Raft: A Consensus Algorithm for Replicated Logs

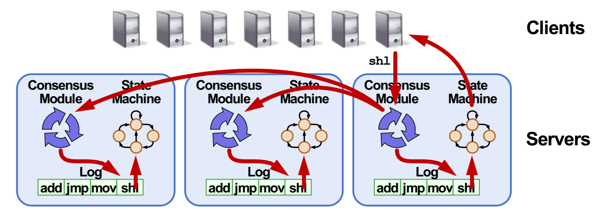
COS 418: Distributed Systems
Lectures 13-14

Mike Freedman, Jialin Ding

RAFT slides based on those from Diego Ongaro and John Ousterhout

1

Goal: Replicated Log



- Replicated log => replicated state machine
  - All servers execute same commands in same order
  - Group of $2f + 1$ replicas can tolerate f replica crashes
- Consensus module ensures proper log replication

2

2

Consensus

Definition:

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

Where do we use consensus?

- What is the order of operations
- Which operations are fully executed (committed) and not
- Who are the members of the group
- Who are the leaders of the group

3

Raft Overview

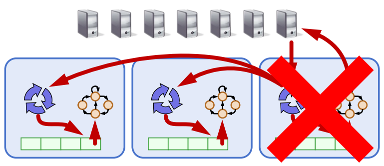
1. Leader election
2. Normal operation (basic log replication)
3. Safety and consistency after leader changes
4. Neutralizing old leaders
5. Client interactions
6. Reconfiguration

4

4

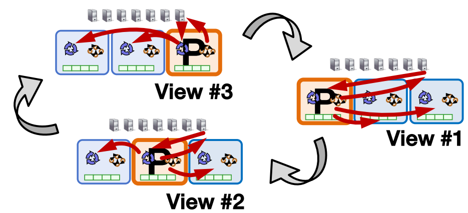
The Need For a Leader Election

- Recall consensus-based replication easier 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



Leaders and Views

- Let different replicas assume role of leader (primary) over time
- System moves through a sequence of views
  - View = { leader, { members }, settings }

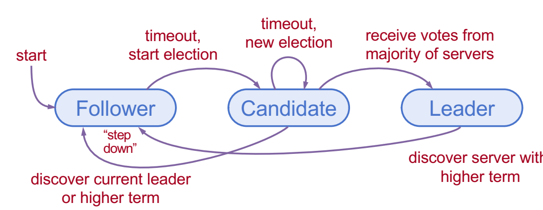


Server States

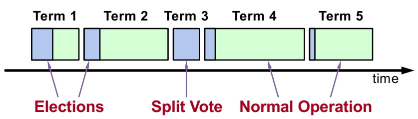
- At any given time, each server is either:
  - Leader: handles all client interactions, log replication
  - Follower: completely passive
  - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers

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



Terms (aka epochs)



- Time divided into non-fixed-time **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

9

9

Elections

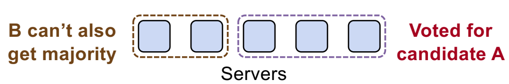
- Start election:
  - Increment current term, change to candidate state, vote for self
- Send RequestVote to all other servers, retry until either:
  1. Receive votes from majority of servers:
     - Become leader
     - Send AppendEntries heartbeats to all other servers
  2. Receive RPC from valid leader: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

10

10

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

11

11

Elections

Technique used throughout distributed systems:

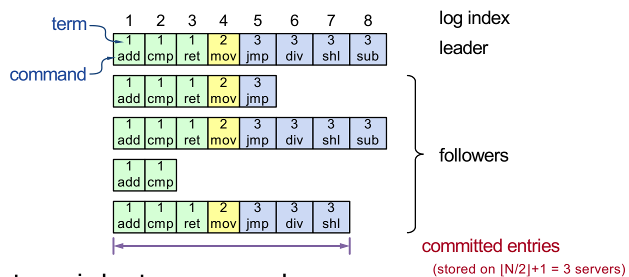
Desynchronizes behavior without centralized coordination!

