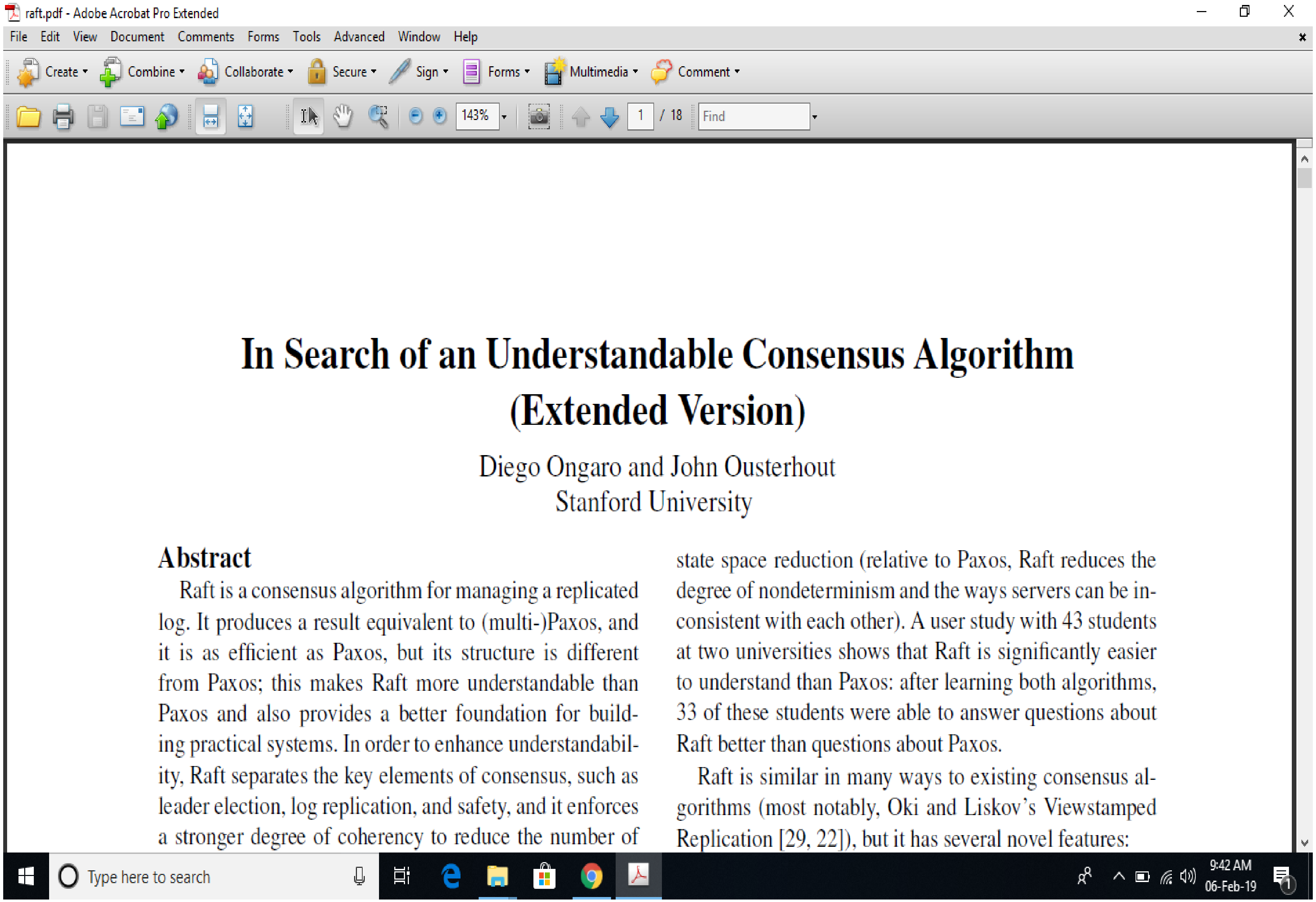
RAFT

Raft is a consensus algorithm that is designed to be easy to understand.

Goals

- Design for understandability
- Strong leader
- Practical to implement


raft.pdf - Adobe Acrobat Pro Extended
File Edit View Document Comments Forms Tools Advanced Window Help
Create Combine Collaborate Secure Sign Forms Multimedia Comment
143%
1 / 18
Find
Type here to search
9:42 AM
06-Feb-19

In Search of an Understandable Consensus Algorithm (Extended Version)

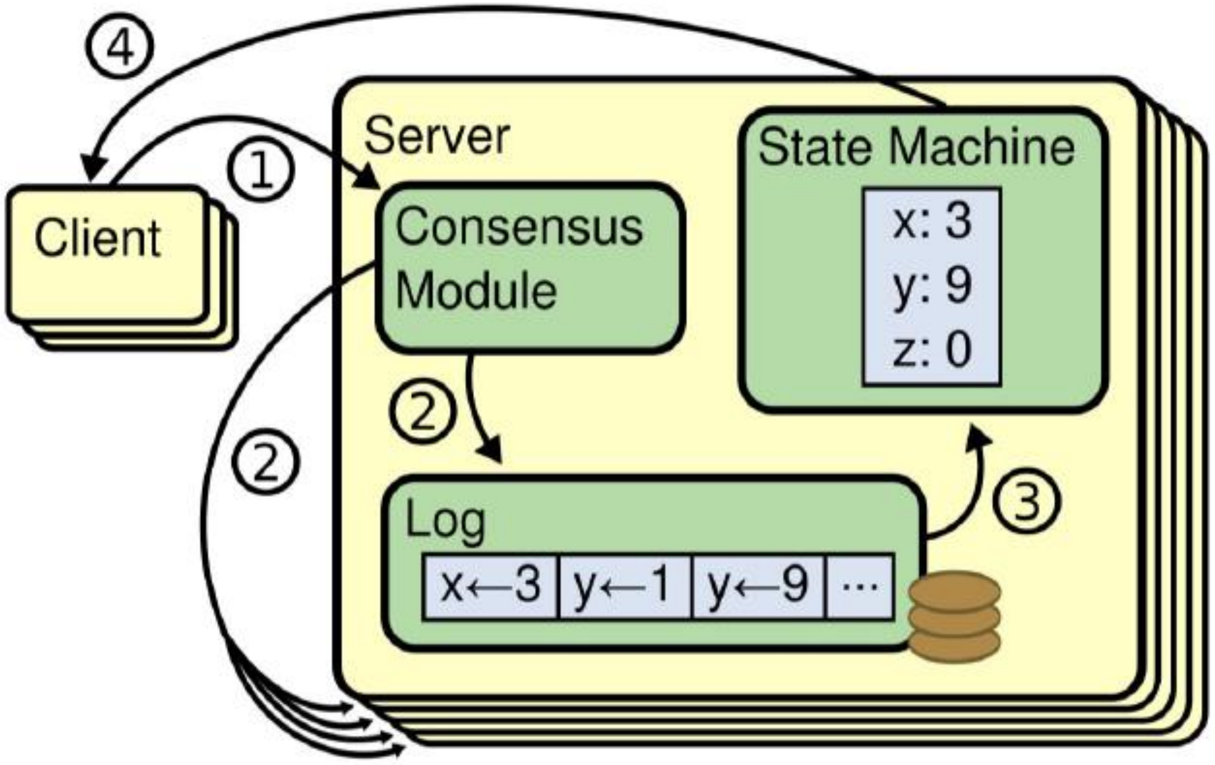
Diego Ongaro and John Ousterhout
Stanford University

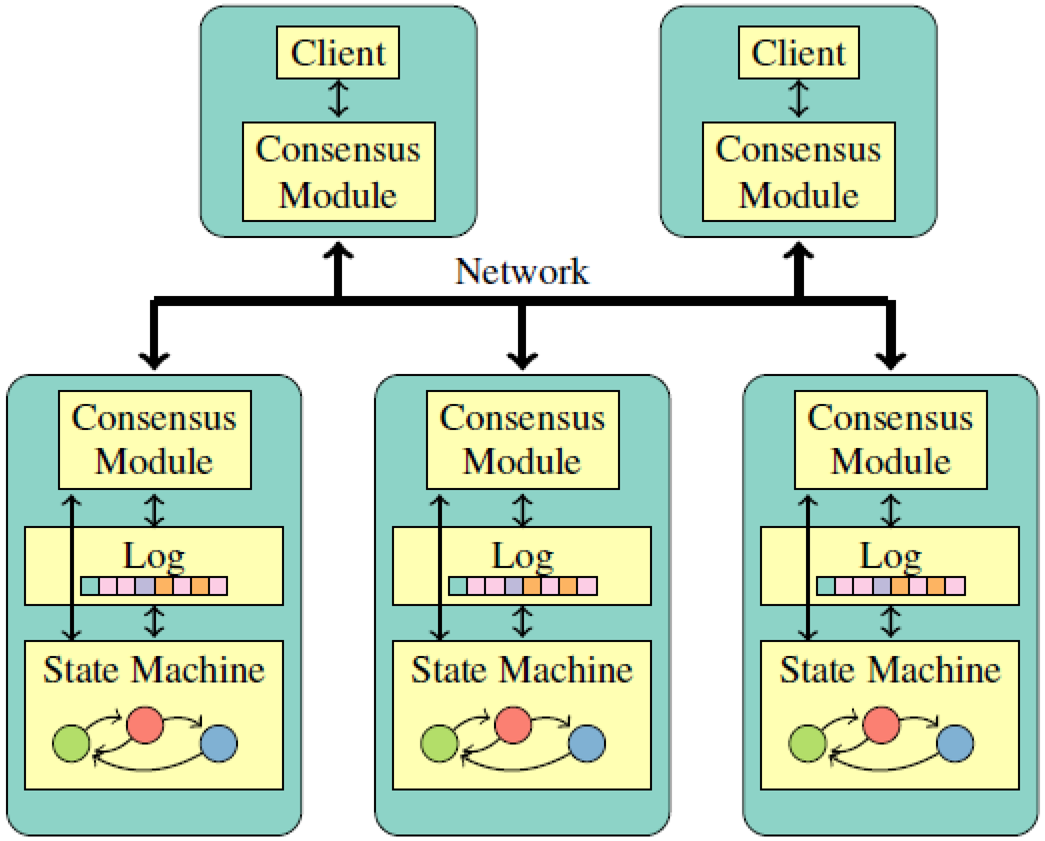
Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

Motivation: Replicated Log




Client
Consensus Module
Client
Consensus Module
Network
Consensus Module
Log
State Machine
Consensus Module
Log
State Machine
Consensus Module
Log
State Machine

- Replicated log $\Rightarrow$ State-machine replication
    - Each server stores a log containing a sequence of state-machine commands.
    - All servers execute the same commands in the same order.
    - Once one of the state machine finishes execution, result is returned to client.
- Consensus module ensures proper log replication
    - Receives commands from clients and adds them to its log
    - Communicates with consensus modules on other servers such that every log eventually contains same commands in same order
- *Failure model:* Fail-stop (may recover and rejoin), delayed/lost messages

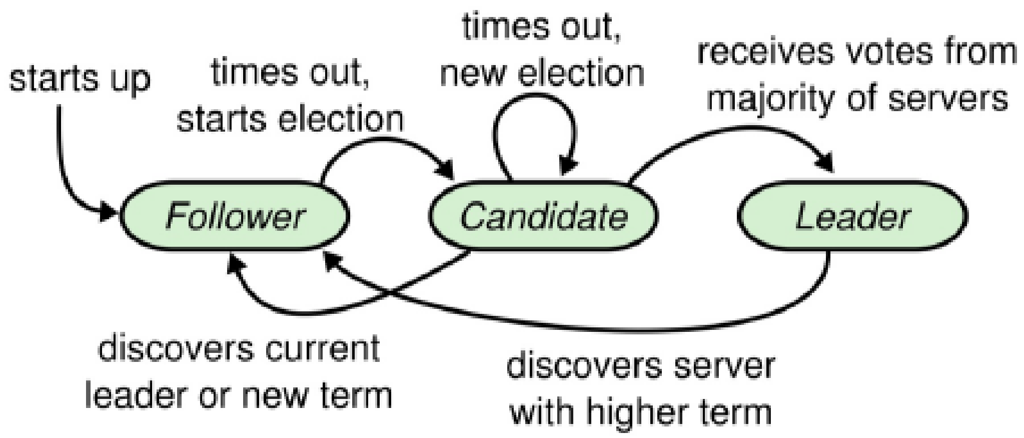
Consensus algorithms

- **Safety**: Never return in incorrect result despite network delays, partitions, duplication, loss, reordering of messages
- **Availability**: Majority of servers should be sufficient
    - Typical setup: 5 servers where 2 servers can fail
- **Performance**: (Minority of) slow servers should not impact the overall system performance

Approaches to consensus

- *Leader-less (symmetric)*
  - All servers are operating equally
  - Clients can contact any server
- *Leader-based (asymmetric)*
  - One server (called leader) is in charge
  - Other server follow the leader's decisions
  - Clients interact with the leader, i.e. all requests are forwarded to the leader
  - If leader crashes, a new leader needs to be (s)elected

