Distributed Systems

The second half of *Concurrent and Distributed Systems*

https://www.cl.cam.ac.uk/teaching/current/ConcDisSys

Dr. Martin Kleppmann (mk428@cam)

University of Cambridge

Computer Science Tripos, Part IB

Lecture 6

Consensus

Fault-tolerant total order broadcast

Total order broadcast is very useful for state machine replication.

Can implement total order broadcast by sending all messages via a single **leader**.

Problem: what if leader crashes/becomes unavailable?

Fault-tolerant total order broadcast

Total order broadcast is very useful for state machine replication.

Can implement total order broadcast by sending all messages via a single **leader**.

Problem: what if leader crashes/becomes unavailable?

- **Manual failover**: a human operator chooses a new leader, and reconfigures each node to use new leader

  Used in many databases! Fine for planned maintenance.

Fault-tolerant total order broadcast

Total order broadcast is very useful for state machine replication.

Can implement total order broadcast by sending all messages via a single **leader**.

Problem: what if leader crashes/becomes unavailable?

- **Manual failover**: a human operator chooses a new leader, and reconfigures each node to use new leader

  Used in many databases! Fine for planned maintenance.

  Unplanned outage? Humans are slow, may take a long time until system recovers...

Fault-tolerant total order broadcast

Total order broadcast is very useful for state machine replication.

Can implement total order broadcast by sending all messages via a single **leader**.

Problem: what if leader crashes/becomes unavailable?

- **Manual failover**: a human operator chooses a new leader, and reconfigures each node to use new leader

  Used in many databases! Fine for planned maintenance.

  Unplanned outage? Humans are slow, may take a long time until system recovers...
- Can we **automatically choose a new leader**?

Consensus and total order broadcast

- Traditional formulation of consensus: several nodes want to come to **agreement** about a single **value**

Consensus and total order broadcast

- Traditional formulation of consensus: several nodes want to come to **agreement** about a single **value**
- In context of total order broadcast: this value is the **next message to deliver**

Consensus and total order broadcast

- Traditional formulation of consensus: several nodes want to come to **agreement** about a single **value**
- In context of total order broadcast: this value is the **next message to deliver**
- Once one node **decides** on a certain message order, all nodes will decide the same order

Consensus and total order broadcast

- Traditional formulation of consensus: several nodes want to come to **agreement** about a single **value**
- In context of total order broadcast: this value is the **next message to deliver**
- Once one node **decides** on a certain message order, all nodes will decide the same order
- Consensus and total order broadcast are formally equivalent

Consensus and total order broadcast

- Traditional formulation of consensus: several nodes want to come to **agreement** about a single **value**
- In context of total order broadcast: this value is the **next message to deliver**
- Once one node **decides** on a certain message order, all nodes will decide the same order
- Consensus and total order broadcast are formally equivalent

Common consensus algorithms:

- **Paxos**: single-value consensus
  Multi-Paxos: generalisation to total order broadcast

Consensus and total order broadcast

- Traditional formulation of consensus: several nodes want to come to **agreement** about a single **value**
- In context of total order broadcast: this value is the **next message to deliver**
- Once one node **decides** on a certain message order, all nodes will decide the same order
- Consensus and total order broadcast are formally equivalent

Common consensus algorithms:

- **Paxos**: single-value consensus
  Multi-Paxos: generalisation to total order broadcast
- **Raft**, **Viewstamped Replication**, **Zab**:
  FIFO-total order broadcast by default

Consensus system models

Paxos, Raft, etc. assume a **partially synchronous, crash-recovery** system model.

Consensus system models

Paxos, Raft, etc. assume a **partially synchronous, crash-recovery** system model.

Why not asynchronous?

- **FLP result** (Fischer, Lynch, Paterson):
  There is no deterministic consensus algorithm that is guaranteed to terminate in an asynchronous crash-stop system model.

Consensus system models

Paxos, Raft, etc. assume a **partially synchronous, crash-recovery** system model.

Why not asynchronous?

- **FLP result** (Fischer, Lynch, Paterson):
  There is no deterministic consensus algorithm that is guaranteed to terminate in an asynchronous crash-stop system model.
- Paxos, Raft, etc. use clocks only used for timeouts/failure detector to ensure progress. Safety (correctness) does not depend on timing.

Consensus system models

Paxos, Raft, etc. assume a **partially synchronous, crash-recovery** system model.