- 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

12

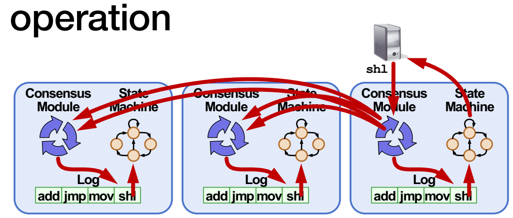
12

Log Structure



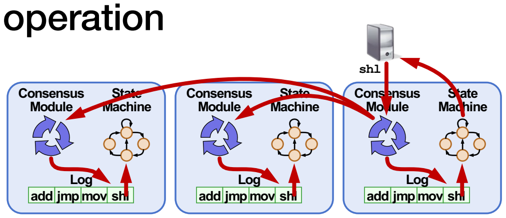
- Log entry = < index, term, command >
- Log stored on stable storage (disk); survives crashes
- Entry committed if known to be stored on majority of servers
  - Durable / stable, will eventually be executed by state machines

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

Normal operation



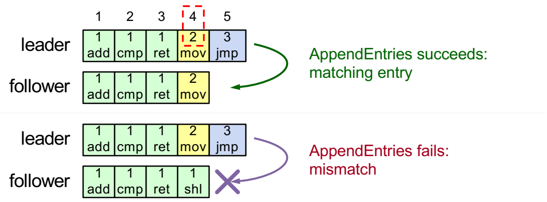
- Crashed / slow followers?
  - Leader retries RPCs until they succeed
- Performance is “optimal” in common case:
  - One successful RPC to any majority of servers

Log Operation: Highly Coherent

	1	2	3	4	5	6
server1	1 add	1 cmp	1 ret	2 mov	3 jmp	3 div
server2	1 add	1 cmp	1 ret	2 mov	3 jmp	4 sub

- 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

Log Operation: Consistency Check



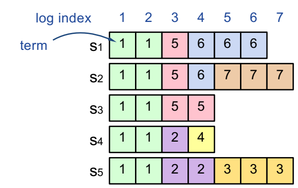
- AppendEntries has <index,term> of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects
- Implements an induction step, ensures coherency

17

17

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



18

18

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

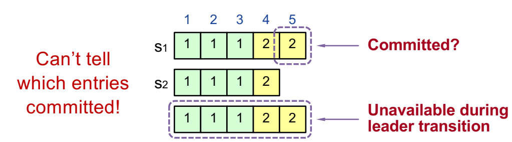
Committed → Present in future leaders' logs

Restrictions on commitment | Restrictions on leader election

19

19

Picking the Best Leader

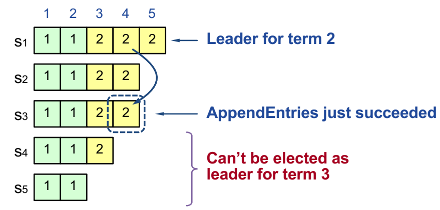


- Elect candidate most likely to contain all committed entries
  - In RequestVote, candidates incl. index + term of last log entry
  - Voter V denies vote if its log is "more complete":
    (newer term) or (entry in higher index of same term)
  - Leader will have "most complete" log among electing majority

20

20

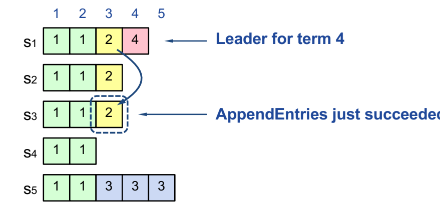
Committing Entry from Current Term



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

21

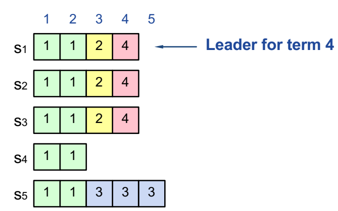
Problem: Committing Entry from Earlier Term



