RAFT continued

Distributed Systems

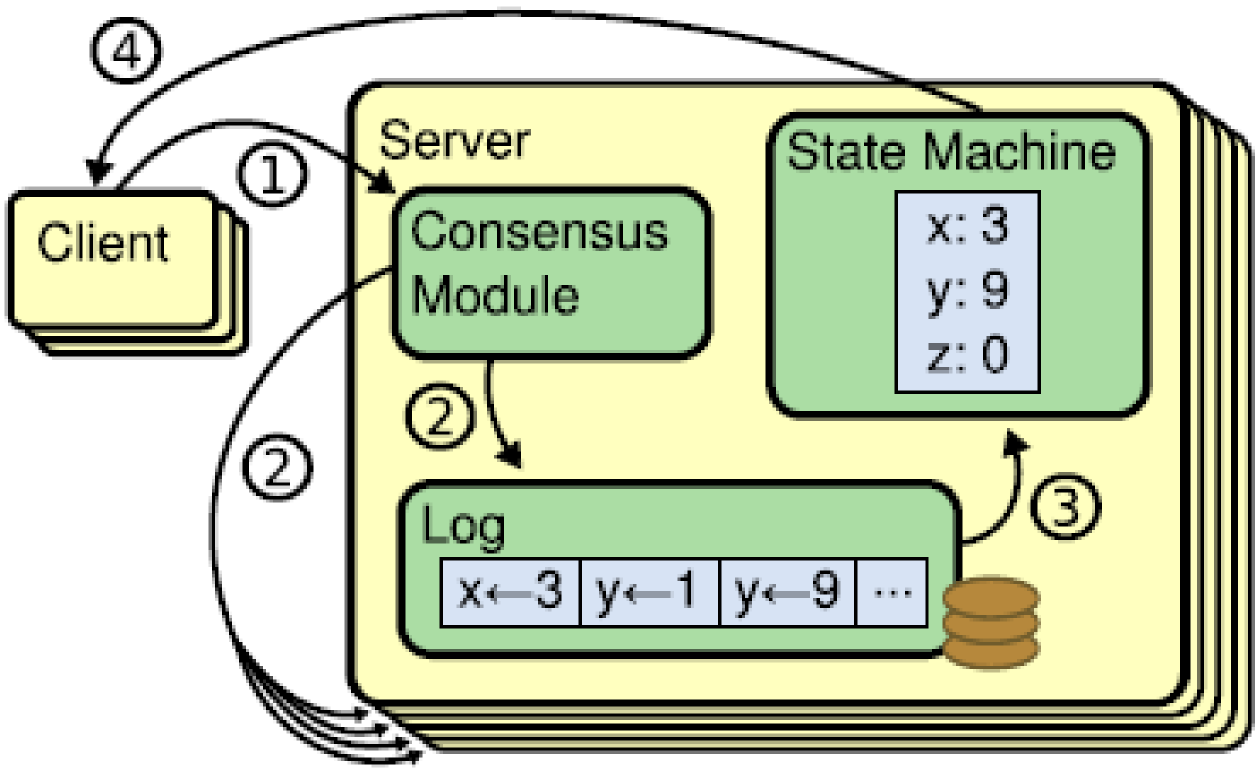
Nikita Borisov

Slide content borrowed from Diego Ongaro, John Ousterhout, and Alberto Montresor

The distributed log (I)

- Each server stores a log containing commands
- Consensus algorithm ensures that all logs contain the ***same commands*** in the same order
- State machines always execute commands ***in the log order***
  - They will remain consistent as long as command executions have ***deterministic results***

The distributed log (II)



The distributed log (III)

- Client sends a command to one of the servers
- Server adds the command to its log
- Server forwards the new log entry to the other servers
- Once a consensus has been reached, each server state machine process the command and sends it reply to the client

Raft consensus algorithm (I)

- Servers start by electing a ***leader***
  - Sole server habilitated to accept commands from clients
  - Will enter them in its log and forward them to other servers
  - Will tell them when it is safe to apply these log entries to their state machines

Raft consensus algorithm (II)

- Decomposes the problem into three fairly independent subproblems
  - ***Leader election:***
    How servers will pick a—***single***—leader
  - ***Log replication:***
    How the leader will accept log entries from clients, propagate them to the other servers and ensure their logs remain in a consistent state
  - ***Safety***

Raft leader election

- Election timeout
  - Used by nodes in Follower state
  - Reset at every AppendEntries (heartbeat) and RequestVote (election)
  - Randomized between 150 and 300 ms
- A timeout triggers transition to Candidate state
  - Increment current term
  - Vote for self
  - Send RequestVote messages to all other nodes
- When receiving RequestVote, vote for requestor if and only if not voted for anyone else in the requested term

Election Logic

Election timeout

```
currentTerm += 1
state = Candidate
votedFor = me
send(RequestVote(who=me,
      term=currentTerm))
```