Server Roles


starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

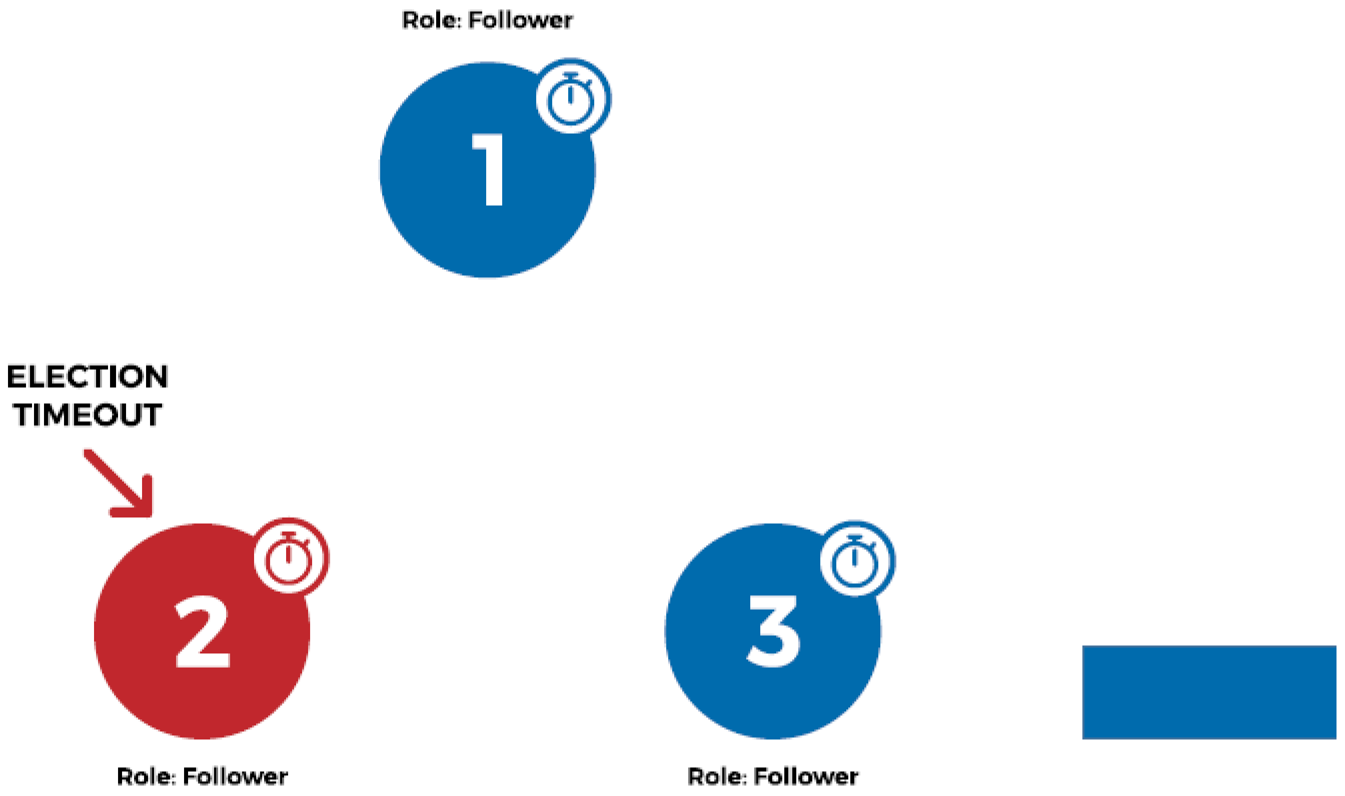
At any time, a server is either

- **Leader**: Handles client interactions and log replication
- **Follower**: Passively follows the orders of the leader
- **Candidate**: Aspirant in leader election