Why not asynchronous?

- **FLP result** (Fischer, Lynch, Paterson):
  There is no deterministic consensus algorithm that is guaranteed to terminate in an asynchronous crash-stop system model.
- Paxos, Raft, etc. use clocks only used for timeouts/failure detector to ensure progress. Safety (correctness) does not depend on timing.

There are also consensus algorithms for a partially synchronous **Byzantine** system model (used in blockchains)

Leader election

Multi-Paxos, Raft, etc. use a leader to sequence messages.

Leader election

Multi-Paxos, Raft, etc. use a leader to sequence messages.

- Use a **failure detector** (timeout) to determine suspected crash or unavailability of leader.
- On suspected leader crash, **elect a new one**.

Leader election

Multi-Paxos, Raft, etc. use a leader to sequence messages.

- Use a **failure detector** (timeout) to determine suspected crash or unavailability of leader.
- On suspected leader crash, **elect a new one**.
- Prevent **two leaders at the same time** ("split-brain")!

Leader election

Multi-Paxos, Raft, etc. use a leader to sequence messages.

- Use a **failure detector** (timeout) to determine suspected crash or unavailability of leader.
- On suspected leader crash, **elect a new one**.
- Prevent **two leaders at the same time** ("split-brain")!

Ensure ≤ 1 leader per **term**:

- Term is incremented every time a leader election is started

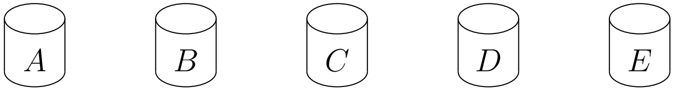
Leader election

Multi-Paxos, Raft, etc. use a leader to sequence messages.

- Use a **failure detector** (timeout) to determine suspected crash or unavailability of leader.
- On suspected leader crash, **elect a new one**.
- Prevent **two leaders at the same time** ("split-brain")!

Ensure ≤ 1 leader per **term**:

- Term is incremented every time a leader election is started
- A node can only **vote once** per term
- Require a **quorum** of nodes to elect a leader in a term



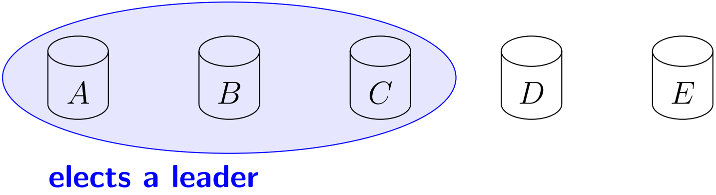
Leader election

Multi-Paxos, Raft, etc. use a leader to sequence messages.

- Use a **failure detector** (timeout) to determine suspected crash or unavailability of leader.
- On suspected leader crash, **elect a new one**.
- Prevent **two leaders at the same time** ("split-brain")!

Ensure ≤ 1 leader per **term**:

- Term is incremented every time a leader election is started
- A node can only **vote once** per term
- Require a **quorum** of nodes to elect a leader in a term



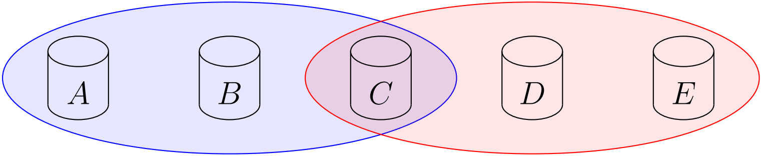
Leader election

Multi-Paxos, Raft, etc. use a leader to sequence messages.

- Use a **failure detector** (timeout) to determine suspected crash or unavailability of leader.
- On suspected leader crash, **elect a new one**.
- Prevent **two leaders at the same time** ("split-brain")!

Ensure ≤ 1 leader per **term**:

- Term is incremented every time a leader election is started
- A node can only **vote once** per term
- Require a **quorum** of nodes to elect a leader in a term



elects a leader

cannot elect a different leader
because C already voted

Can we guarantee there is only one leader?

Can guarantee unique leader **per term**.

Can we guarantee there is only one leader?

Can guarantee unique leader **per term**.

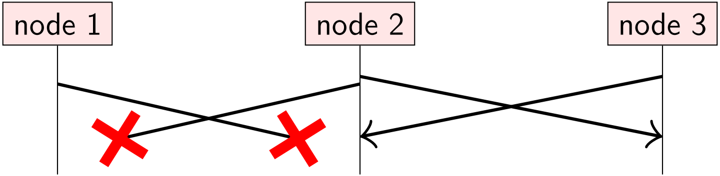
Cannot prevent having multiple leaders from different terms.