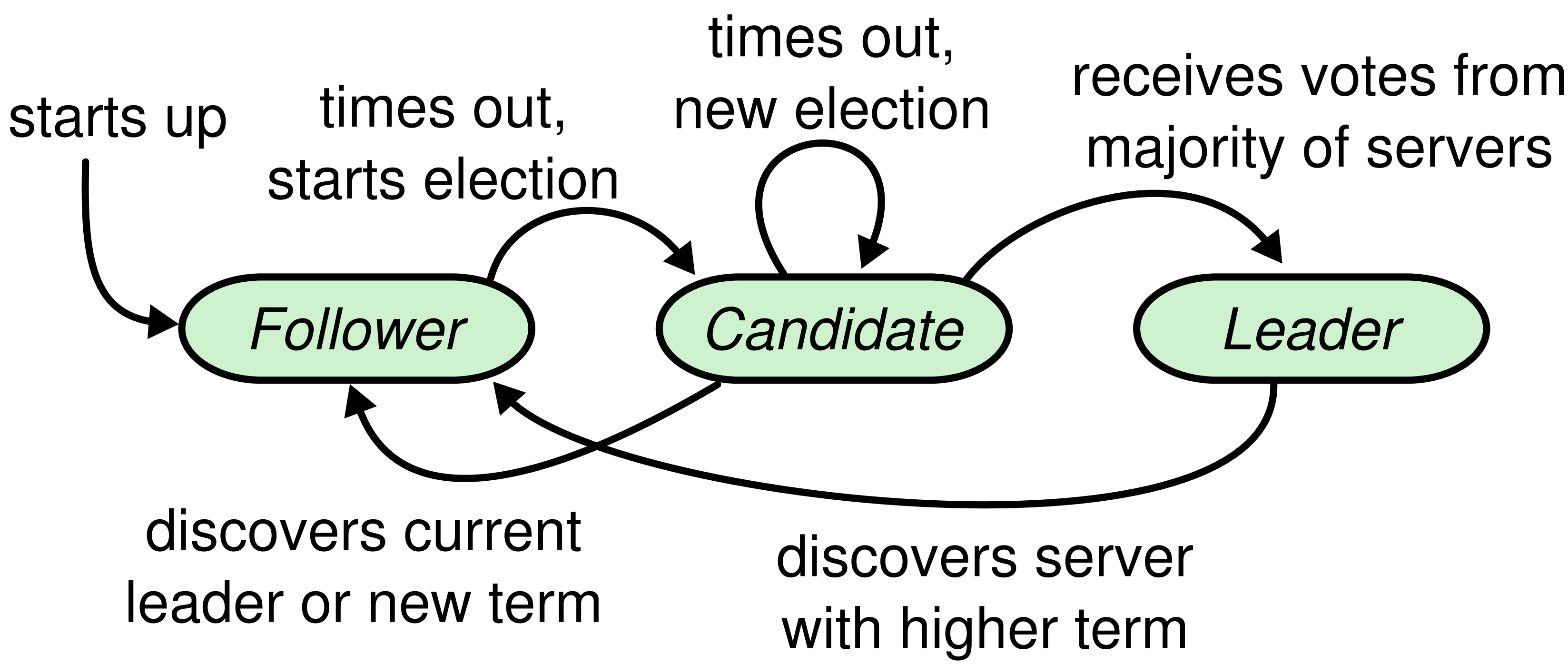
Receive RequestVote(*who*, *term*)

```
if currentTerm < term:
      currentTerm = term
      state = Follower
      votedFor = who
      reply(currentTerm, True)
      resetTimeout()
else:
      reply(currentTerm, False)
```

Candidate logic

1. Receive majority of votes
   - Transition to Leader state,
   - Send AppendEntries to all nodes
2. Receive AppendEntries from another leader
   - Transition to Follower state
3. Receive no vote with larger term #
   - Update term
   - Transition to Follower state
   - Wait for AppendEntries or timeout
4. Election timeout expires with no majority
   - Increment term, start new election

State machine



Raft properties

1. At most one leader elected per term

Why?

- Each node votes for at most one leader in a term
- (strict) majority needed for election

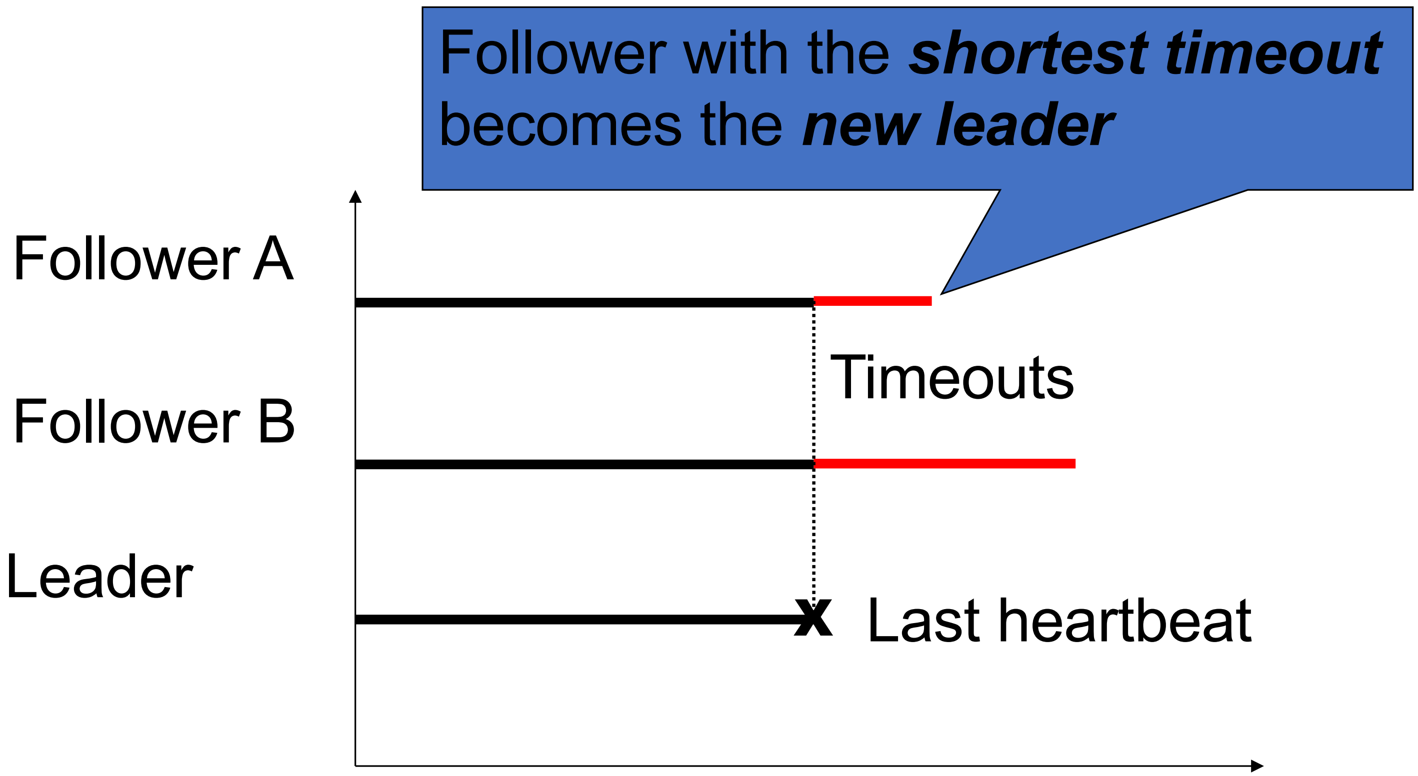
Leader election and FLP

- Is **totally correct** leader election possible in async systems?
- No! Leader election equivalent to consensus
- How is leader election in Raft not totally correct?
- Split elections