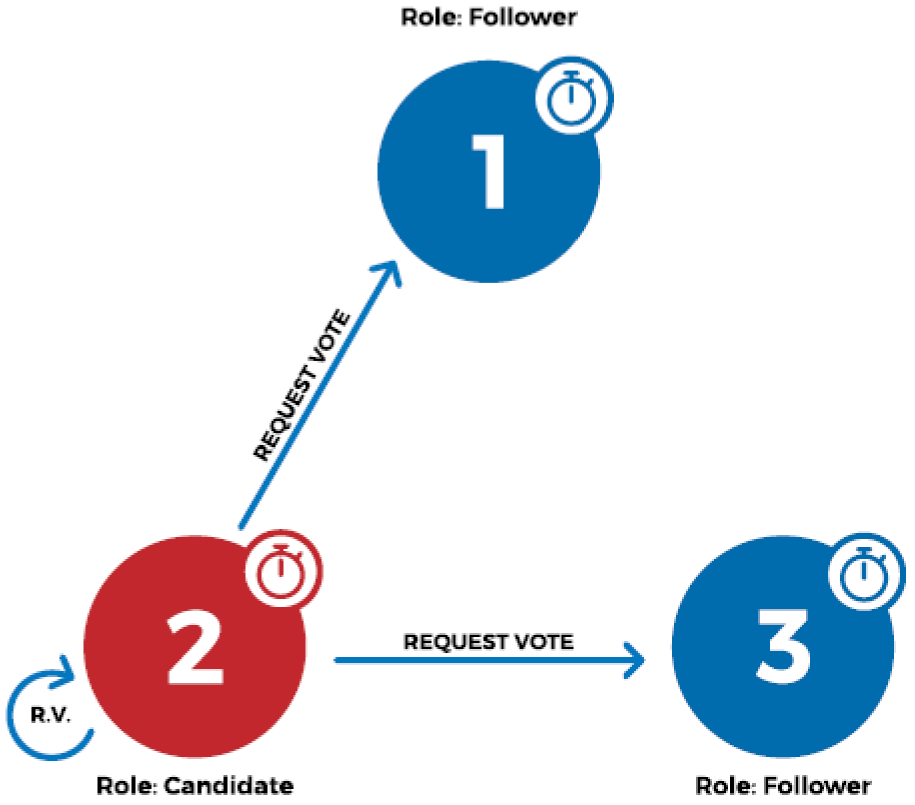
During normal operation

1 leader, N-1 followers

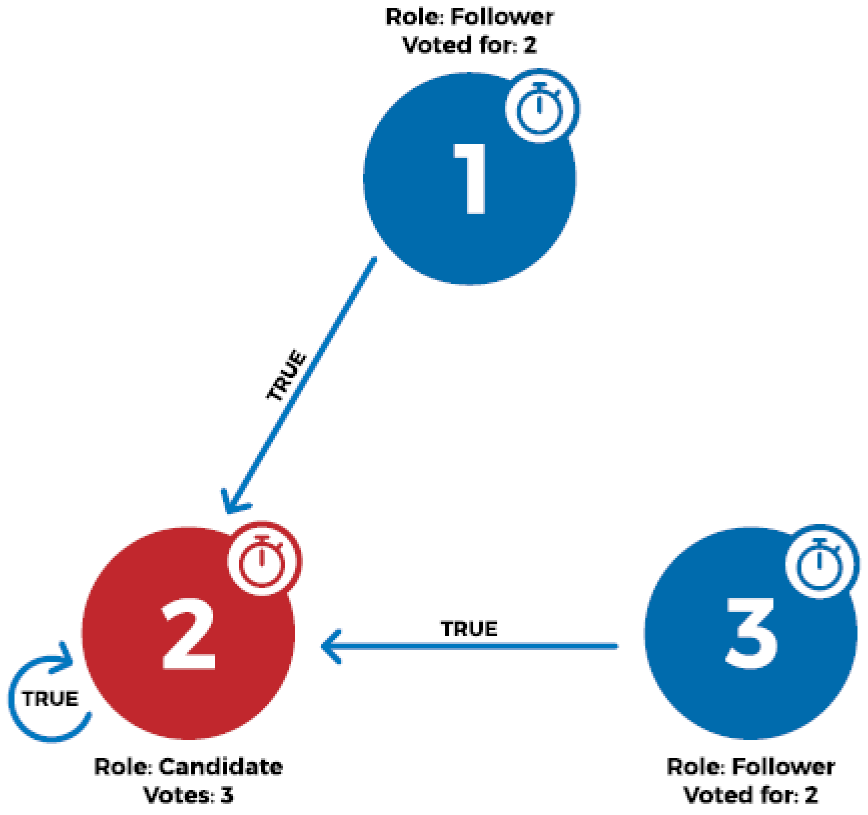
Leader election



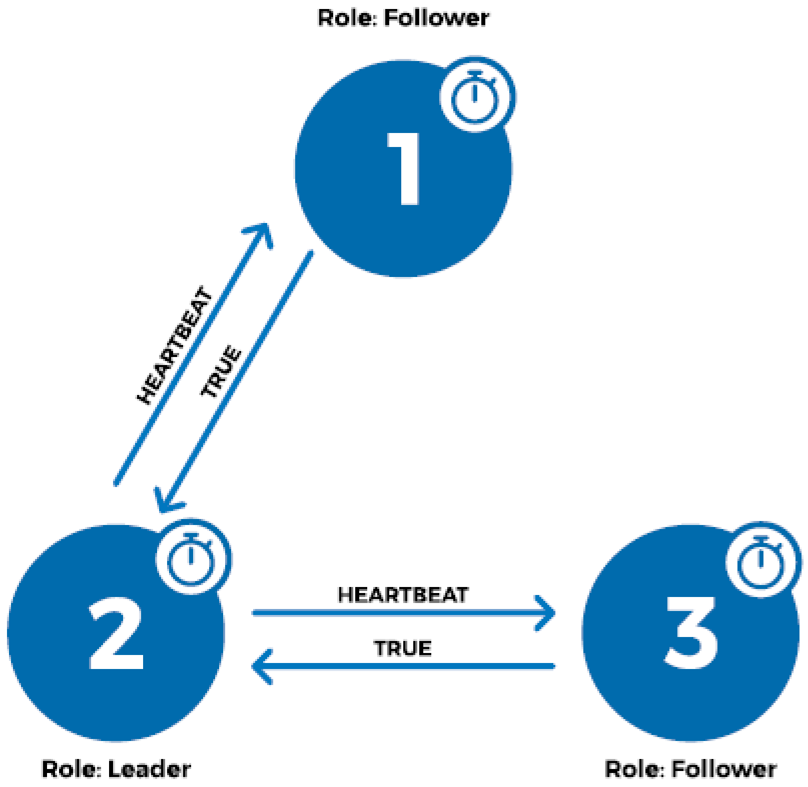
Leader election



Leader election



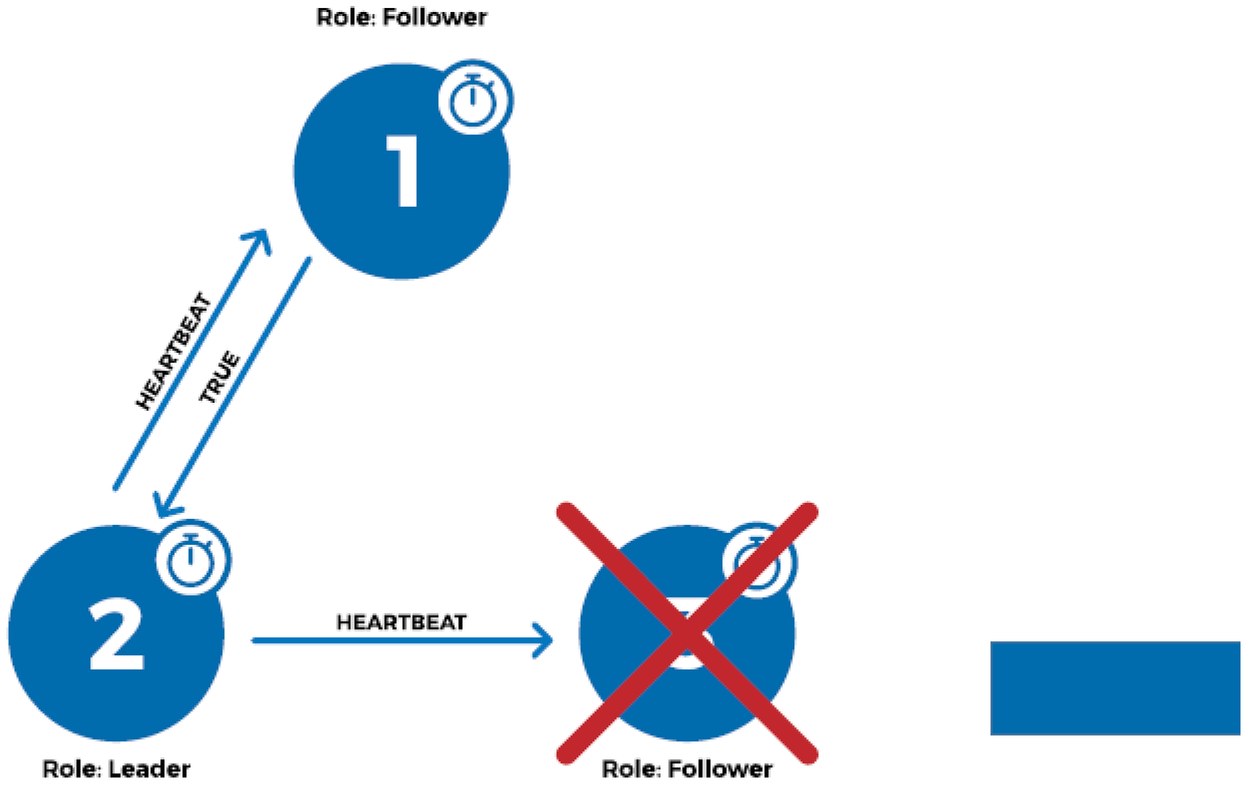
Leader election



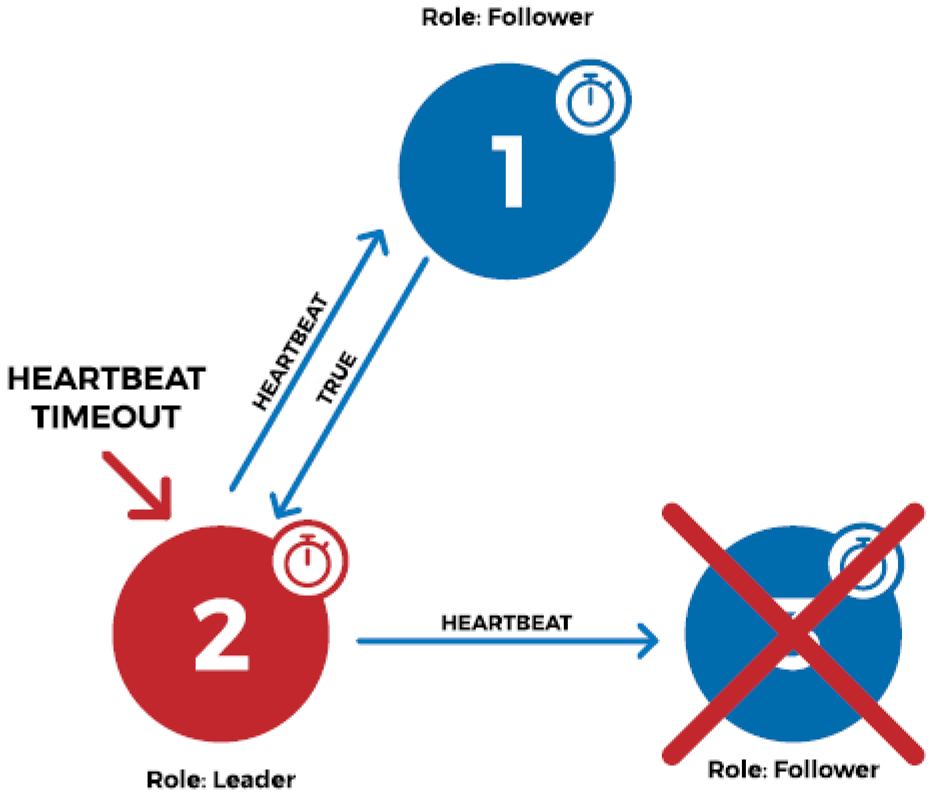
Raft

Node failure

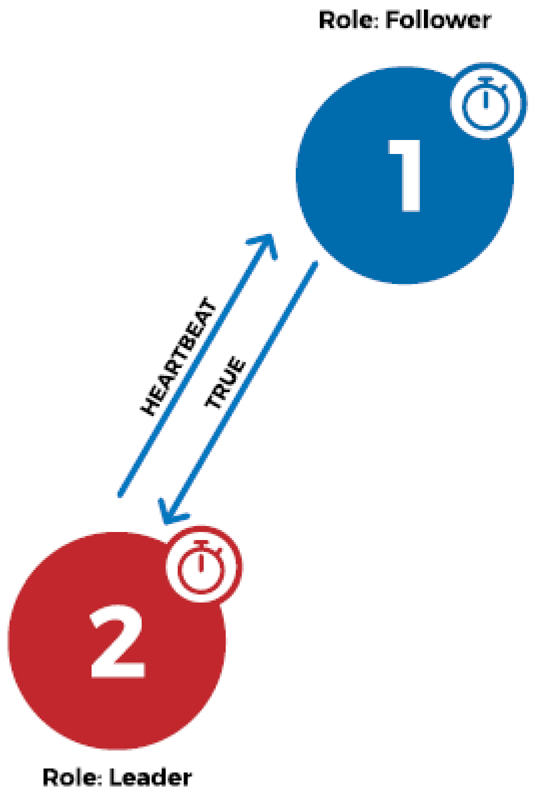
Dead node detection



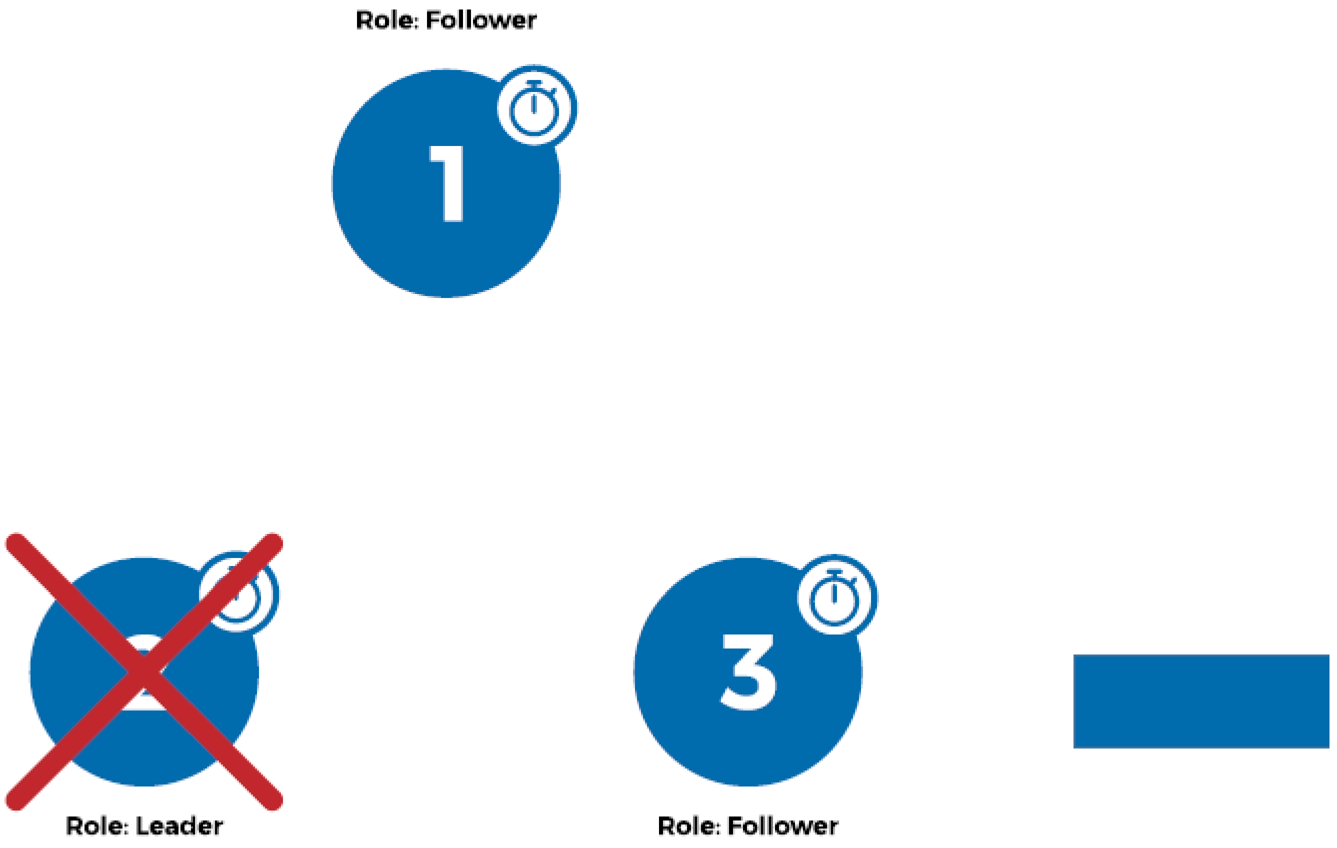
Dead node detection



