Consensus: Paxos + RAFT

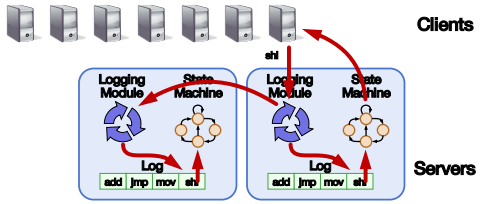
COS 418: Distributed Systems
Lecture 13

Mike Freedman

RAFT slides based on those from Diego Ongaro and John Ousterhout

1

Review: Primary-Backup Replication

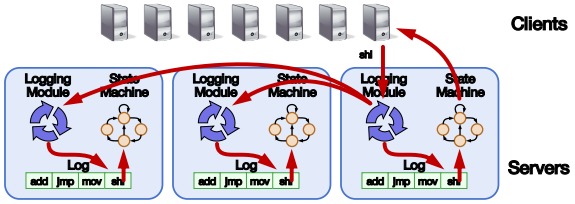


- Nominate one replica primary
  - Clients send all requests to primary
  - Primary orders clients' requests

2

2

From Two to Many Replicas



- Primary-backup with many replicas
  - Primary waits for acknowledgement from **all** backups
  - All updates to set of replicas needs to update shared disk

3

3

What else can we do with more replicas?

- Viewstamped Replication:
  - State Machine Replication for any number of replicas
  - Replica group: Group of $2f + 1$ replicas
    - Protocol can tolerate f replica crashes
- Differences with primary-backup
  - No shared disk (no reliable failure detection)
  - Don't need to wait for **all** replicas to reply
  - Need more replicas to handle f failures ($2f+1$ vs $f+1$)

4

4

The Need For a View Change

- So far: Works for *f* failed backup replicas
- But what if the *f* failures include a failed primary?
  - All clients' requests go to the failed primary
  - System halts despite merely *f* failures
- Need to **agree** on who the next primary should be

5

5

Consensus

Definition:

1. A general agreement about something
2. An idea or opinion that is shared by all the people in a group

6

Consensus Used in Systems

Group of servers want to:

- Make sure all servers in group receive the same updates in the same order as each other
- Maintain own lists (views) on who is a current member of the group, and update lists when somebody leaves/fails
- Elect a leader in group, and inform everybody
- Ensure mutually exclusive (one process at a time only) access to a critical resource like a file

7

7

Flavors of Paxos: Basic Paxos

- Run the full protocol each time
  - e.g., for each slot in the command log
- Takes 2 rounds until a value is chosen

8

Basic Paxos

Phase 1

- Proposer:
  - Choose proposal n, send <prepare, n> to all acceptors
- Acceptors:
  - If $n > n_{highest}$
    - $n_{highest} = n$
    - Reply < promise, n, ($n_{accepted}$, $v_{accepted}$) || Ø >
  - Else
    - Reply < prepare-failed >

Phase 2

- Proposer:
  - If promise from majority of acceptors,
    - Determine $v_{accepted}$ with highest $n_{accepted}$, if exists
    - Send <accept, (n, $v_{accepted}$ || v)> to all acceptors
- Acceptors:
  - If $n \geq n_{highest}$
    - Accept proposal:

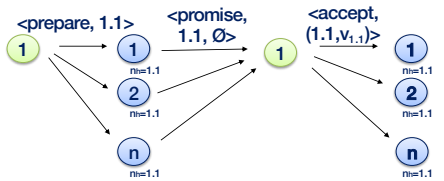
      $n_{accepted} = n_{highest} = n$

      $v_{accepted} = v$

9

9

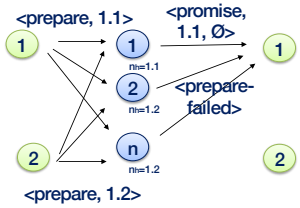
Example runs of Paxos



10

10

Example runs of Paxos

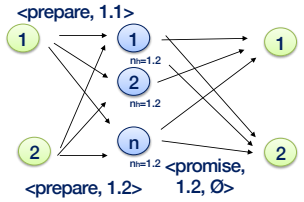


Case: Proposer 1 fails to get majority of promises for 1.1, because majority of acceptors had received 1.2 prior to 1.1