Avoiding split elections

- Raft uses randomized election timeouts
  - Chosen randomly from a fixed interval
- Increases the chances that a single follower will detect the loss of the leader before the others

Example

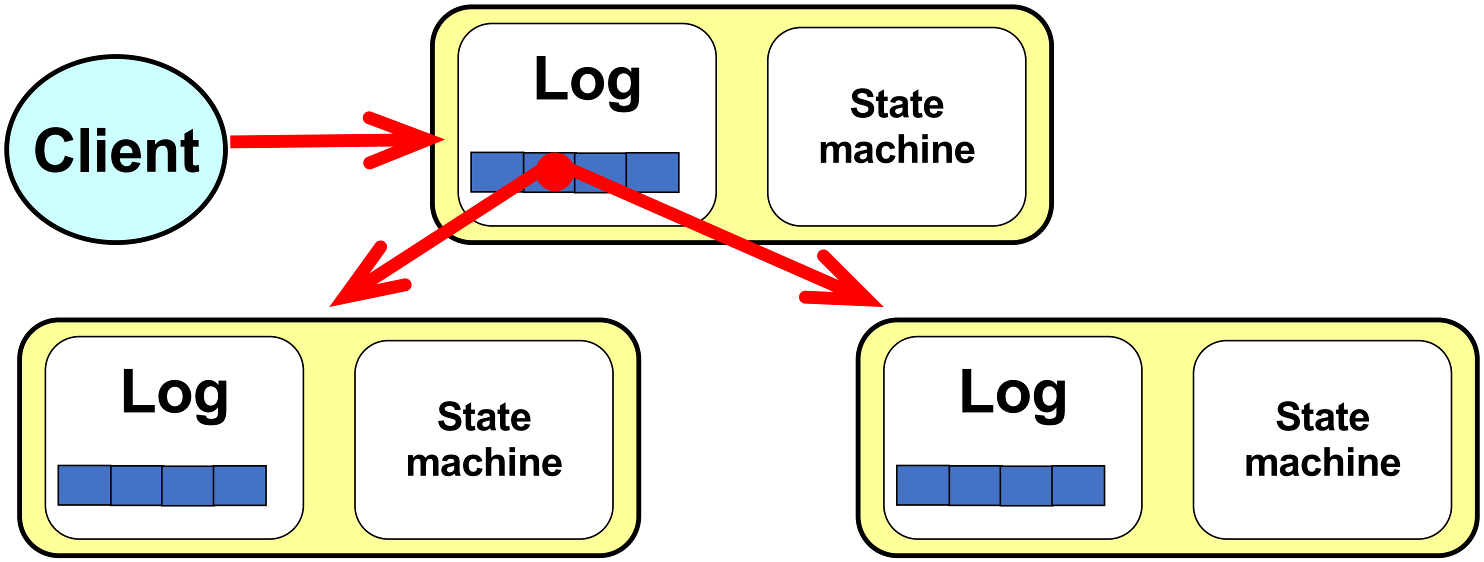


Follower with the ***shortest timeout*** becomes the ***new leader***
Follower A
Timeouts
Follower B
Leader
Last heartbeat

Log replication

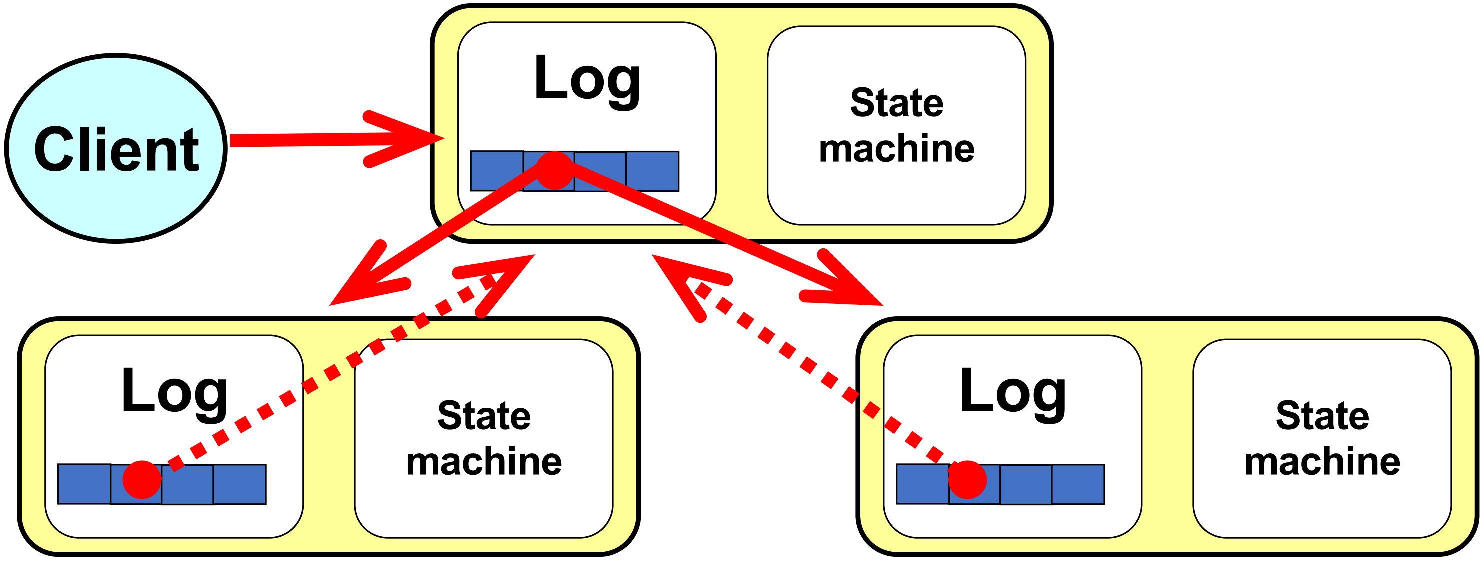
- Leaders
  - Accept client commands
  - Append them to their log (new entry)
  - Issue **AppendEntry** RPCs in parallel to all followers
  - Apply the entry to their state machine once it has been safely replicated
    - Entry is then ***committed***