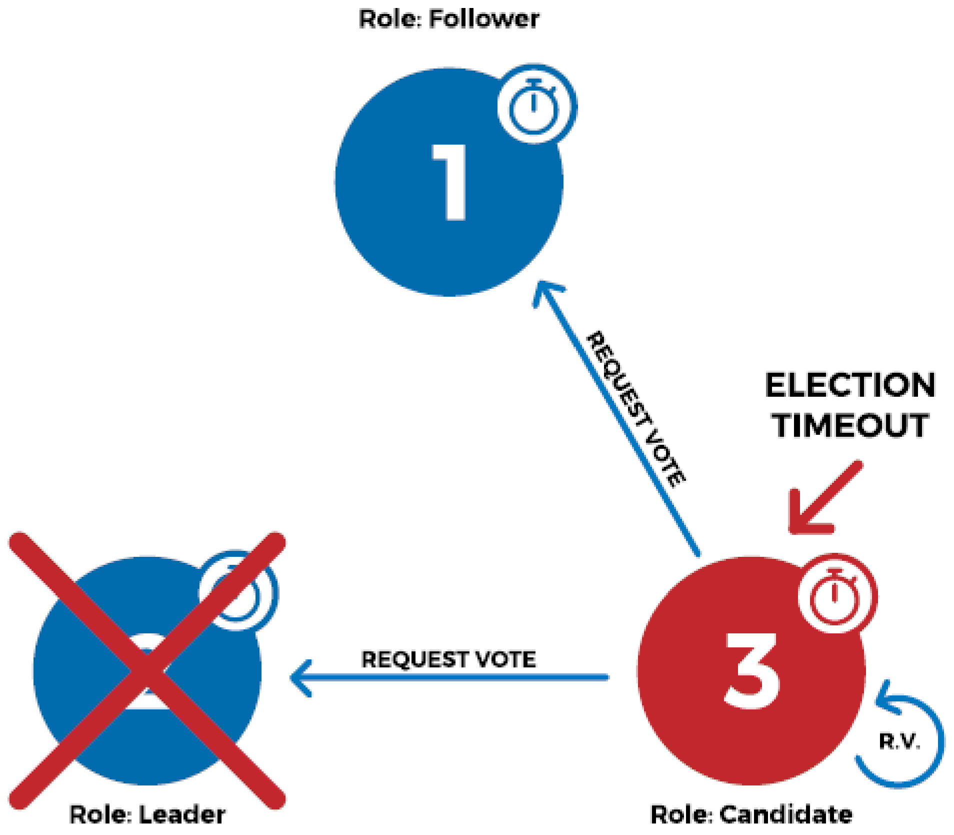
Dead node detection



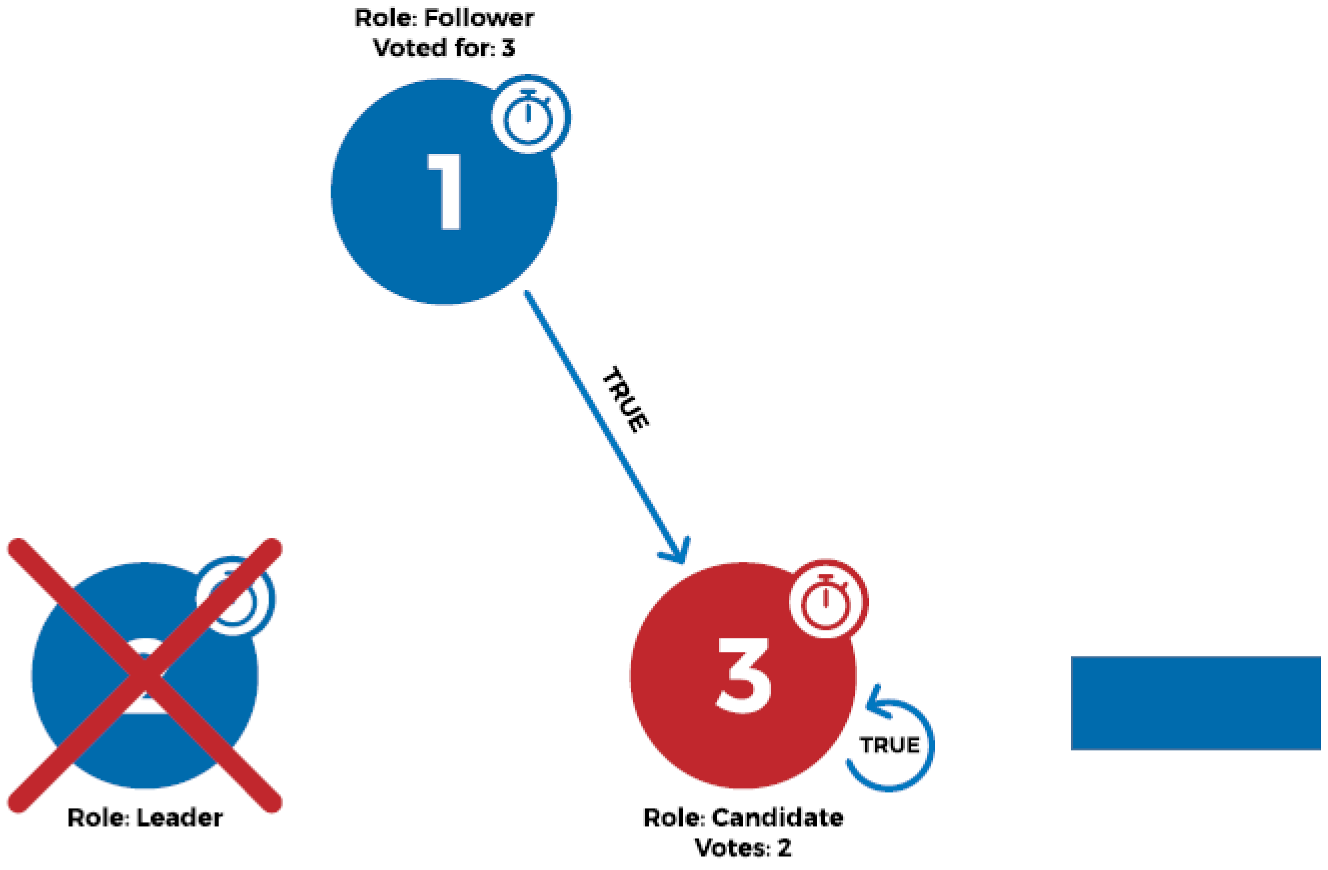
Leader node failure



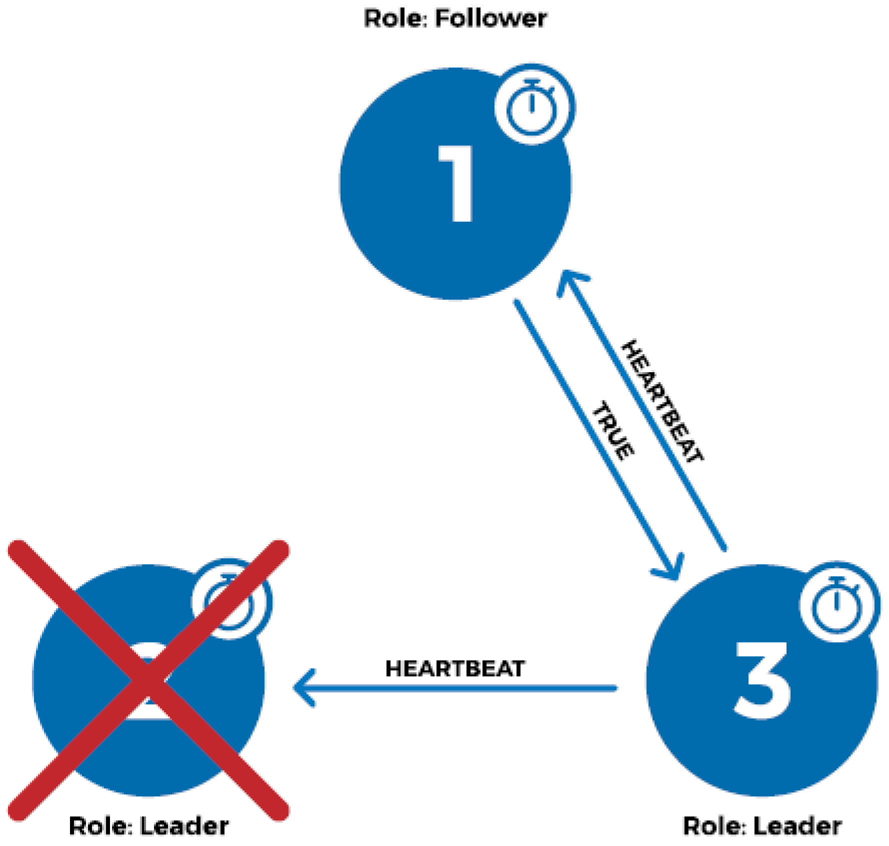
Leader node failure



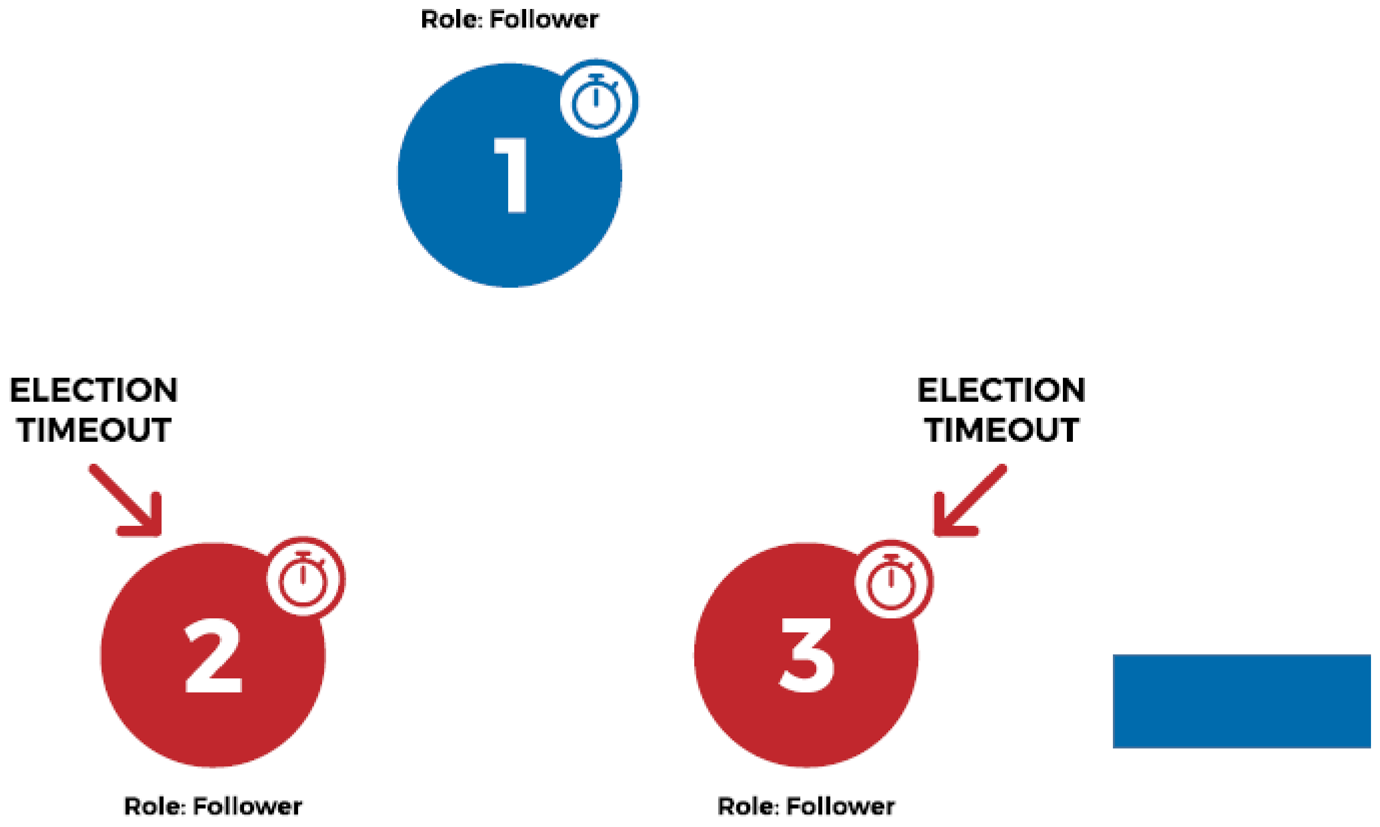
Leader node failure



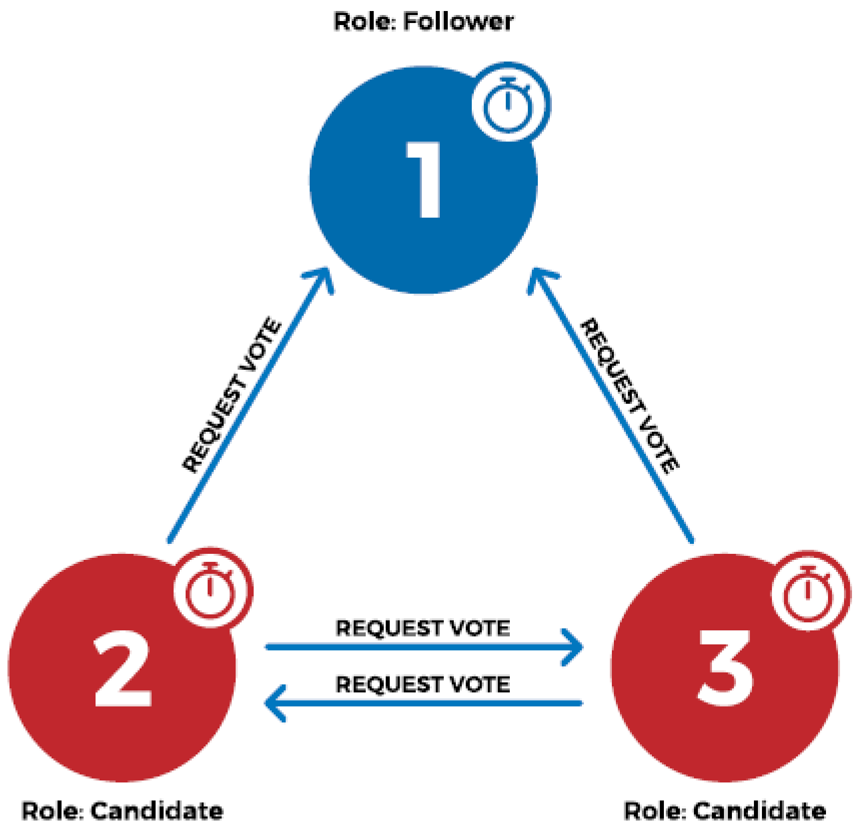
Leader node failure



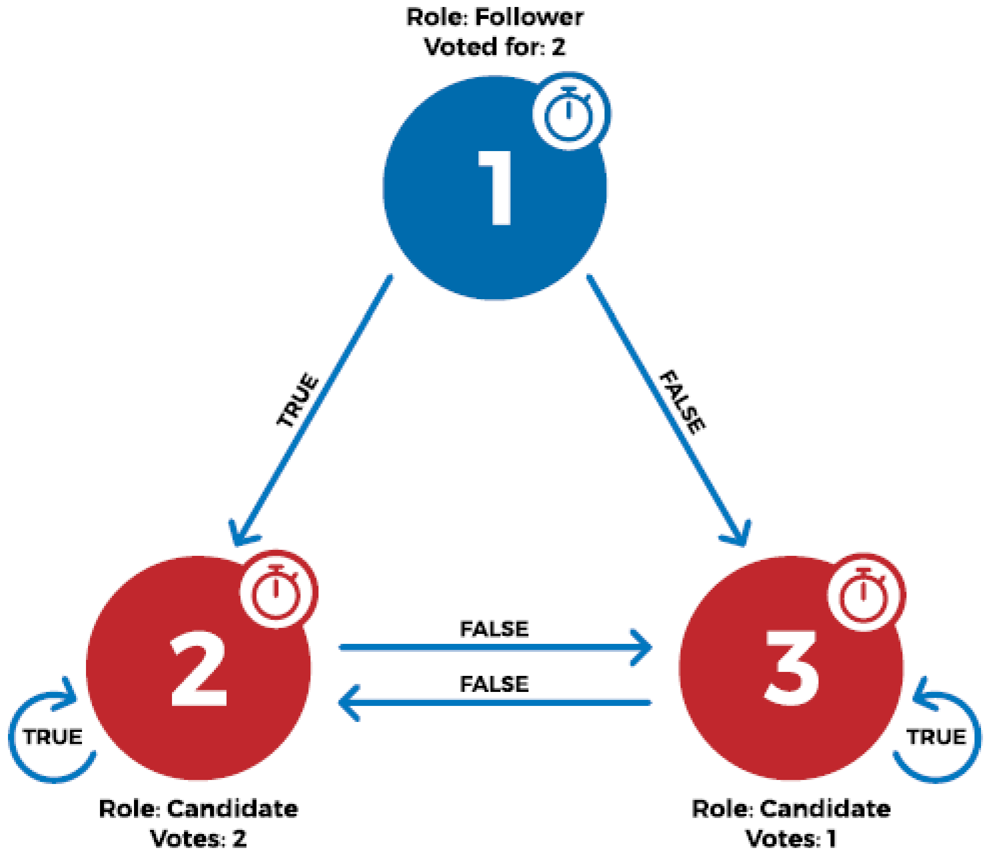
Election race condition



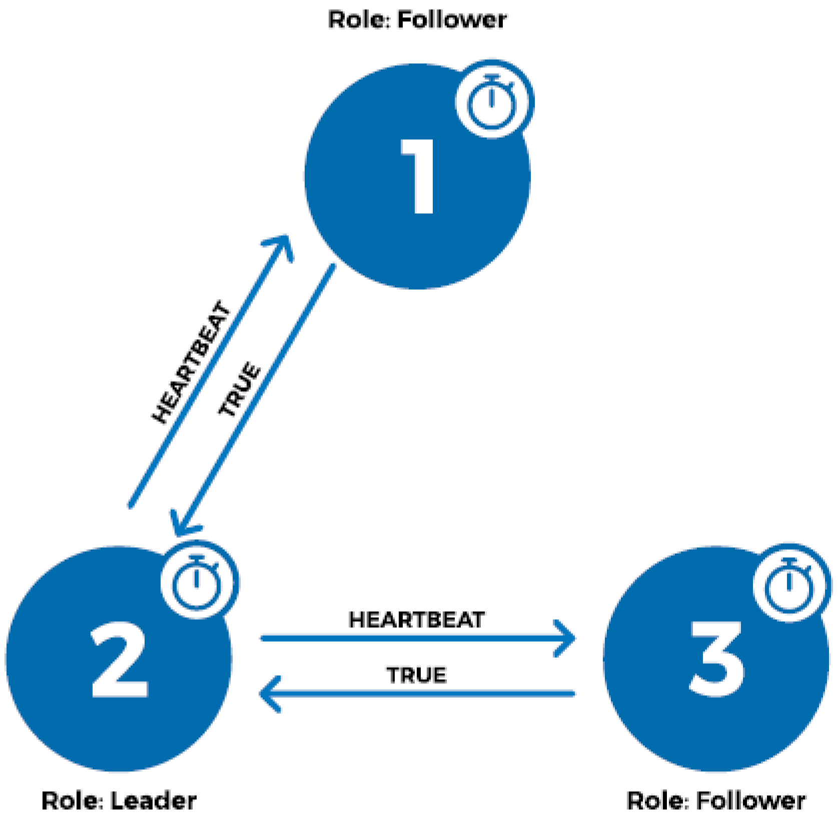
Election race condition



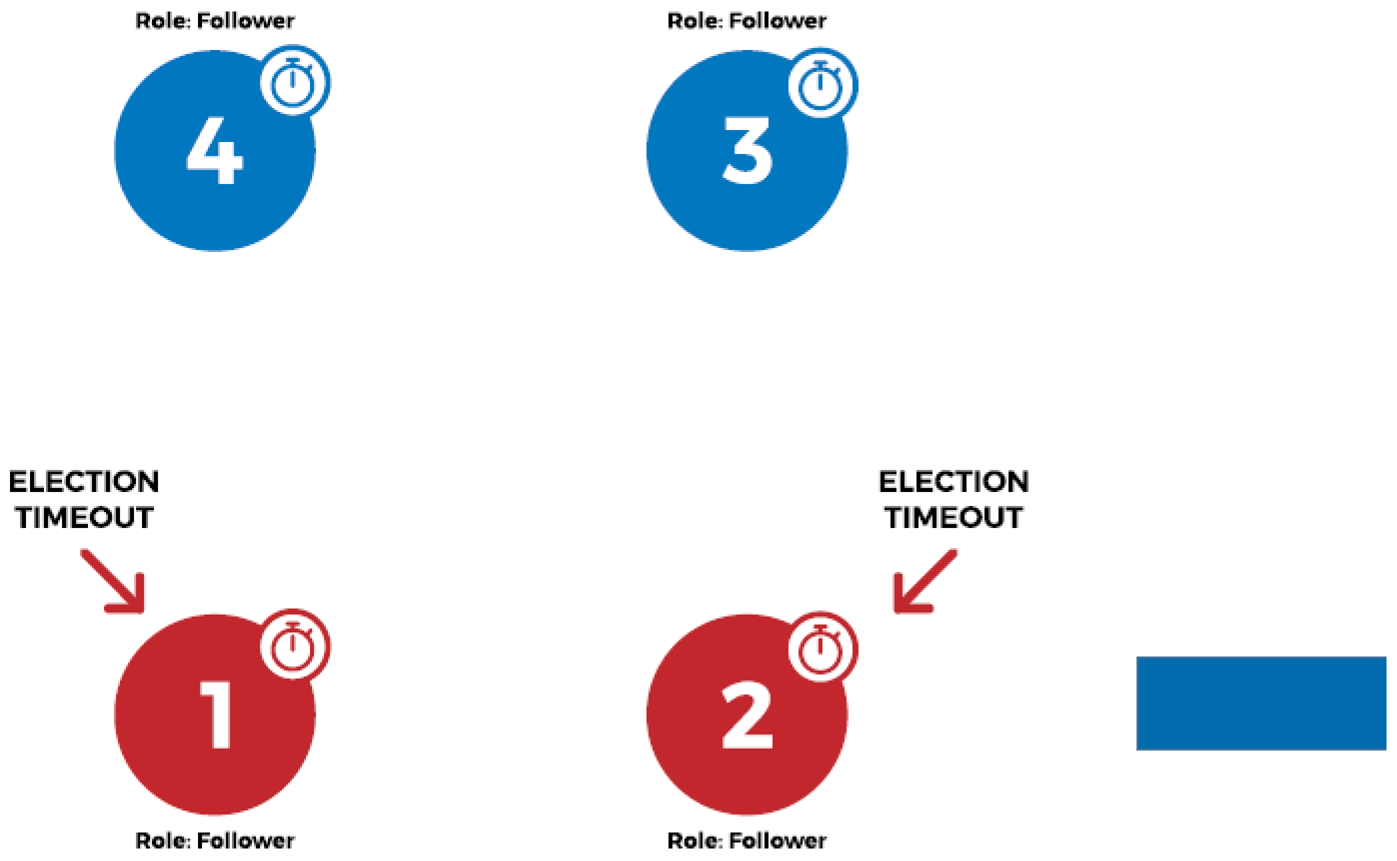