Can we guarantee there is only one leader?

Can guarantee unique leader **per term**.

Cannot prevent having multiple leaders from different terms.

Example: node 1 is leader in term t, but due to a network partition it can no longer communicate with nodes 2 and 3:



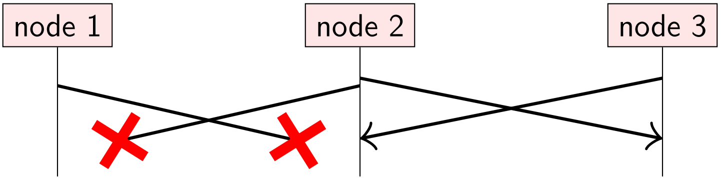
Nodes 2 and 3 may elect a new leader in term $t + 1$.

Can we guarantee there is only one leader?

Can guarantee unique leader **per term**.

Cannot prevent having multiple leaders from different terms.

Example: node 1 is leader in term t, but due to a network partition it can no longer communicate with nodes 2 and 3:

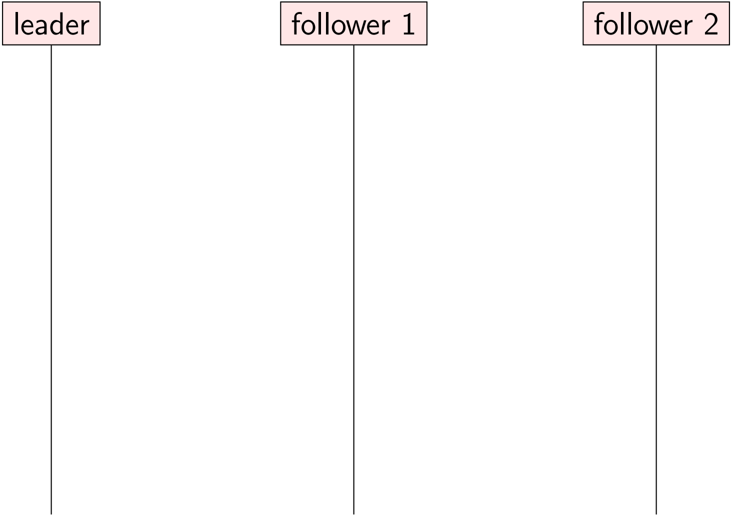


Nodes 2 and 3 may elect a new leader in term $t + 1$.

Node 1 may not even know that a new leader has been elected!

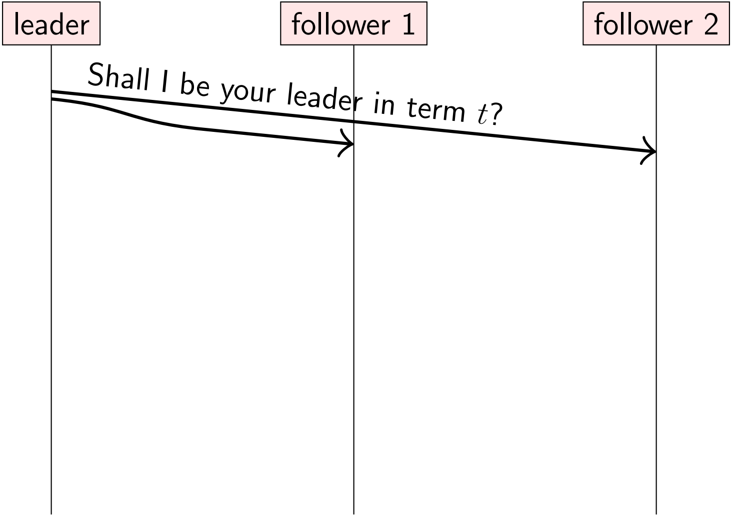
Checking if a leader has been voted out

For every decision (message to deliver), the leader must first get acknowledgements from a quorum.