A client sends a request



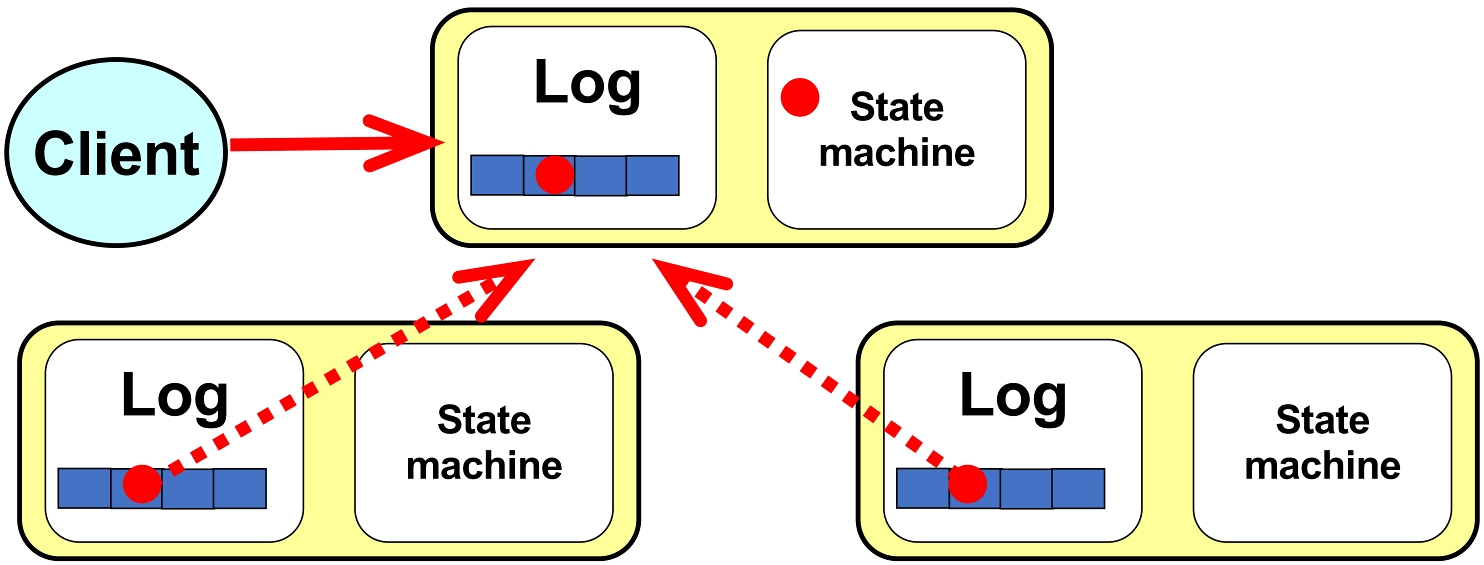
- Leader stores request on its log and forwards it to its followers

The followers receive the request



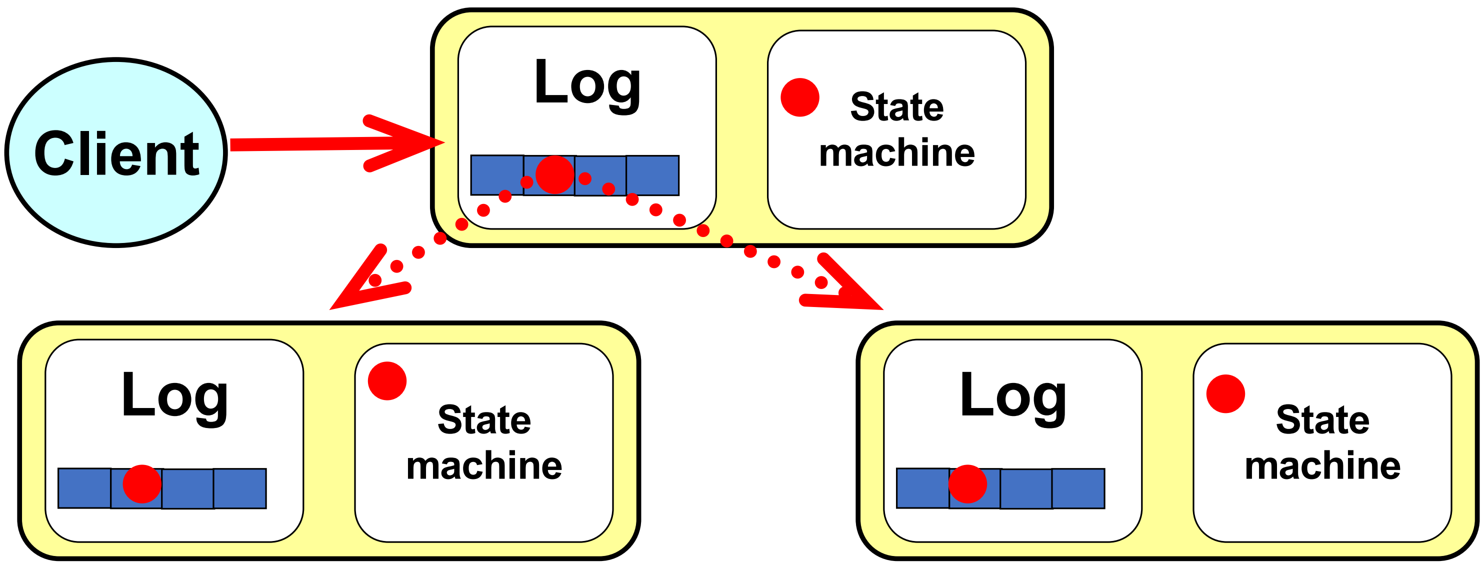
- Followers store the request on their logs and acknowledge its receipt