Election race condition



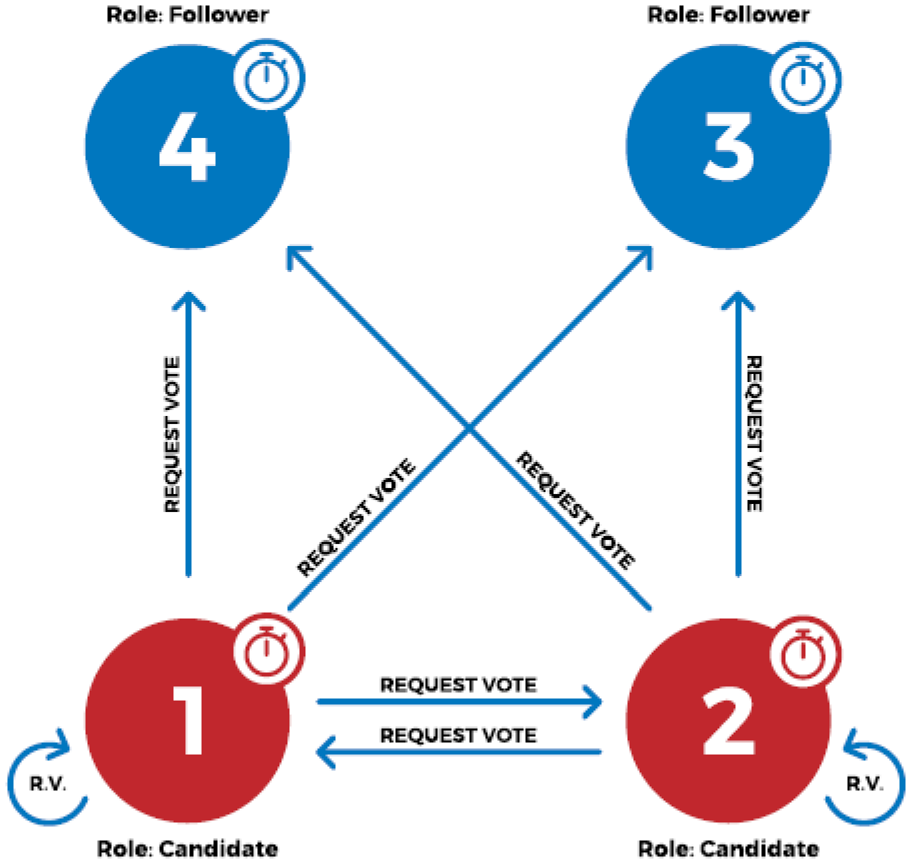
Election race condition



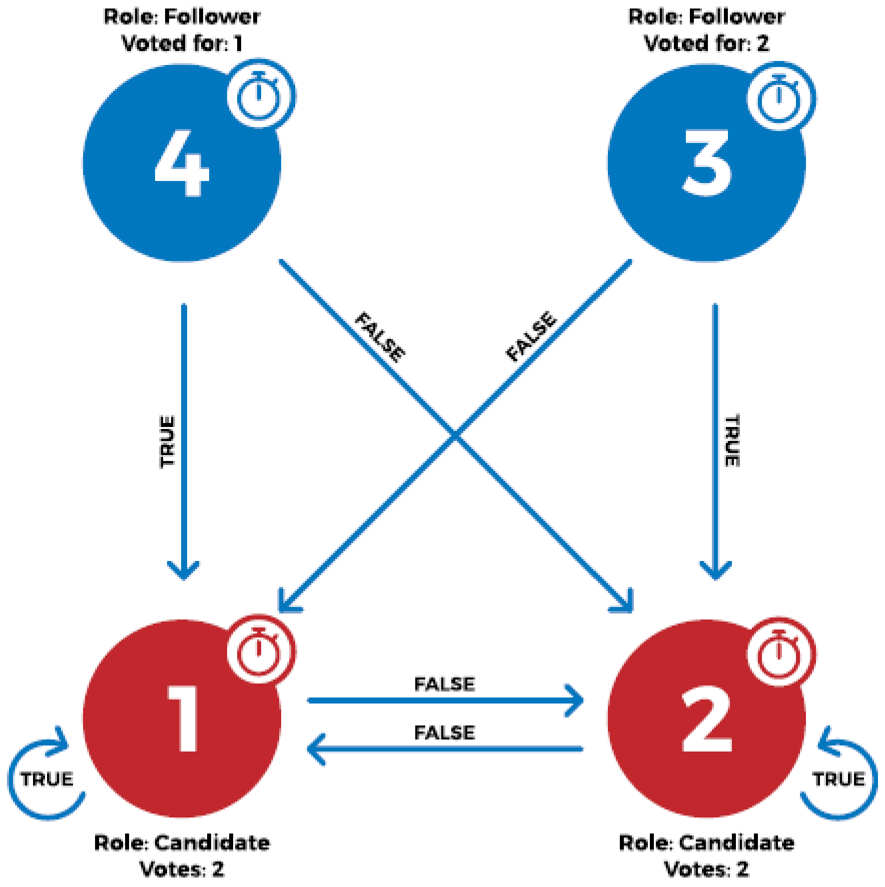
Split votes


Role: Follower
4
Role: Follower
3
ELECTION TIMEOUT
1
Role: Follower
ELECTION TIMEOUT
2
Role: Follower

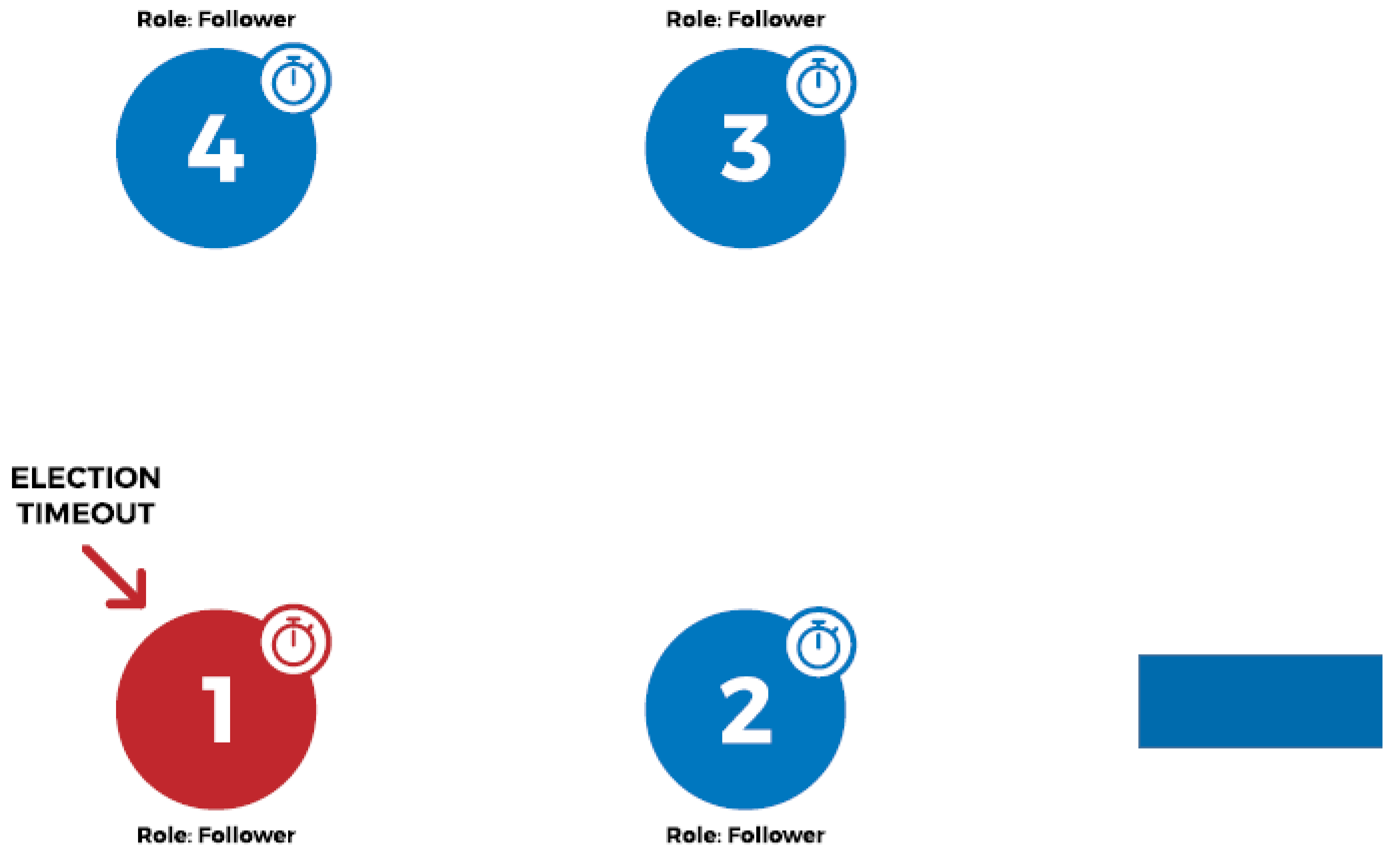
Split votes


Role: Follower
4
Role: Follower
3
REQUEST VOTE
REQUEST VOTE
REQUEST VOTE
REQUEST VOTE
1
REQUEST VOTE
REQUEST VOTE
2
R.V.
R.V.
Role: Candidate
Role: Candidate

