Raft: A consensus algorithm for replicated logs

Etienne Renault

2 octobre 2020

https://www.lrde.epita.fr/~renault/teaching/algorep/

Raft

Goal

Replicate logs (commands) in a set of servers

Overview

Once all servers agree for a log entry, all server can run it.
$\Rightarrow$ All server will then compute the same value (replication)

Note. Suppose that each server run a deterministic program (state machine)

Limitations and Restrictions

Progress

System makes progress as long as any **majority** of servers are up

Fault Tolerance

Support fail-stop and delayed-lost messages.

$\Rightarrow$ **Not Byzantine**

Approaches to consensus

Symmetric (leader-less)

- All servers have equal roles
- Client can contact any server
- $\Rightarrow$ Paxos style

Asymmetric (leader-based)

- At any given time, one server is in charge, others accepts its decision
- Clients communicate with the leader
- $\Rightarrow$ Raft style

Raft Summary

1. Leader election
2. Normal operation
3. Safety and consistency
4. Neutralize old leaders
5. Client protocol
6. Configuration changes

A word on Raft

Raft decomposes the problem in two phases :

- **Normal operations :**
  - the leader propagates information
  - More efficient than leader-less approaches
- **Leader changes :** may leave the system in an inconsistent state that the next leader has to cleanup.

Raft is an RPC based protocol !

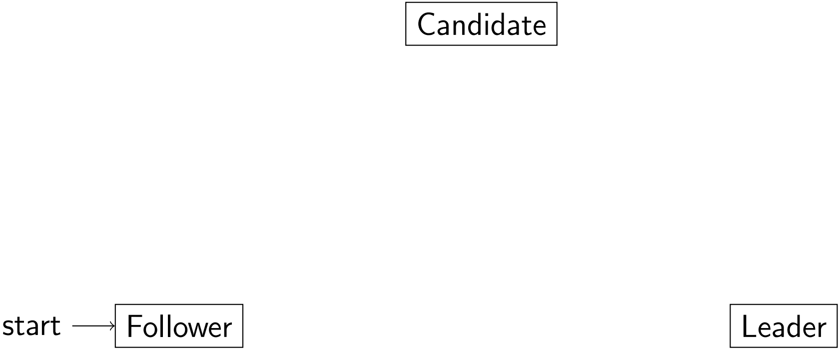
Server State 1/2

A Raft Server can be in three different states :

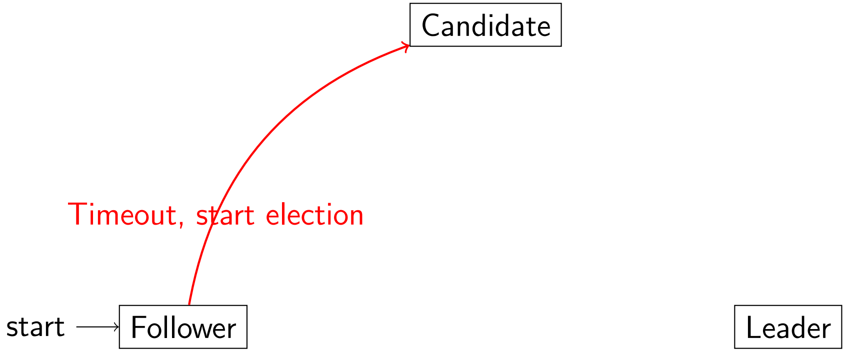
- **Leader** : Handle Client communication and log replication
- **Follower** : Passive component, response to messages only
- **Candidate** : used to elect a new leader.