The leader tallies followers' ACKs



- Once it ascertains the request has been processed by a majority of the servers, it updates its state machine

The leader tallies followers' ACKs



- Leader's heartbeats convey the news to its followers: they update their state machines

AppendEntries processing

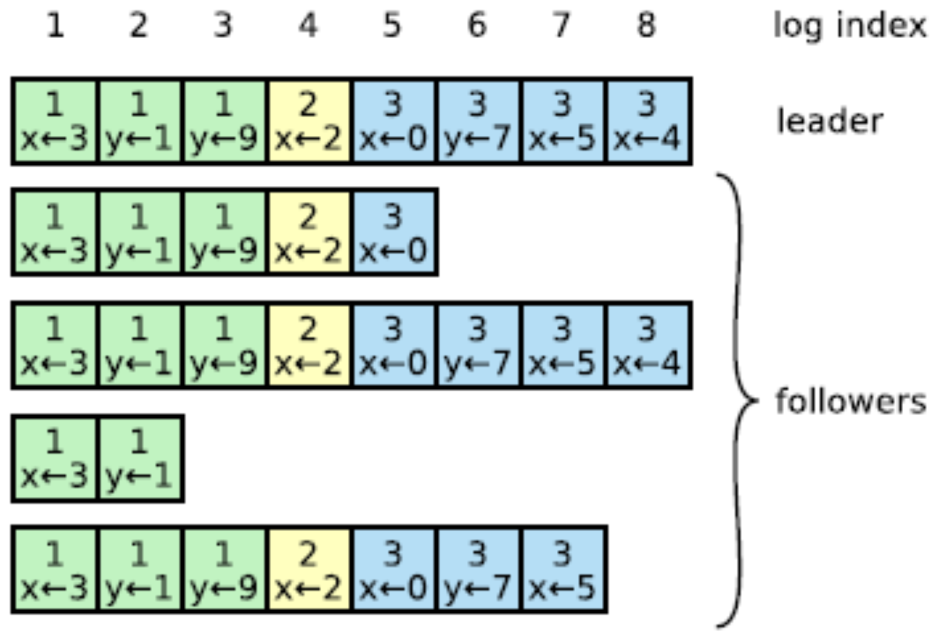
- AppendEntries contains
  - Leader's term
  - Leader's identity
  - Index of last previously broadcast entry (*prevLogIndex*)
  - Index of last committed entry (*leaderCommit*)
  - New entries
- If needed, update current term and set state to Follower
  - If current term > leader term, inform leader instead
- Check if *prevLogIndex* matches, and reconcile if it doesn't
  - Followers update their logs to match leader
  - Handles lost heartbeats, recovery from partition
- Update own commit index
- Add new entries
- Acknowledge

Raft properties

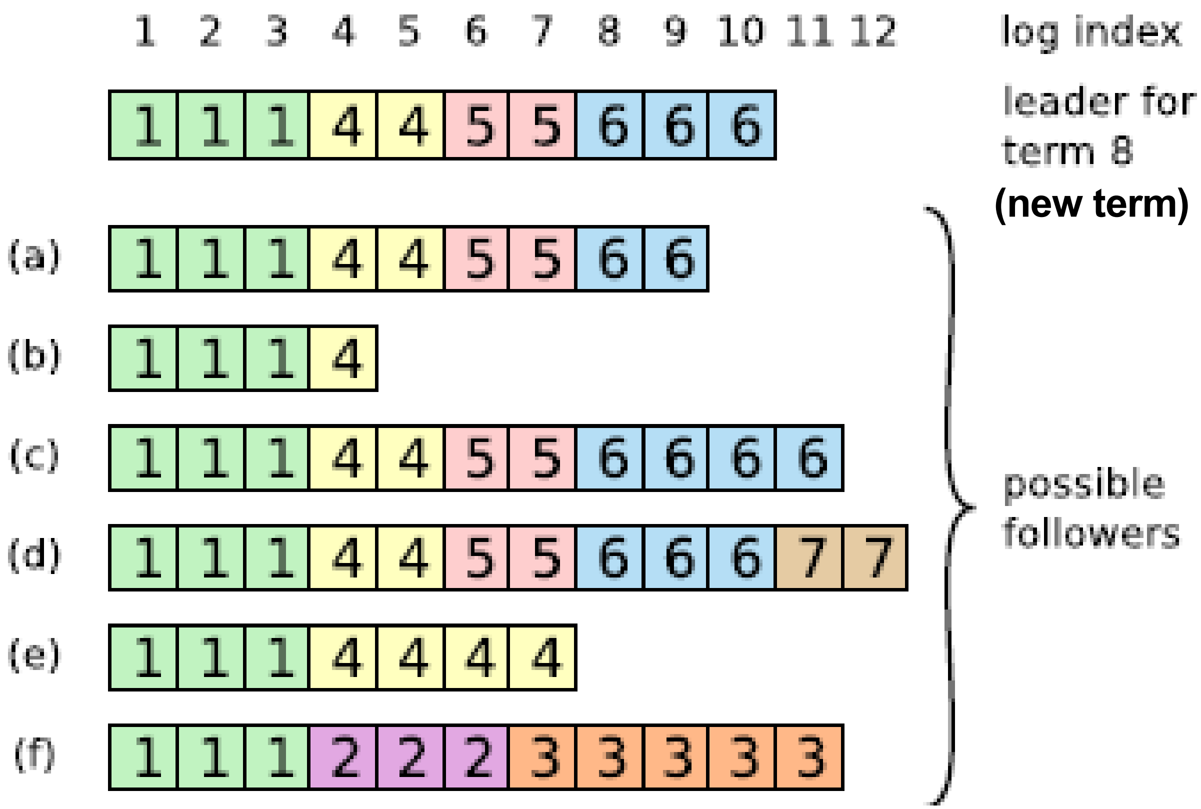
1. At most one leader elected per term
2. Log entries for a term are prefixes of the leader

   Why?
3. Committed log entries are replicated to majority of nodes

   Which entries might be committed?