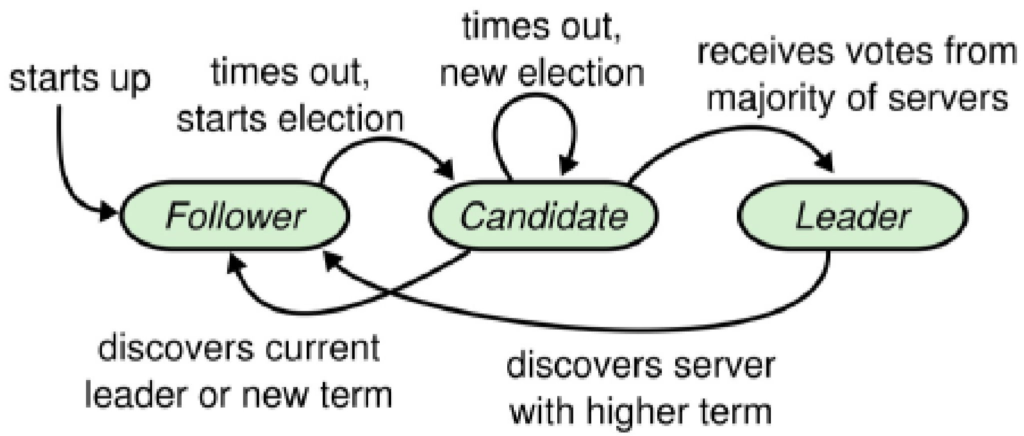
Split votes


Role: Follower
Voted for: 1
4
Role: Follower
Voted for: 2
3
TRUE
FALSE
FALSE
TRUE
1
2
FALSE
FALSE
TRUE
TRUE
Role: Candidate
Votes: 2
Role: Candidate
Votes: 2

Split votes


Role: Follower
4
Role: Follower
3
ELECTION TIMEOUT
1
Role: Follower
2
Role: Follower

Server Roles


starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

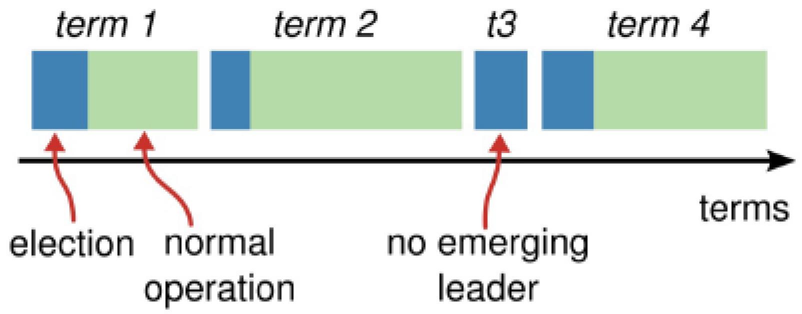
At any time, a server is either

- **Leader**: Handles client interactions and log replication
- **Follower**: Passively follows the orders of the leader
- **Candidate**: Aspirant in leader election

During normal operation

1 leader, N-1 followers

Terms



- Time is divided into **terms**
- Each terms begins with an election
- After a successful election, a single leader operates till the end of the term
- Transitions between terms are observed on servers at different times

Key role of Terms

Identify obsolete information

Messages

- RAFT only needs 2 messages.
- RequestVote includes term
- AppendEntries includes term and log entries
- Term acts as a logical clock

States

3 states a node can be in.

Follower

Candidate

Leader

Leader

- Only a single leader within a cluster
- Receives commands from client
- Commits commands to the log

Follower

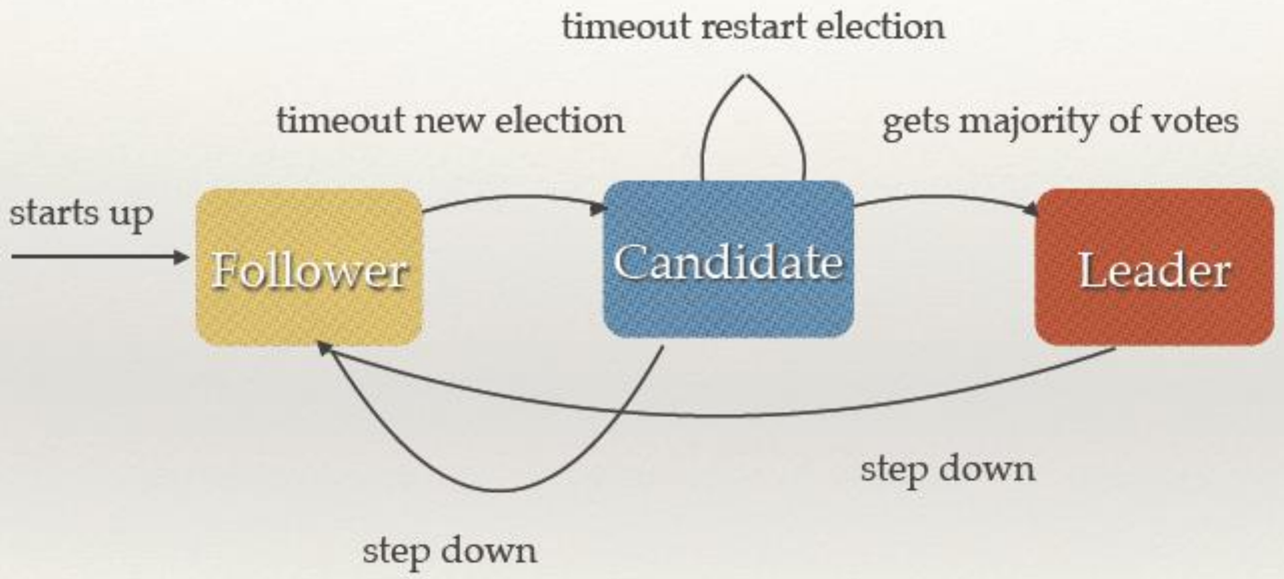
Follower

- Appends commands to log
- Votes for candidates
- Otherwise passive

Candidate

- Initiates Election
- Coordinates Votes

Leader Election



Leader election

- Servers start as followers
  - Followers expect to receive messages from leaders or candidates
  - Leaders must send **heartbeats** to maintain authority
- If *electionTimeout* elapses with no message, follower assumes that leader has crashed
- Follower starts new election
  - Increment current term (locally)
  - Change to candidate state
  - Vote for self
  - Send *RequestVote* message to all other servers
- Possible outcomes
  - Receive votes from majority of servers $\Rightarrow$ Become new leader
  - Receive message from valid leader $\Rightarrow$ Step down and become follower
  - No majority (*electionTimeout* elapses) $\Rightarrow$ Increment term and start new election

Properties of Leader Election

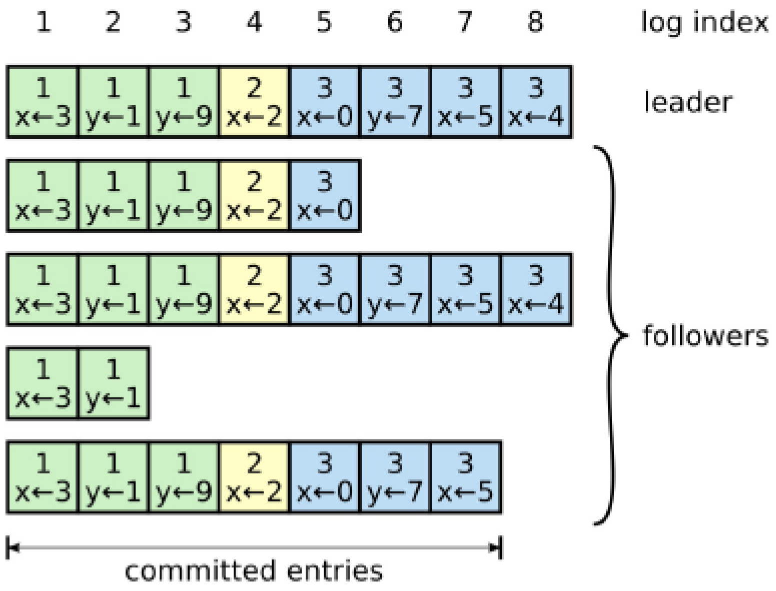
Safety: At most one leader per term

- Each server gives only one vote per term, namely to the first *RequestVote* message it receives (persist on disk)
- At most one server can accumulate majorities in same term

Liveness: Some candidate must eventually win

- Choose election timeouts randomly at every server
- One server usually times out and wins election before others consider elections
- Works well if time out is (much) larger than broadcast time

Log replication



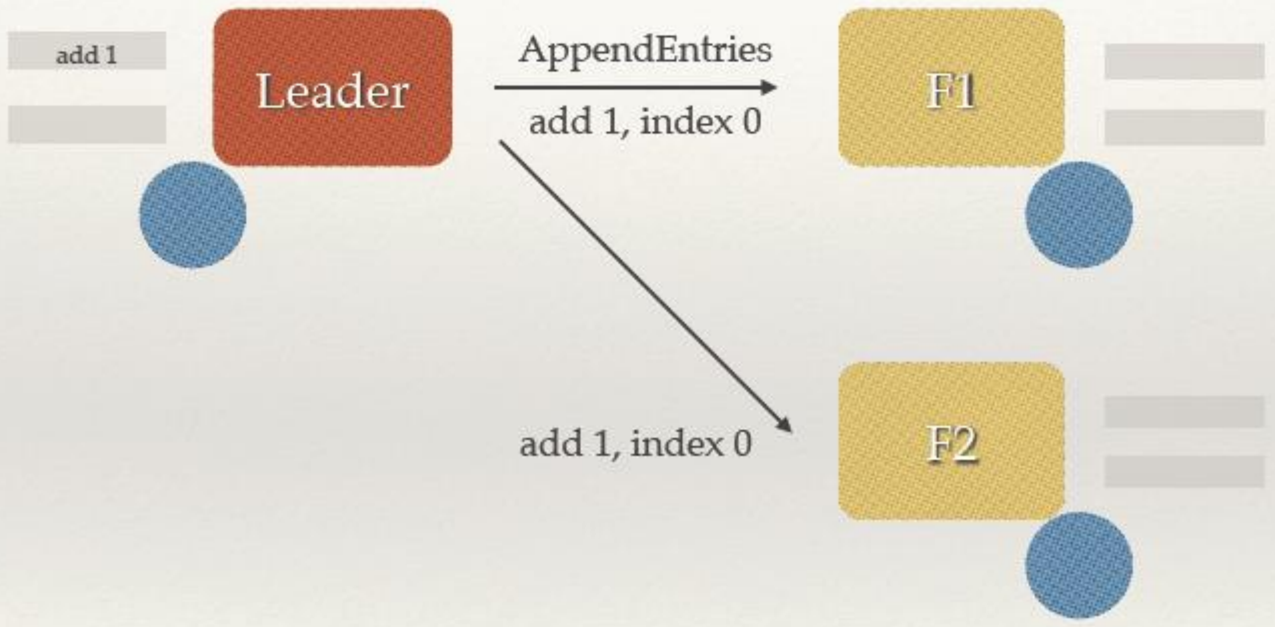
- Log entry comprises index + term + command
- Stored durably on disk to survive crashes
- Entry is **committed** if it is known to be stored on majority of servers

Operation (when no faults occur)

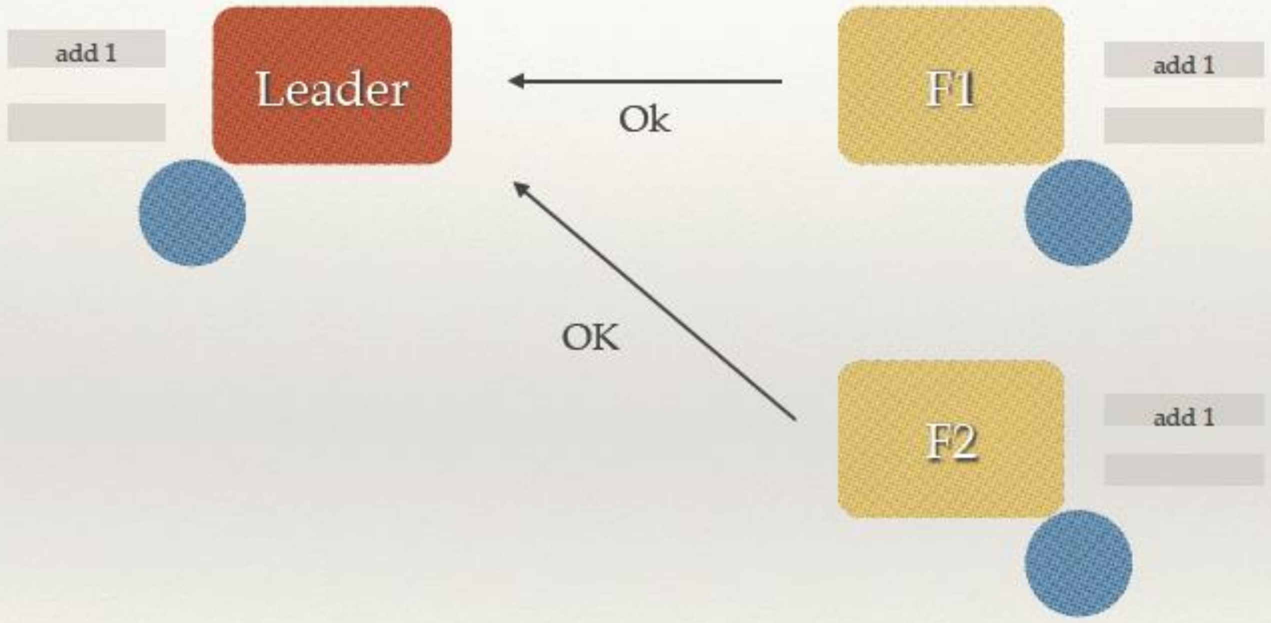
1. Client sends command to leader
2. Leader appends command to its own log
3. Leader sends *AppendEntry* to followers
4. Once new entry is committed
    - Leader executes command and returns result to client
    - Leader notifies followers about committed entries in subsequent *AppendEntries*
    - Followers pass committed commands to their state machines

