mv /jinyang/x, /jinyang/y]...
    - [update T set grade=10 where uid=123] [insert into T values ...] ...

- Correctness ←→ servers execute the same sequence of log
  - Operations must be deterministic
VR overview: primary-backup

- VR assumes a static configuration of servers, e.g. S0, S1, S2
- To handle primary failure, VR moves through a sequence of “views”
  - 0, 1, 2, 3, ....
  - Deterministic mapping from view-number to primary: primary = view-number % total_servers
  - e.g., 0→S0, 1→S1, 2→S2, 3→S0, ...
VR correctness conditions

• An op is “committed” if it is replicated by a threshold number of servers
  – Once committed, an op’s position in log is fixed

• Correctness
  – No two different ops are committed at the same log position
Key mechanism for correctness: quorum intersection

- Primary waits for quorum = majority servers (including self) before considering an op committed
- If backup is slow (or temporarily crashed), the primary can still commit as usual.
- Can two primaries commit different ops at same position?
Key mechanism for correctness: quorum intersection

S0: primary of v0, committed op0 at pos=10 on S0, S2, S3
S1: primary of v1, committed op1 at pos=10 on S1, S3, S4

S3 must have received from both S0 (view-0) and S1 (view-1) S3 could ensure that op0 and op1 are not replicated at the same position.
Key mechanism for correctness: quorum intersection

• Correctness condition: all committed ops in view v-1 must be known to primary in view v.

• How?
  – View v is only active after v’s primary has learned the log state of majority of nodes (at earlier views)
Basic VR protocol

• Server state:
  – currentViewNumber
  – lastNormalViewNumber
  – status (NORMAL, VIEW-CHANGE, or RECOVERING)
  – op-number
  – commit-number
  – log
VR normal case processing

On receiving Prepare m, server must:
- reject if its currentView > m.v
- reject if its status != NORMAL
- append cmd to log at position op-number (must process according to op-number order)

Primary considers a cmd committed after getting majority OKs
VR normal case processing

• What’s the latency of committing a command?
  – from the primary’s perspective
  – from the client’s perspective

• How does a backup learn a command’s commit status?
  – Primary piggybacks “commit-number” in its Prepare msg.
View-change: when the primary fails

On receiving ViewChange m, server does:
- reject if its currentView > m.v
- set currentView=m.v, status=VIEW-CHANGE
- return its log and the latest normal view.

View-change to v1 succeeds if majority replied OK. S1 decides on the newLog for v1.
View-change: what log for new view?

- Rule 1: Pick the log with the biggest `latestNormalView`

1. Network partitions S1, S2 from S0
2. v1 becomes active, primary S1 replicates B
3. Primary S0 replicates C, D at itself
4. Network heals. S1 crashes

Which log should v2 have?
View-change: what log for new view?

• Rule 2: if >1 logs exist in rule-1, pick the longest one

1. Primary S0 crashes after replicating C to S0 and S2.
2. S1 tries to view-change for v1

which log should v1 have?
Other details

- A recovered server might be out of sync with primary.
  - To recover, it transfers primary’s log

- How to transfer logs efficiently?
  - checkpointing etc.