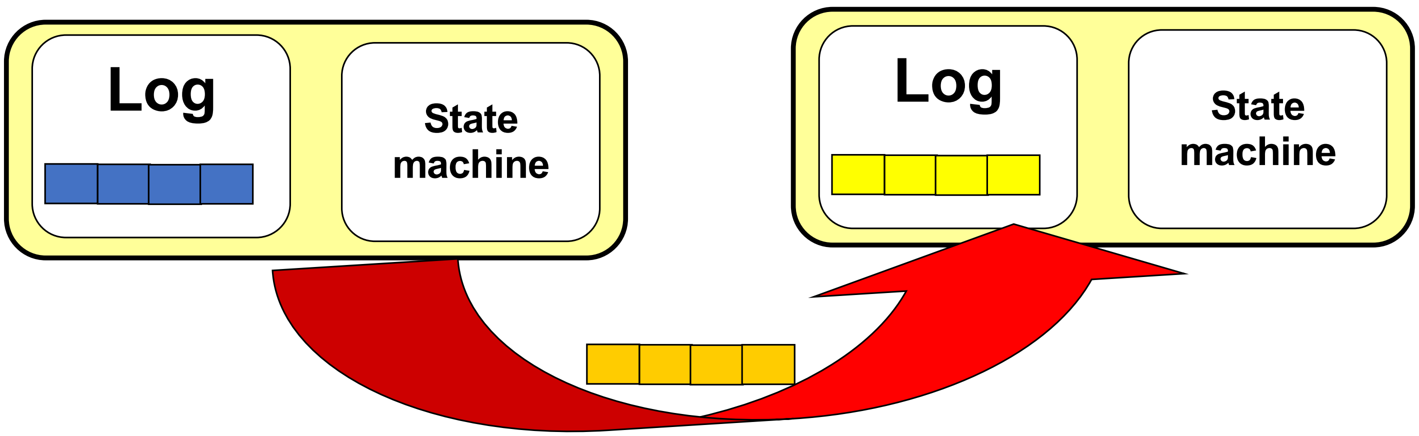
Log reconciliation



How could (f) happen?

- (f) leader for term 2
- Appends 3 [2] entries without committing
- Crashes, recovers, gets elected leader for term 3
- Appends 5 [3] entries without committing

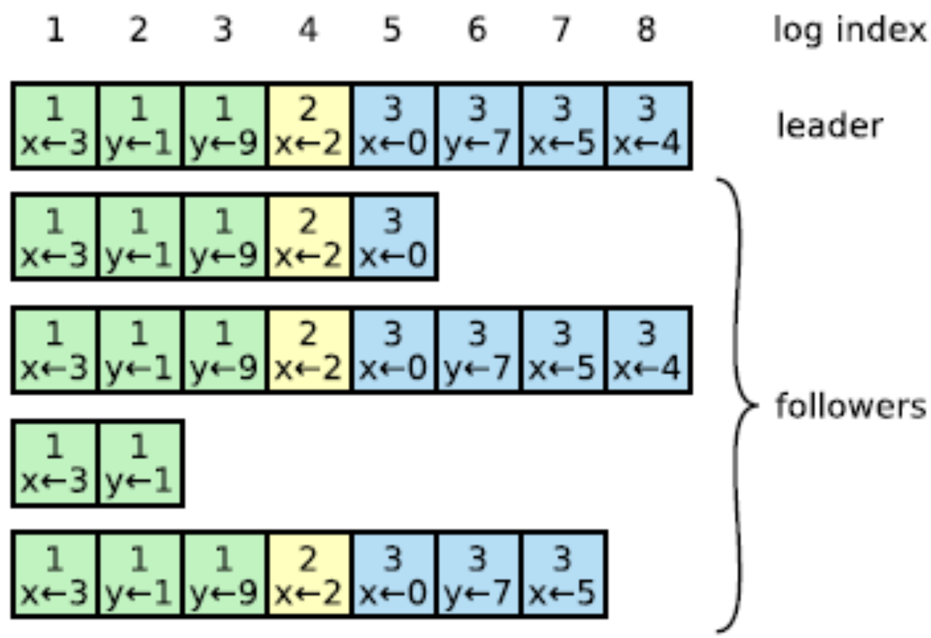
The new leader is in charge



- Newly elected candidate forces all its followers to duplicate in their logs the contents of its own log
- Conflicting log entries are **overwritten**

Raft properties

1. At most one leader elected per term
2. Log entries ~~for a term~~ **of any follower** are prefixes of the leader
3. Committed log entries are replicated to majority of nodes



Safety

- Two main issues
  - What if the log of a new leader did not contain all previously committed entries?
    - Must impose conditions on new leaders
  - How to commit entries from a previous term?
    - Must tune the commit mechanism

Election restriction (I)

- The log of any new leader ***must*** contain all previously committed entries
  - Candidates include in their ***RequestVote*** RPCs information about the state of their log
  - Before voting for a candidate, servers check that the log of the candidate is at least as up to date as their own log
  - Majority rule does the rest