$\Rightarrow$ 1 RTT to any majority of servers

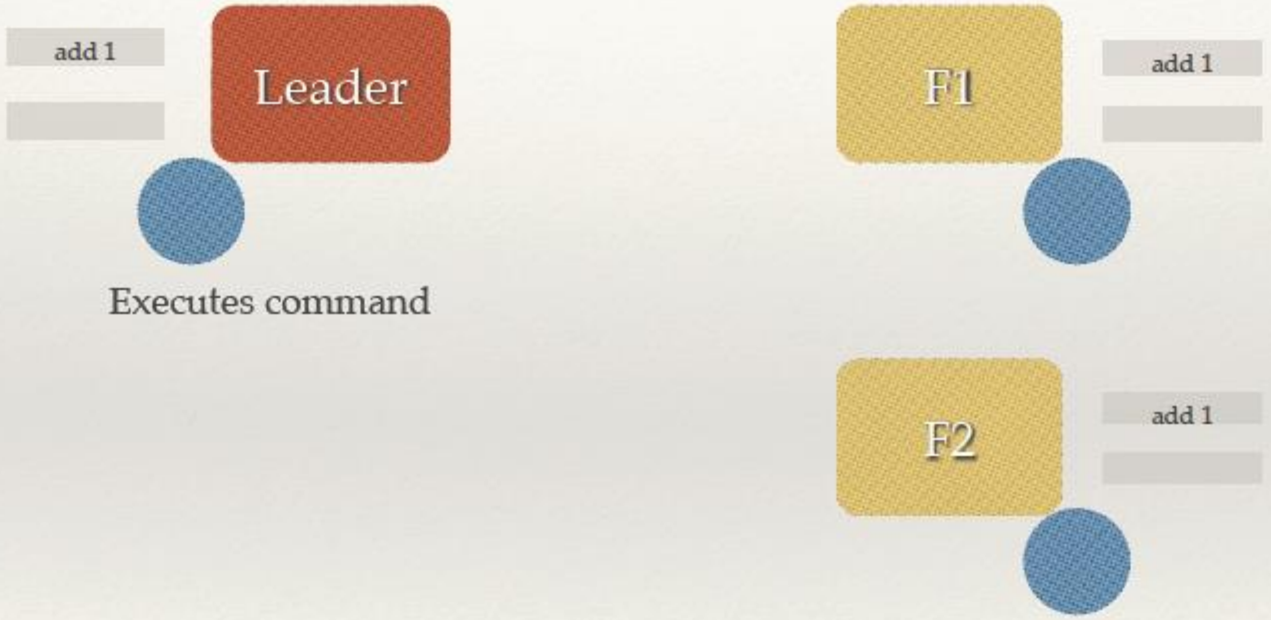
Log Replication



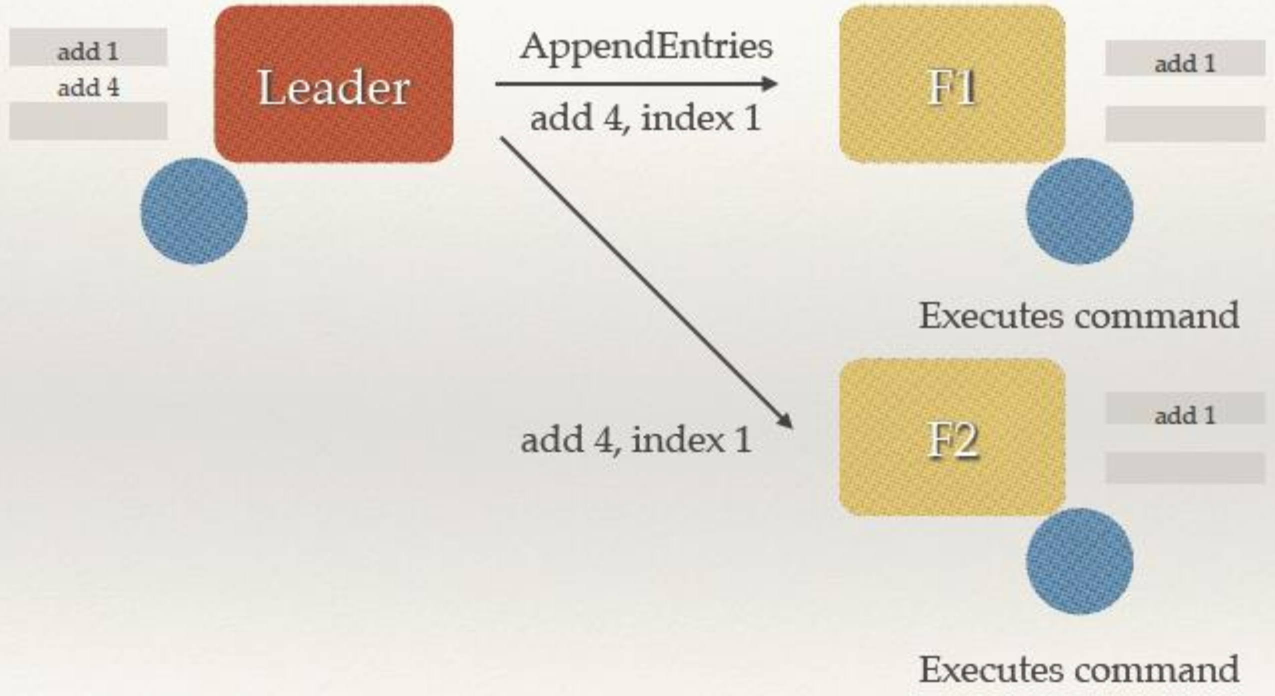
Log Replication



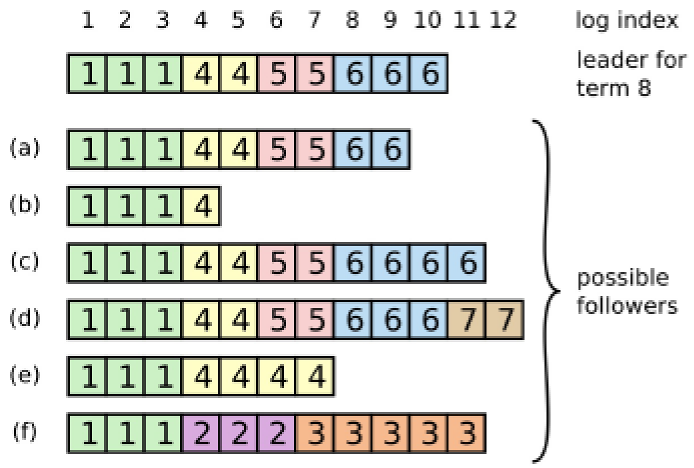
Log Replication



Log Replication



Log consistency



At beginning of new leader's term:

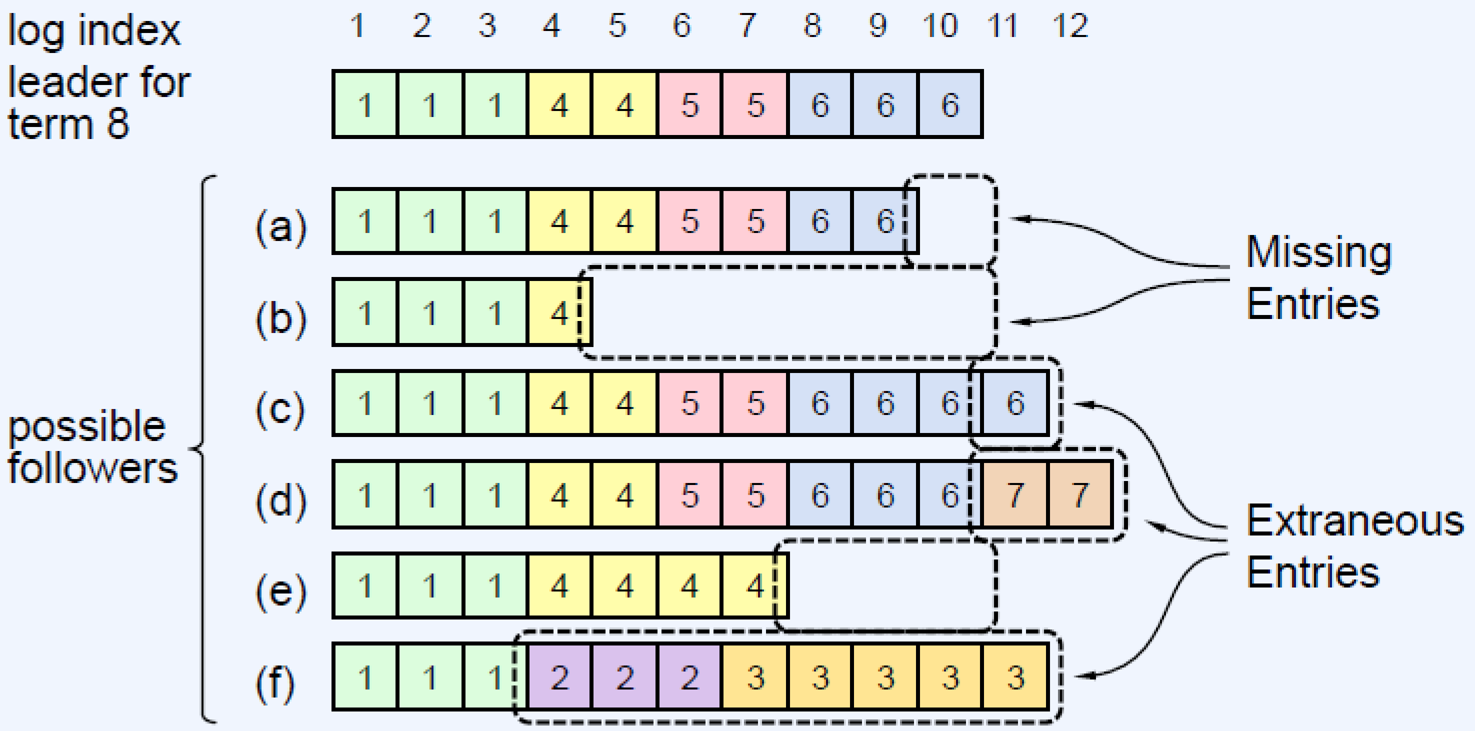
- Followers might miss entries
- Followers may have additional, uncommitted entries
- Both

Goal

Make follower's log identical to leader's log – without changing the leader log!

Log Inconsistencies

Leader changes can result in log inconsistencies:

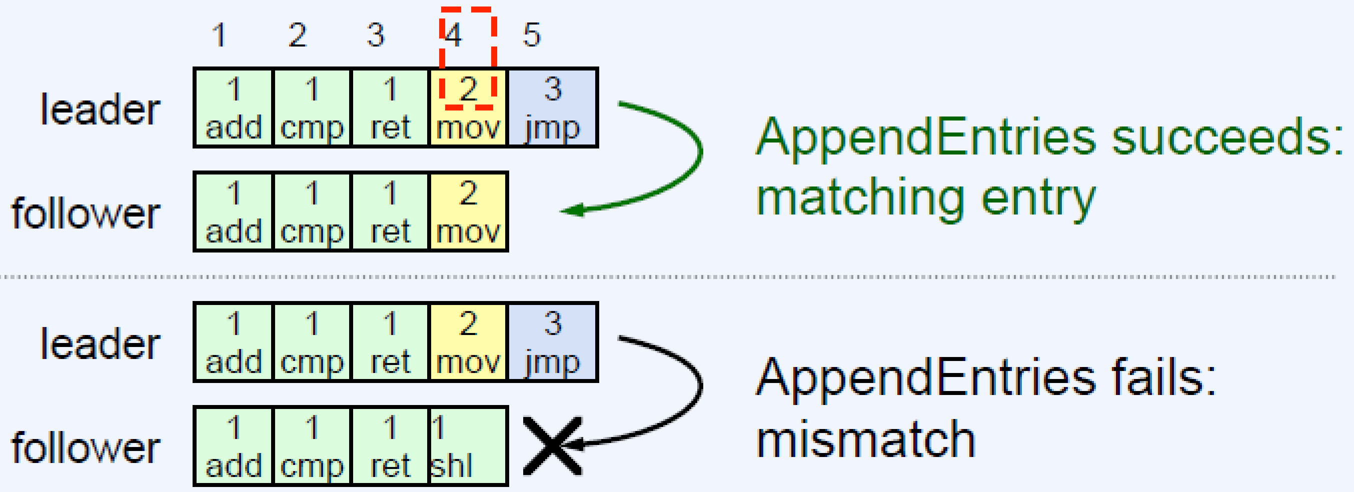


Log Matching Property

- Raft maintains the following properties, which together constitute the Log Matching Property:
  - If two entries in different logs have the same index and term, then they store the same command.
  - If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.