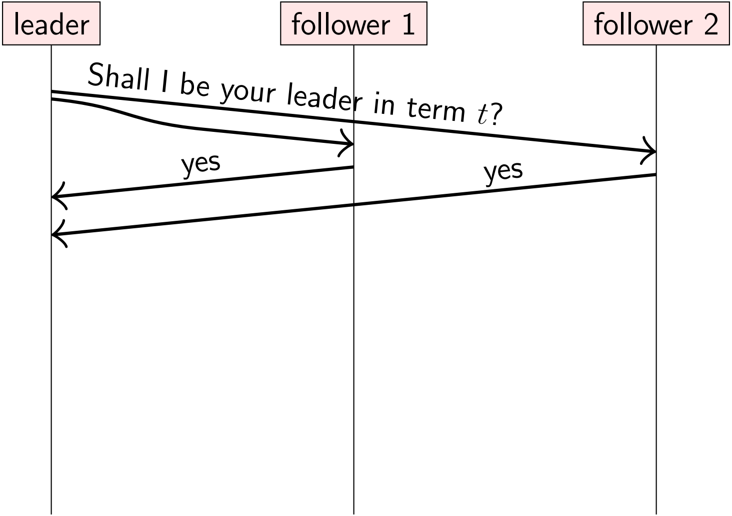
Checking if a leader has been voted out

For every decision (message to deliver), the leader must first get acknowledgements from a quorum.



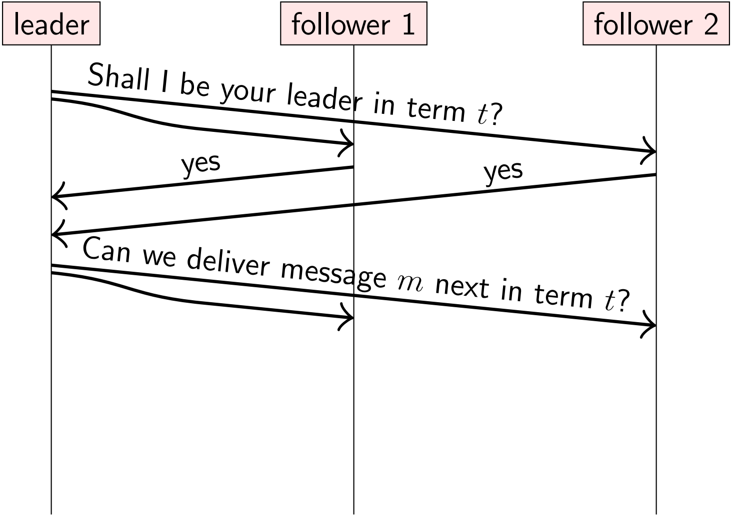
Checking if a leader has been voted out

For every decision (message to deliver), the leader must first get acknowledgements from a quorum.



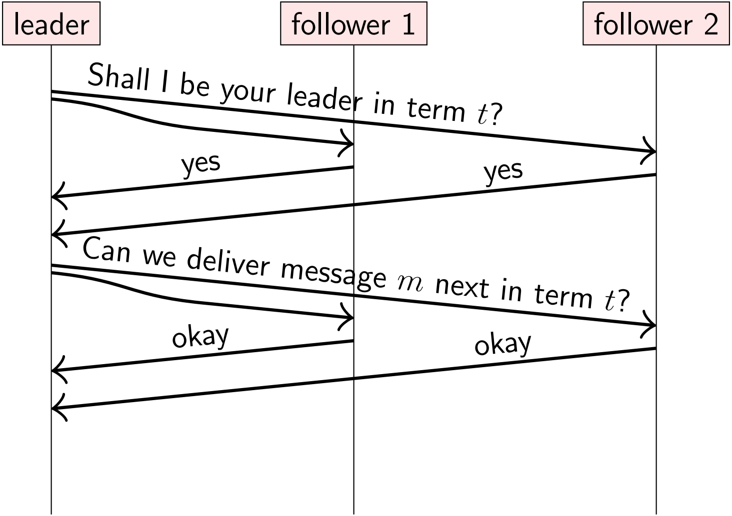
Checking if a leader has been voted out

For every decision (message to deliver), the leader must first get acknowledgements from a quorum.