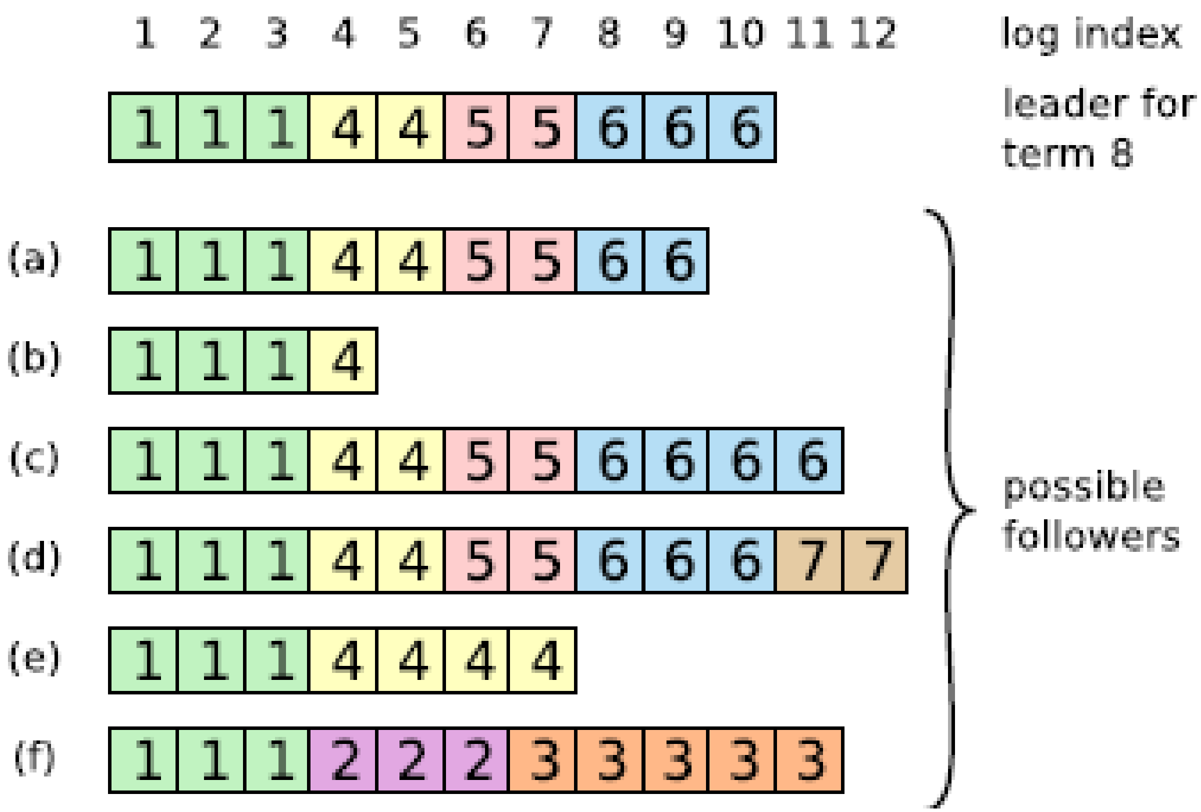
Election restriction

Receive RequestVote(*who, term, log*)

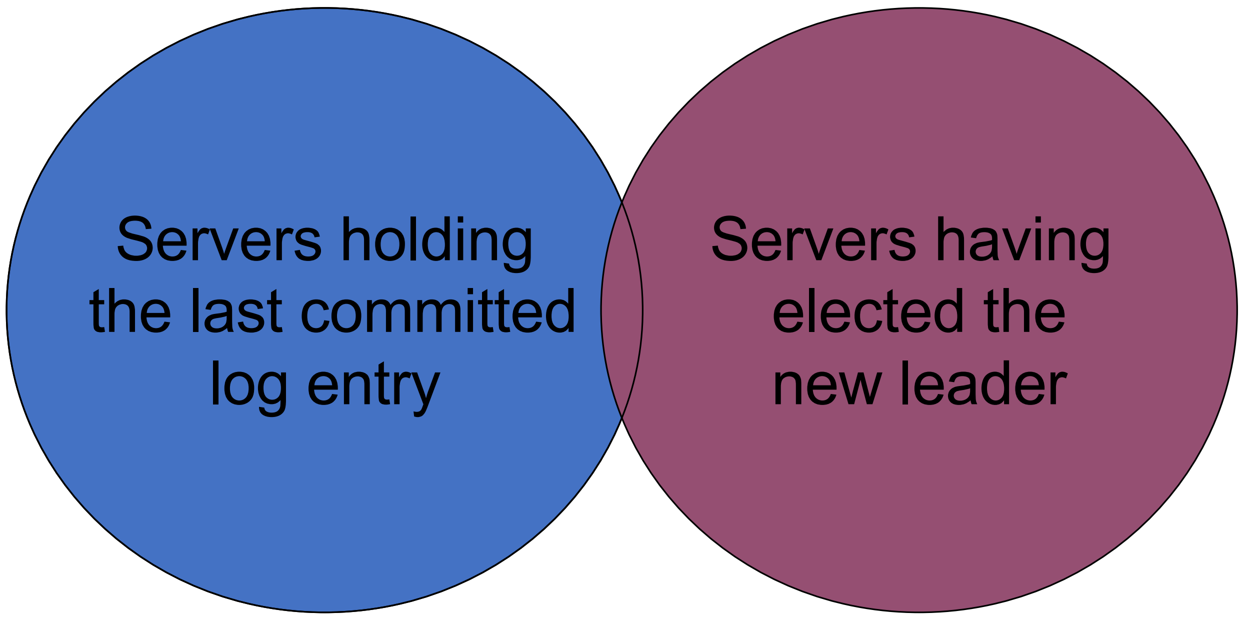
```
if currentTerm < term and \
      upToDate(log):
      currentTerm = term
      state = Follower
      votedFor = who
      reply(currentTerm, True)
      resetTimeout()
else:
      reply(currentTerm, False)
```

```
upToDate(log):
  logTerm = log[-1].term
  myTerm = self.log[-1].term
  if logTerm > myTerm:
    return True
  if logTerm == myTerm and \
       len(log) >= len(self.log):
    return True
  return False
```

Election Restriction



Election restriction (II)