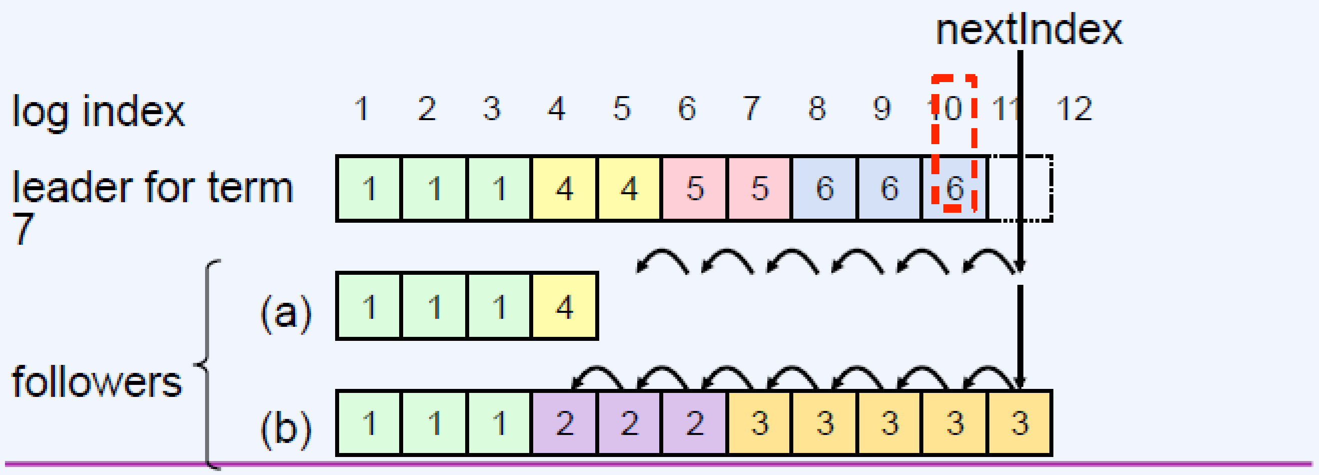
AppendEntries Consistency Check

- 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



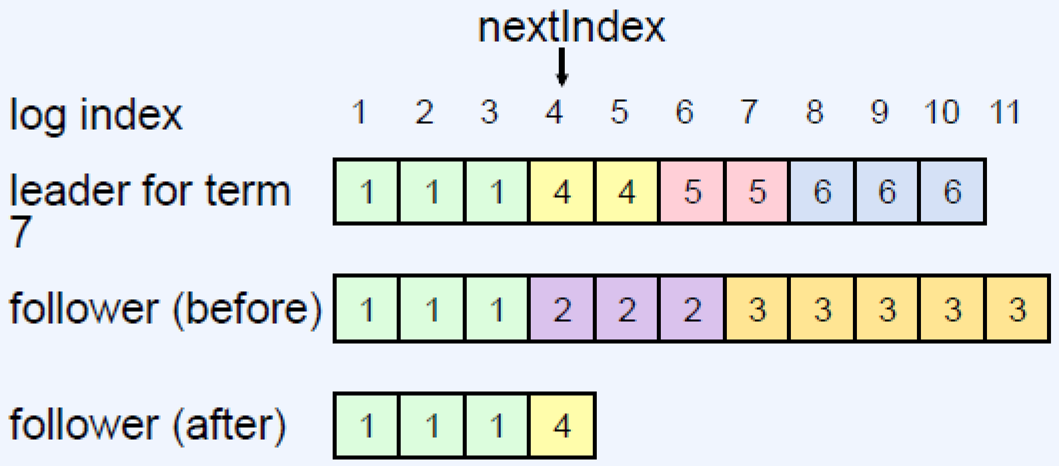
Repairing Follower Logs

- 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)
- When AppendEntries consistency check fails, decrement nextIndex and try again:



Repairing Logs, cont’d

- When follower overwrites inconsistent entry, it deletes all subsequent entries:

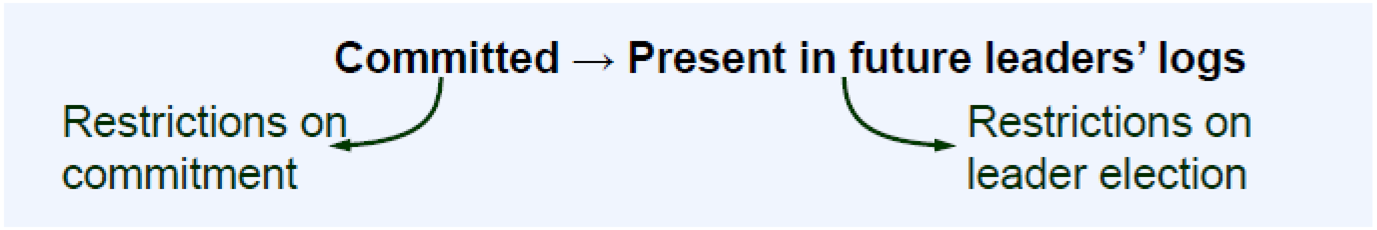


log index	1	2	3	4	5	6	7	8	9	10	11
leader for term 7	1	1	1	4	4	5	5	6	6	6	
follower (before)	1	1	1	2	2	2	3	3	3	3	3
follower (after)	1	1	1	4							

Safety Requirement

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

- If a leader has decided that a log entry is committed, this entry will be present in the logs of all future leaders.
  - Restriction on commitment
  - Restriction on leader election

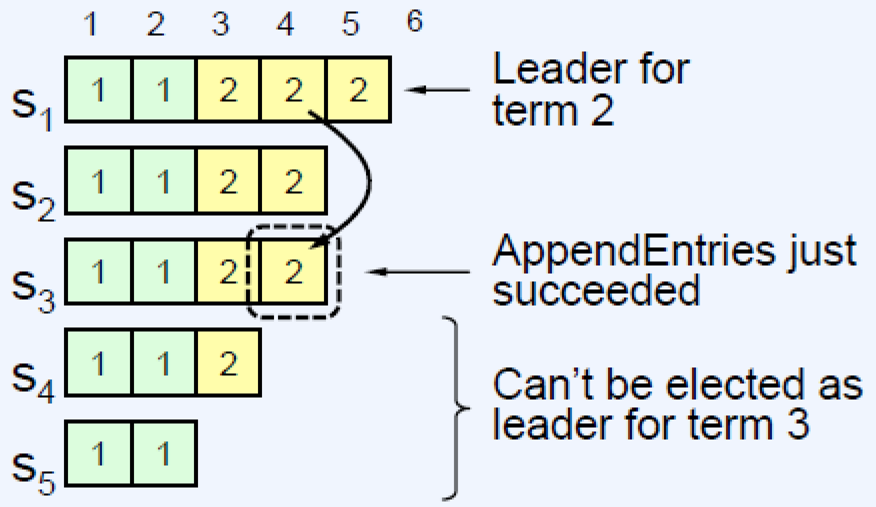

Committed → Present in future leaders' logs
Restrictions on commitment
Restrictions on leader election

Restriction on leader election

- Candidates can't tell which entries are committed
- Choose candidate whose log is most likely to contain all committed entries
  - Candidates include log info in *RequestVote*, i.e. index + term of last log entry
  - Server denies a candidate its vote if the server's log contains more information; i.e. last term in server is larger than last term in candidate, or, if they are equal, server's log contains more entries than candidate's log

Committing Entry from Current Term

- Case #1/2: Leader decides entry in current term is committed


1
2
3
4
5
6
s1
s2
s3
s4
s5
Leader for term 2
AppendEntries just succeeded
Can't be elected as leader for term 3

- Safe: leader for term 3 must contain entry 4

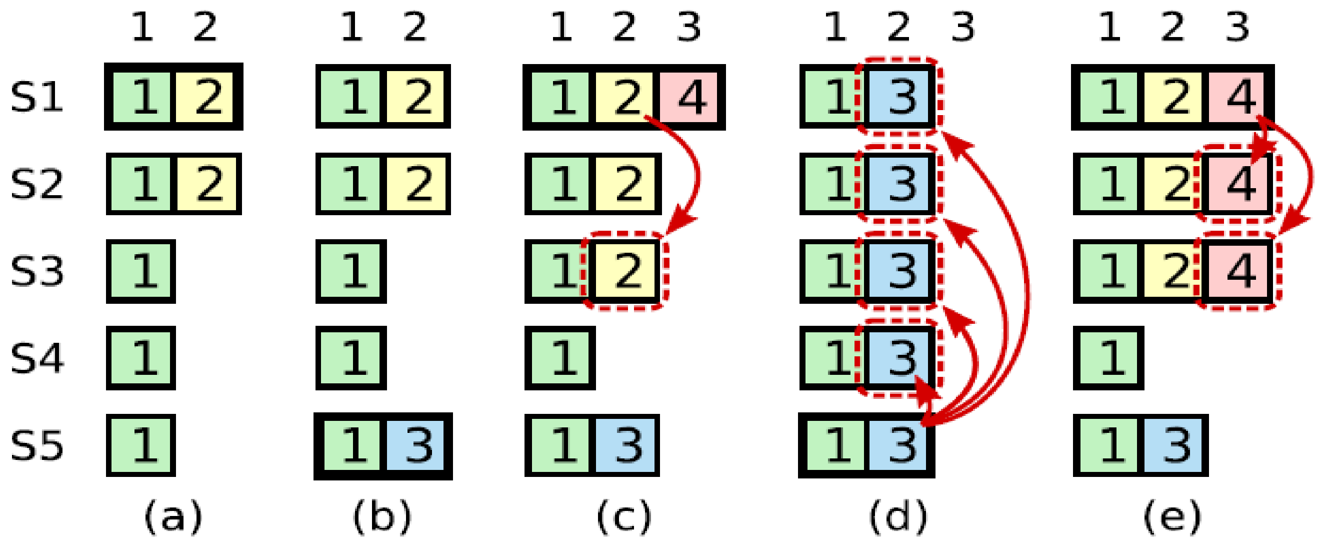
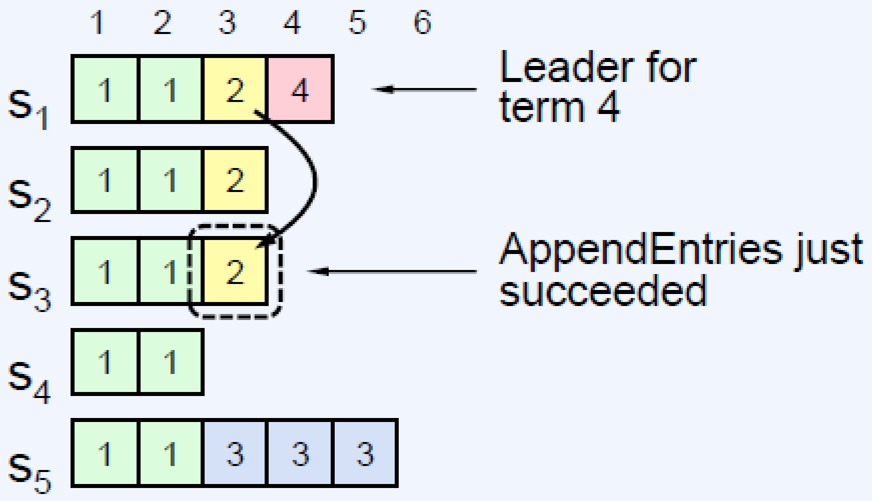


Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed. 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.

Committing Entry from Earlier Term

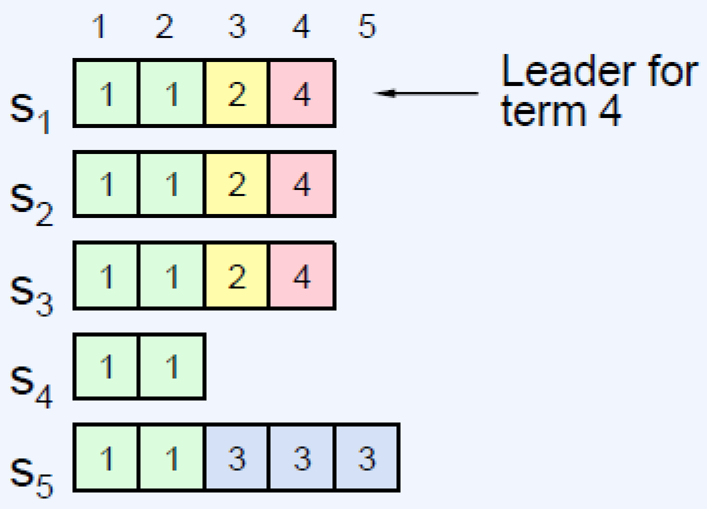
- Case #2/2: Leader is trying to finish committing entry from an earlier term



- Entry 3 not safely committed:
  - s_5 can be elected as leader for term 5
  - If elected, it will overwrite entry 3 on s_1, s_2, and s_3!

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:
  - s_5 cannot be elected leader for term 5
  - Entries 3 and 4 both safe



Combination of election rules and commitment rules makes Raft safe

When old leaders recover

- E.g. temporarily disconnected from network
- How does the leader realize that it has been replaced?
  - Every request contains term of sender
  - If sender's term is older, request is rejected; sender reverts to follower and updates its term
  - If receiver's term is older, it reverts to follower, updates its term und process then the message
- Why does it work?
  - Election updates terms of majority of servers
  - Old leader cannot commit new log entries

Guarantees

Election Safety: At most one leader can be elected in a given term.

Leader Append-Only: A leader never overwrites or deletes entries in its log; it only appends new entries.

Log Matching: If two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.

Leader Completeness: If a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.

State-Machine Safety: If a server has applied a log entry at a given index to its state machine, then no other server will every apply a different log entry for the same index.

RAFT Summary

- 2 types of messages, RequestVote and AppendEntries
- 3 states, Leader, Follower and Candidate
- Save Entries to persistent log