At most 1 viable leader at a time

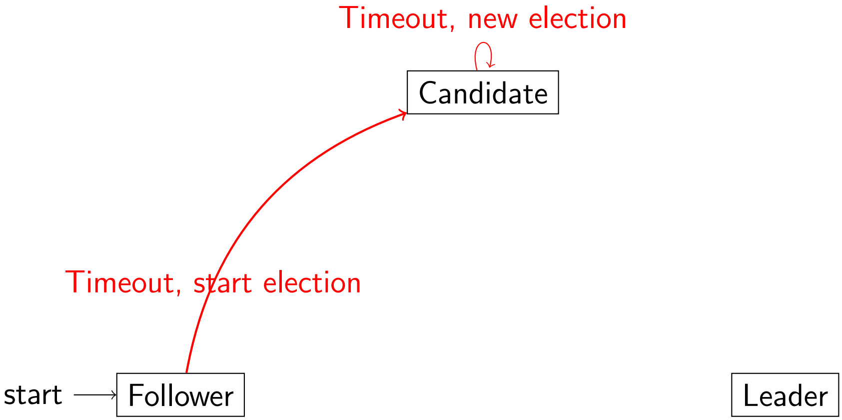
Server State 2/2



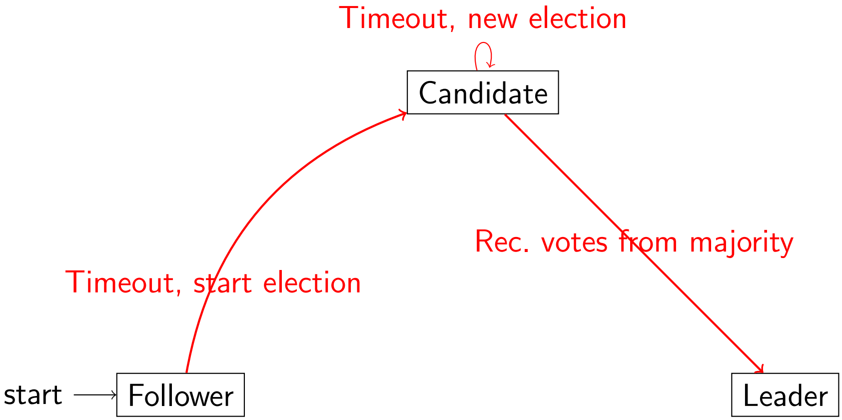
Server State 2/2



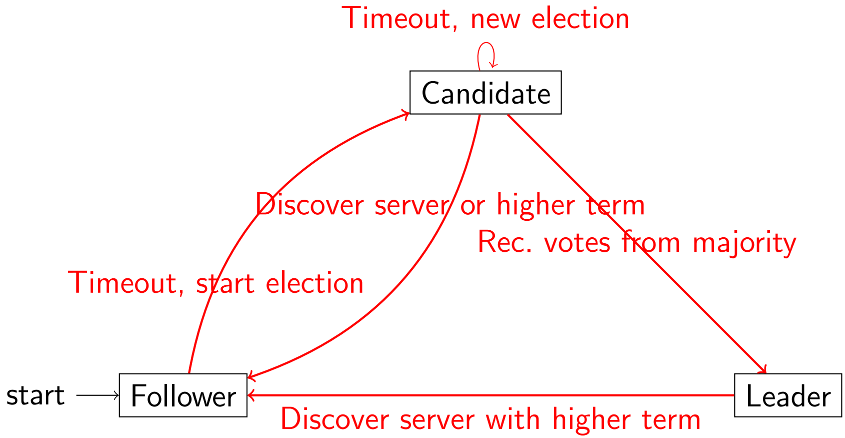
Server State 2/2



Server State 2/2



Server State 2/2



Time divided into Terms

Each term is uniquely identified

Term description :

- 2 phases per Term : Election / normal operations
- at most one leader per term
- some term have no leader
- Objective $\Rightarrow$ identify obsolete information

Each server maintains current term value (into disk)

Hearbeats

Leaders send heartbeats to maintain authority
(empty AppendEntries RPCs)

If election timeout elapses with no RPCs

- Follower assumes leader has crashed
- Follower starts new election
- Timeouts typically 100-500ms

Election Basics

- Increment current term
- Change to Candidate state
- Vote for self
- Send RequestVote RPCs 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

Properties of the election

Safety : allow at most one winner per term

- Each server gives out only one vote per term (persist on disk)
- Two different candidates can't accumulate majorities in same term

Liveness : some candidate must eventually win

- Choose election timeouts randomly in [T, 2T]
- One server usually times out and wins election before others wake up
- Works well if T greater broadcast time

How to replicate log entries ?

Each server has its own copy of the log

- A log structure is divided into entries.
- Entries are identified by indexes (position in log)
- Entries contains :
  - command for the state machine
  - a term number, that corresponds to the term number where the entry was created by the leader

An entry is committed if known and stored by a majority of server
$\Rightarrow$ The command can the be executed

Normal Operations

1. Client sends command to leader
2. Leader appends command to its log
3. Leader sends AppendEntries RPCs to followers
4. Once new entry committed :
   - Leader passes command to its state machine, returns result to client
   - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
   - Followers pass committed commands to their state machines
5. Crashed/slow followers ? ⇒ Leader retries RPCs until they succeed

Performance is optimal in common case

Log consistency

High level of coherency between logs

1. If log entries on different servers have same index and term
   $\Rightarrow$ They store the same command
   $\Rightarrow$ The logs are identical in all preceding entries

2. If a given entry is committed, all preceding entries are also committed

Consistency check

Each AppendEntries RPC contains index, term of entry preceding new ones

Follower must contain matching entry ; otherwise it rejects request

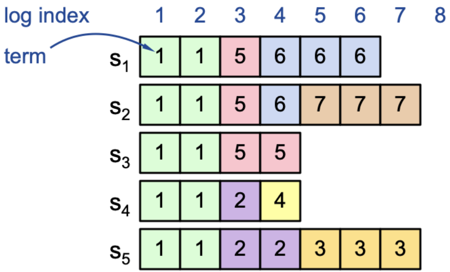
Implements an induction step, ensures coherency

Leader Changes

- At beginning of new leader's term :
  - Old leader may have left entries partially replicated
  - No special steps by new leader : just start normal operation
  - Leader's log is "the truth"
  - Will eventually make follower's logs identical to leader's
  - Multiple crashes can leave many extraneous log entries

Leader Changes

- At beginning of new leader's term :
  - Old leader may have left entries partially replicated
  - No special steps by new leader : just start normal operation
  - Leader's log is "the truth"
  - Will eventually make follower's logs identical to leader's
  - Multiple crashes can leave many extraneous log entries