11

11

Example runs of Paxos

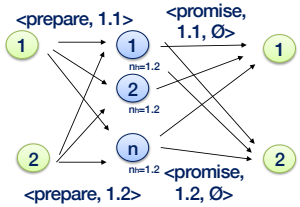


Case: Proposer 2 receives majority of promises for 1.2, progresses to phase 2

12

12

Example runs of Paxos

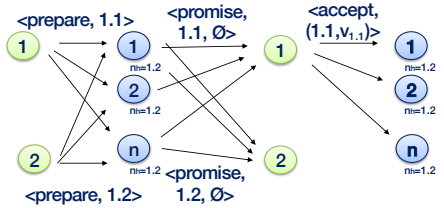


Case: Proposer 1 receives majority of promises for 1.1, progresses to phase 2

Case: Proposer 2 receives majority of promises for 1.2, progresses to phase 2

13

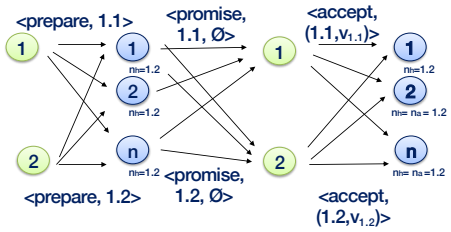
Example runs of Paxos



Reject <accept, 1.1>, given 1.1 < n_h

14

Example runs of Paxos

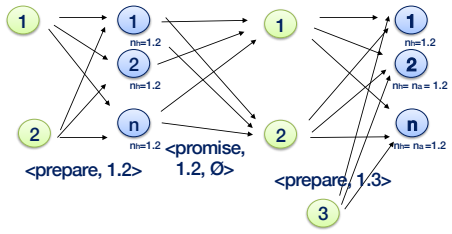


Reject <accept, 1.1>, given 1.1 < n_h

Proposal <1.2, $v_{1.2}$> accepted by majority of acceptors

15

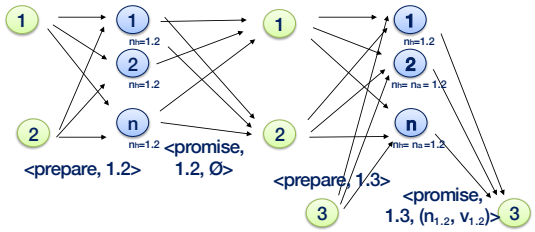
Example runs of Paxos



Proposal 1.3 is highest, but 1.2 accepted already by majority

16

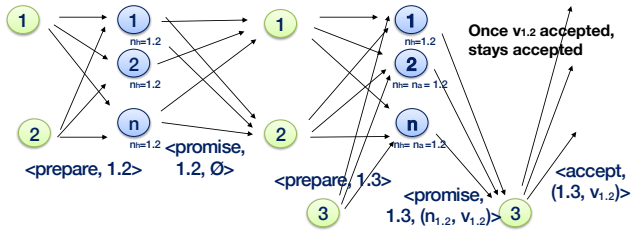
Example runs of Paxos



17

17

Example runs of Paxos



18

18

Flavors of Paxos: Multi-Paxos

- Elect a leader and have them run 2nd phase directly
  - e.g., for each slot in the command log
  - Leader election uses Basic Paxos
- Takes 1 round until a value is chosen
  - Faster than Basic Paxos
- Used extensively in practice!
  - RAFT is similar to Multi Paxos

19

19

RAFT: A CONSENSUS ALGORITHM FOR REPLICATED LOGS

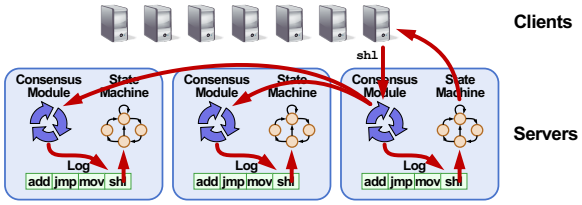
Diego Ongaro and John Ousterhout

Stanford University

20

20

Goal: Replicated Log



- Replicated log => replicated state machine
  - All servers execute same commands in same order
- Consensus module ensures proper log replication