Two majorities of the same cluster ***must*** intersect

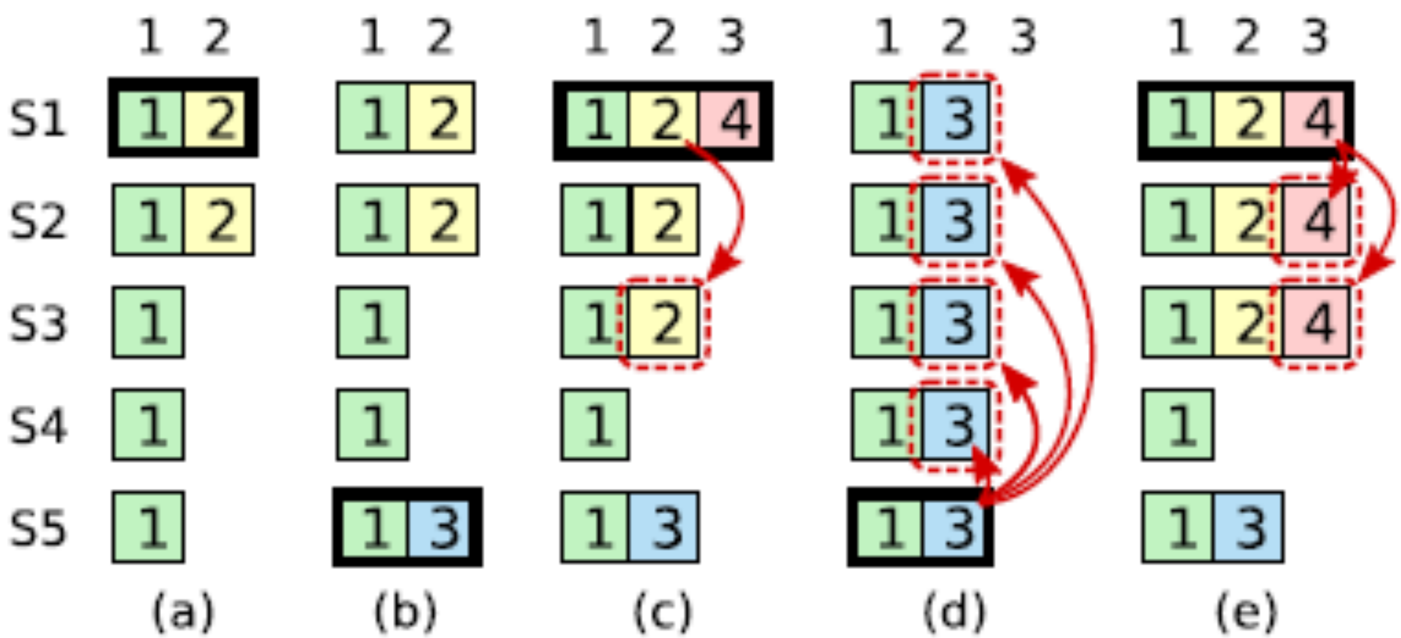
Raft properties

1. At most one leader elected per term
2. Log entries of any follower are prefixes of the leader
3. Committed log entries are replicated to majority of nodes
4. Current leader’s log contains all committed entries

Committing entries from a previous term

- A leader cannot immediately conclude that an entry from a previous term is committed even if it is stored on a majority of servers.
  - *See next figure*
- Leader should never commit log entries from previous terms by counting replicas
- Should only do it for entries from the current term
- Once it has been able to do that for one entry, all prior entries are committed indirectly

Committing entries from a previous term



Explanations

- In (a) S1 is leader and partially replicates the log entry at index 2.
- In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2.
- In (c) S5 crashes; S1 restarts, is elected leader, and continues replication.
  - Log entry from term 2 has been replicated on a majority of the servers, but it is not committed.

Explanations

- If S1 crashes as in (d), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3.
- However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win an election).
- At this point all preceding entries in the log are committed as well.

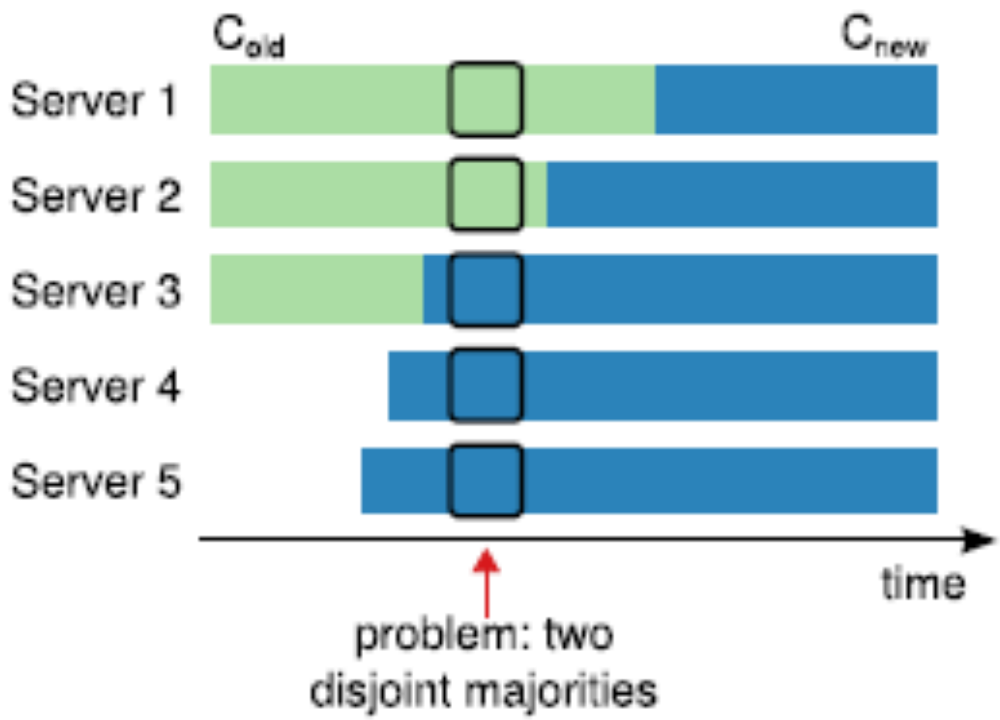
Cluster membership changes

- Not possible to do an atomic switch
  - Changing the membership of all servers at one
- Will use a two-phase approach:
  - Switch first to a transitional ***joint consensus*** configuration
  - Once the joint consensus has been committed, transition to the new configuration

The joint consensus configuration

- Log entries are transmitted to all servers, old and new
- Any server can act as leader
- Agreements for entry commitment and elections requires majorities from both old and new configurations
- Cluster configurations are stored and replicated in special log entries

The joint consensus configuration



Implementations

- Two thousand lines of C++ code, not including tests, comments, or blank lines.
- About 25 independent third-party open source implementations in various stages of development
- Some commercial implementations