- 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

22

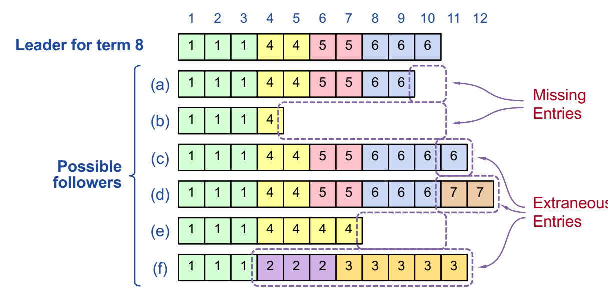
Solution: New Commitment Rules



- 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

23

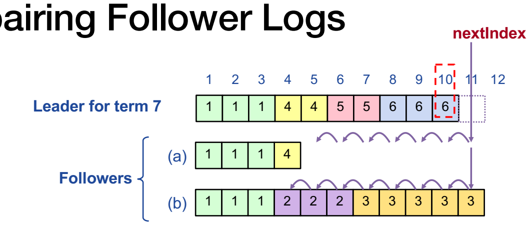
Challenge: Log Inconsistencies



Leader changes can result in log inconsistencies

24

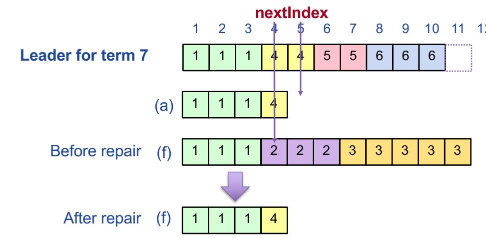
Repairing Follower Logs



- 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

25

Repairing Follower Logs



26

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

27

27

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

28

28

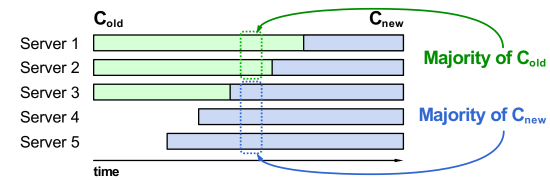
RECONFIGURATION

29

29

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

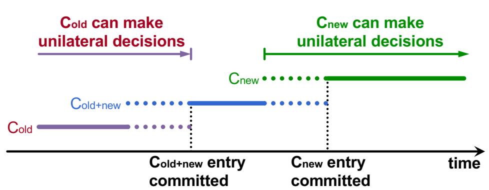


30

30

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

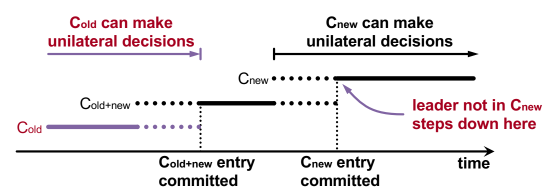


31

31

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



32

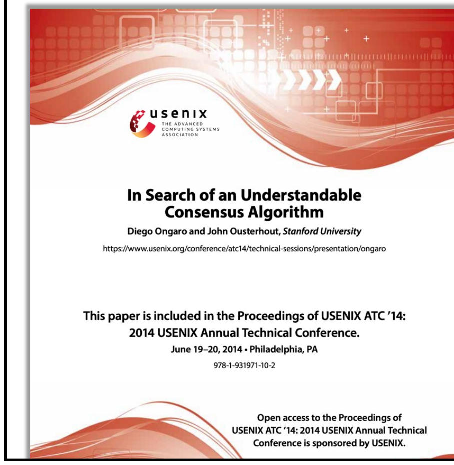
32

Summary

- RAFT “looks like a single machine” that does not fail
  – Use majority (f+1) out of 2f+1 replicas to make progress
- RAFT is similar to multi-paxos / viewstamped replication
  – Details make it easier to understand and implement
- Strong leader add constraints, but makes things simple
  – Only vote for a leader with a log ≥ your log
  – Leader’s log is canonical, gets others replica’s logs to match

33

33



usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

In Search of an Understandable Consensus Algorithm

Diego Ongaro and John Ousterhout, *Stanford University*

https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

This paper is included in the Proceedings of USENIX ATC ’14: 2014 USENIX Annual Technical Conference.
June 19–20, 2014 • Philadelphia, PA
978-1-931971-10-2

Open access to the Proceedings of USENIX ATC ’14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

Production use of Raft [edit]

- CockroachDB uses Raft in the Replication Layer.[5]
- Etcd uses Raft to manage a highly-available replicated log [6]
- Hazelcast uses Raft to provide its CP Subsystem, a strongly consistent layer for distributed data structures. [7]
- MongoDB uses a variant of Raft in the replication set.
- Neo4j uses Raft to ensure consistency and safety. [8]
- RabbitMQ uses Raft to implement durable, replicated FIFO queues. [9]
- ScyllaDB uses Raft for metadata (schema and topology changes) [10]
- Splunk Enterprise uses Raft in a Search Head Cluster (SHC) [11]
- TiDB uses Raft with the storage engine TiKV.[12]
- YugabyteDB uses Raft in the DocDB Replication [13]
- ClickHouse uses Raft for in-house implementation of ZooKeeper-like service [14]
- Redpanda uses the Raft consensus algorithm for data replication [15]

34

34