21

Raft Overview

1. Leader election
2. Normal operation (basic log replication)
3. Safety and consistency after leader changes
4. Neutralizing old leaders
5. Client interactions
6. Reconfiguration

22

Server States

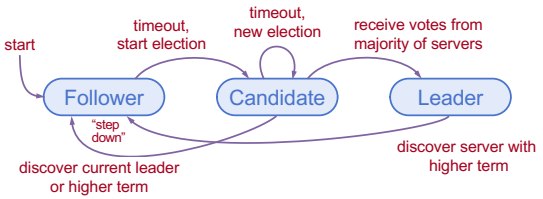
- At any given time, each server is either:
  - Leader: handles all client interactions, log replication
  - Follower: completely passive
  - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers

Follower Candidate Leader

23

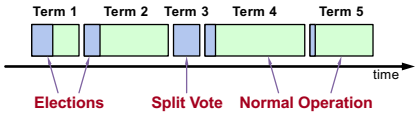
Liveness Validation

- Servers start as followers
- Leaders send heartbeats (empty AppendEntries RPCs) to maintain authority over followers
- If electionTimeout elapses with no RPCs (100-500ms), follower assumes leader has crashed and starts new election



24

Terms (aka epochs)



- Time divided into terms
  - Election (either failed or resulted in 1 leader)
  - Normal operation under a single leader
- Each server maintains current term value
- Key role of terms: identify obsolete information

25

25

Elections

- Start election:
  - Increment current term, change to candidate state, vote for self
- Send RequestVote to all other servers, retry until either:
  1. Receive votes from majority of servers:
     - Become leader
     - Send AppendEntries heartbeats to all other servers
  2. Receive RPC from valid leader:
     - Return to follower state
  3. No-one wins election (election timeout elapses):
     - Increment term, start new election

26

26

Elections

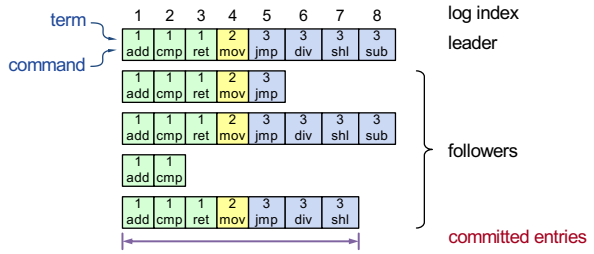
- Safety: allow at most one winner per term
  - Each server votes only once per term (persists on disk)
  - Two different candidates can't get majorities in same term

- Liveness: some candidate eventually wins
  - Each choose election timeouts randomly in [T, 2T]
  - One usually initiates and wins election before others start
  - Works well if T >> network RTT

27

27

Log Structure

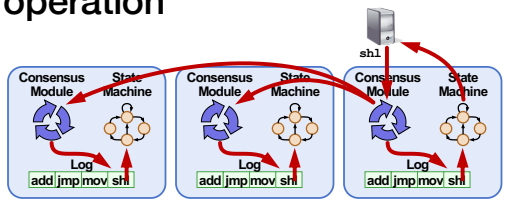


- Log entry = < index, term, command >
- Log stored on stable storage (disk); survives crashes
- Entry committed if known to be stored on majority of servers
  - Durable / stable, will eventually be executed by state machines

28

28

Normal operation

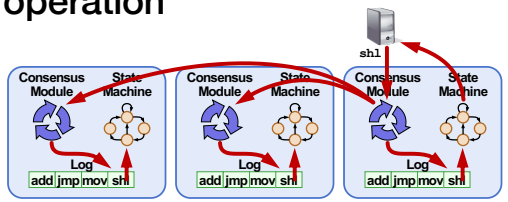


- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries RPCs to followers
- Once new entry committed:
  - Leader passes command to its state machine, sends result to client
  - Leader piggybacks commitment to followers in later AppendEntries
  - Followers pass committed commands to their state machines

29

29

Normal operation

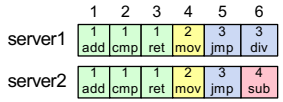


- Crashed / slow followers?
  - Leader retries RPCs until they succeed
- Performance is "optimal" in common case:
  - One successful RPC to any majority of servers