Safety Requirement

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

Raft safety property

If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders

- Leaders never overwrite entries in their logs
- Only entries in the leader's log can be committed
- Entries must be committed before applying to state machine

Safety Requirement

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

Raft safety property

If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders

- Leaders never overwrite entries in their logs
- Only entries in the leader's log can be committed
- Entries must be committed before applying to state machine

$\Rightarrow$ Committed implies Present in future leaders' logs

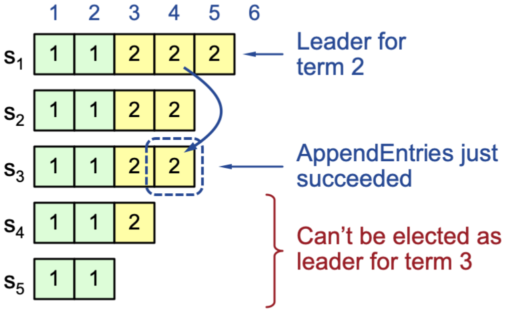
Picking the Best Leader

During elections, choose candidate with log most likely to contain all committed entries

- Candidates include log info in RequestVote RPCs (index & term of last log entry)
- Voting server V denies vote if its log is "more complete" : $(lastTermV > lastTermC) || (lastTermV == lastTermC) \&\& (lastIndexV > lastIndexC)$

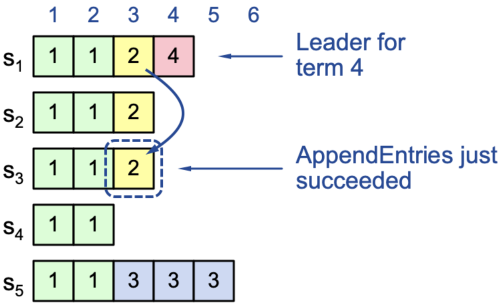
Example 1

Leader decides entry in current term is committed
⇒ Safe : leader for term 3 must contain entry 4



Example 1

Leader is trying to finish committing entry from an earlier term
$\Rightarrow$ Entry 3 not safely committed
s5 can be elected as leader for term 5
If elected, it will overwrite entry 3 on s1, s2, and s3 !

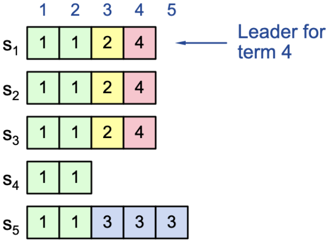


The aforementionned rules must be refined !

New Commitment Rules

For a leader to decide an entry is committed :

- Must be stored on a majority of servers
- At least one new entry from leader's term must also be stored on majority of servers



Once entry 4 committed s5 cannot be elected leader for term 5 and entries 4 and 5 are safe

Log Inconsistencies

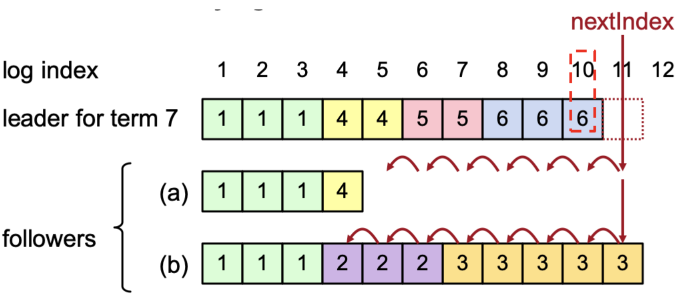
Leader changes can result in log inconsistencies

New leader must make follower logs consistent with its own

- Delete extraneous entries
- Fill in missing entries

Repairing Follower logs

When AppendEntries consistency check fails,
decrement nextIndex and try again
(Leader keeps nextIndex for each follower)



Neutralizing Old Leaders

Deposed leader may not be dead

Temporarily disconnected from network OR Other servers elect a new leader OR Old leader becomes reconnected, attempts to commit log entries

Terms used to detect stale leaders (and candidates)

$\Rightarrow$ Every RPC contains term of sender

$\Rightarrow$ Comparison between sender's term and receiver's one

if mismatch notify sender !

Client Protocol

1. Send commands to leader
   - If leader unknown, contact any server
   - If contacted server not leader, it will redirect to leader

Leader does not respond until command has been logged, committed, and executed by leader's state machine

In case of request timeout (leader crash)

Client reissues command to some other server
Eventually redirected to new leader
Retry request with new leader

Client Protocol (2)

What if leader crashes after executing command, but before responding ?

$\Rightarrow$ Must not execute command twice

Solution : client embeds a unique id in each command :

- Server includes id in log entry
- Before accepting command, leader checks its log for entry with that id
- If id found in log, ignore new command, return response from old command

Configuration Changes

Cannot switch directly from one configuration to another : conflicting majorities could arise

Raft uses a 2-phase approach :

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

Raft Summary

1. Leader election
2. Normal operation
3. Safety and consistency
4. Neutralize old leaders
5. Client protocol
6. Configuration changes