30

30

Log Operation: Highly Coherent

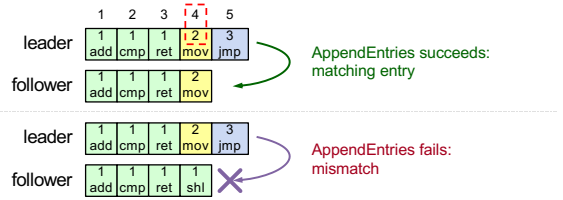


- If log entries on different server have same index and term:
  - Store the same command
  - Logs are identical in all preceding entries
- If given entry is committed, all preceding also committed

31

31

Log Operation: Consistency Check



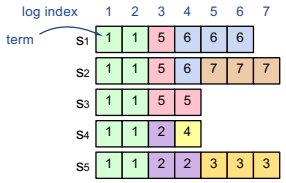
- AppendEntries has <index,term> of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects
- Implements an induction step, ensures coherency

32

32

Leader Changes

- New leader's log is truth, no special steps, start normal operation
  - Will eventually make follower's logs identical to leader's
  - Old leader may have left entries partially replicated
- Multiple crashes can leave many extraneous log entries



33

33

Safety Requirement

Once log entry applied to a state machine, no other state machine must apply a different value for that log entry

- Raft safety property: If leader has decided log entry is committed, entry will be present in logs of all future leaders
- Why does this guarantee higher-level goal?
  1. Leaders never overwrite entries in their logs
  2. Only entries in leader's log can be committed
  3. Entries must be committed before applying to state machine

Committed → Present in future leaders' logs

Restrictions on commitment

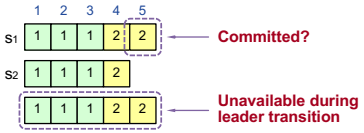
Restrictions on leader election

34

34

Picking the Best Leader

Can't tell which entries committed!

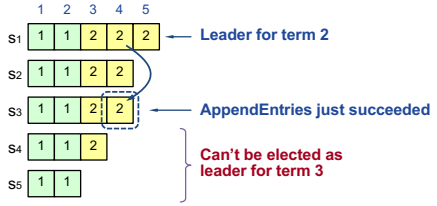


- Elect candidate most likely to contain all committed entries
  - In RequestVote, candidates incl. index + term of last log entry
  - Voter V denies vote if its log is "more complete": (newer term) or (entry in higher index of same term)
  - Leader will have "most complete" log among electing majority

35

35

Committing Entry from Current Term

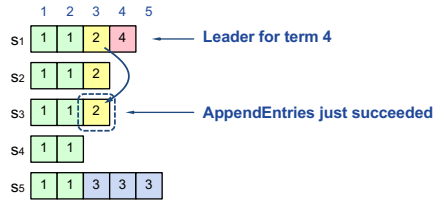


- Case #1: Leader decides entry in current term is committed
- Safe: leader for term 3 must contain entry 4

36

36

Committing Entry from Earlier Term

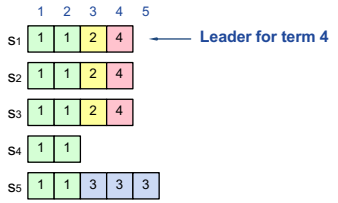


- Case #2: Leader trying to finish committing entry from earlier
- Entry 3 not safely committed:
  - s_5 can be elected as leader for term 5 (how?)
  - If elected, it will overwrite entry 3 on s_1, s_2, and s_3

37

37

New Commitment Rules

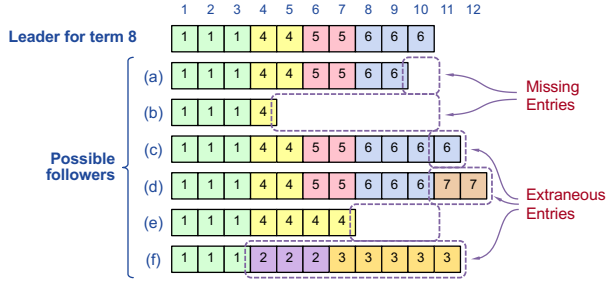


- For leader to decide entry is committed:
  1. Entry stored on a majority
  2. ≥ 1 new entry from leader's term also on majority
- Example; Once e4 committed, s_5 cannot be elected leader for term 5, and e3 and e4 both safe

38

38

Challenge: Log Inconsistencies

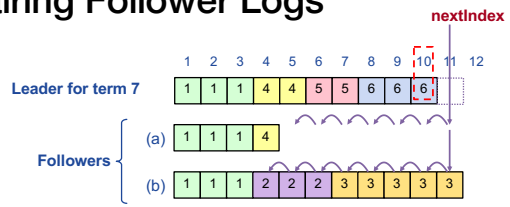


Leader changes can result in log inconsistencies

39

39

Repairing Follower Logs

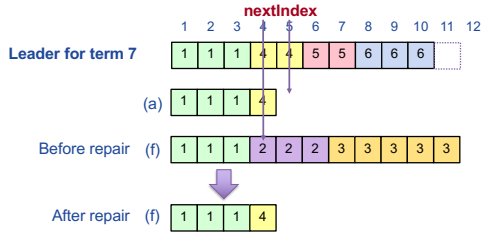


- New leader must make follower logs consistent with its own
  - Delete extraneous entries
  - Fill in missing entries
- Leader keeps nextIndex for each follower:
  - Index of next log entry to send to that follower
  - Initialized to (1 + leader's last index)
- If AppendEntries consistency check fails, decrement nextIndex, try again

40

40

Repairing Follower Logs



41

Neutralizing Old Leaders

- Leader temporarily disconnected
  → other servers elect new leader
  → old leader reconnected
  → old leader attempts to commit log entries
- Terms used to detect stale leaders (and candidates)
  - Every RPC contains term of sender
  - Sender's term < receiver:
    - Receiver: Rejects RPC (via ACK which sender processes…)
  - Receiver's term < sender:
    - Receiver reverts to follower, updates term, processes RPC
- Election updates terms of majority of servers
  - Deposed server cannot commit new log entries

42

Client Protocol

- Send commands to leader
  - If leader unknown, contact any server, which redirects client to leader
- Leader only responds after command logged, committed, and executed by leader
- If request times out (e.g., leader crashes):
  - Client reissues command to new leader (after possible redirect)
- Ensure exactly-once semantics even with leader failures
  - E.g., Leader can execute command then crash before responding
  - Client should embed unique request ID in each command
  - This unique request ID included in log entry
  - Before accepting request, leader checks log for entry with same id

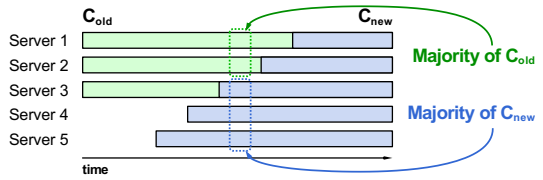
43

RECONFIGURATION

44

Configuration Changes

- View configuration: { leader, { members }, settings }
- Consensus must support changes to configuration: e.g., replace failed machine, change degree of replication
- Cannot switch directly from one config to another: conflicting majorities could arise

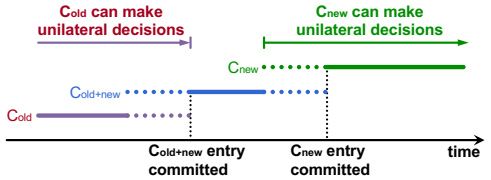


45

45

2-Phase Approach via Joint Consensus

- Joint consensus in intermediate phase: need majority of both old and new configurations for elections, commitment
- Configuration change just a log entry; applied immediately on receipt (committed or not)
- Once joint consensus is committed, begin replicating log entry for final config

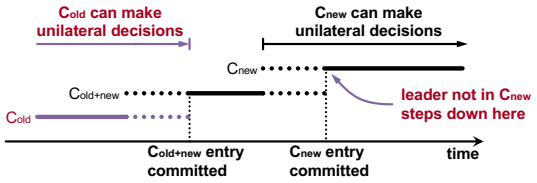


46

46

2-Phase Approach via Joint Consensus

- Any server from either configuration can serve as leader
- If leader not in C_{new}, must step down once C_{new} committed



47

47