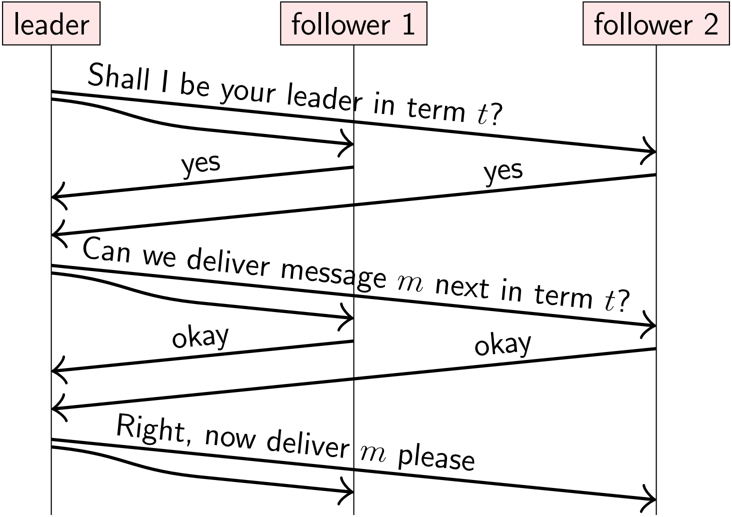
Checking if a leader has been voted out

For every decision (message to deliver), the leader must first get acknowledgements from a quorum.

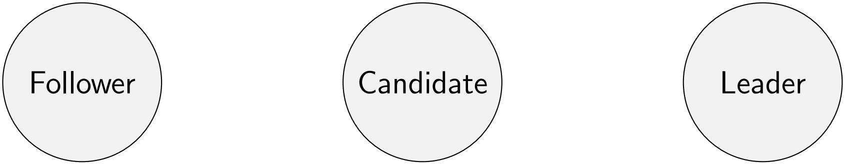


Checking if a leader has been voted out

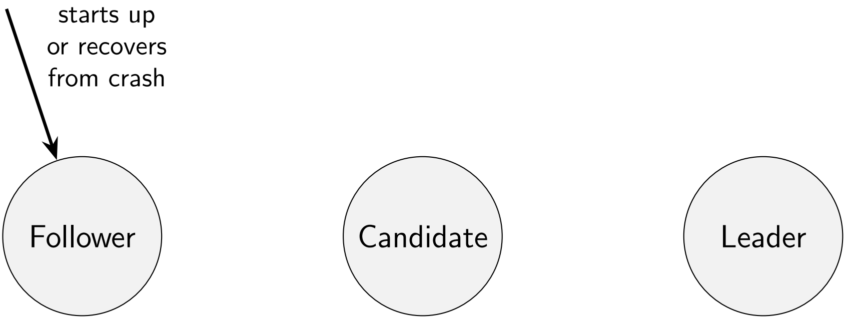
For every decision (message to deliver), the leader must first get acknowledgements from a quorum.



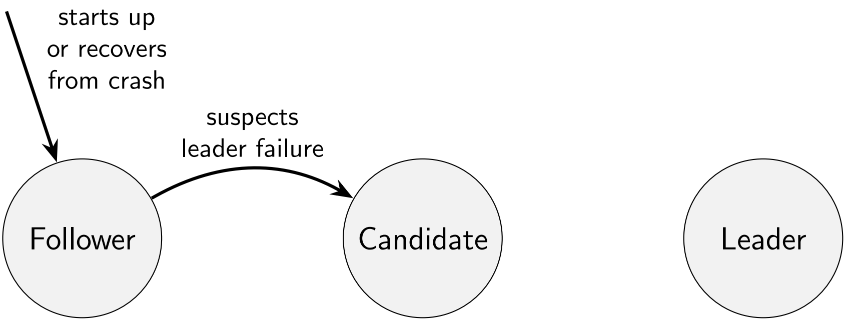
Node state transitions in Raft



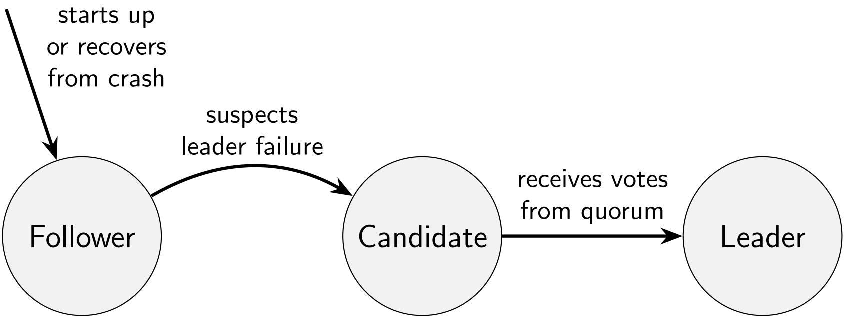
Node state transitions in Raft



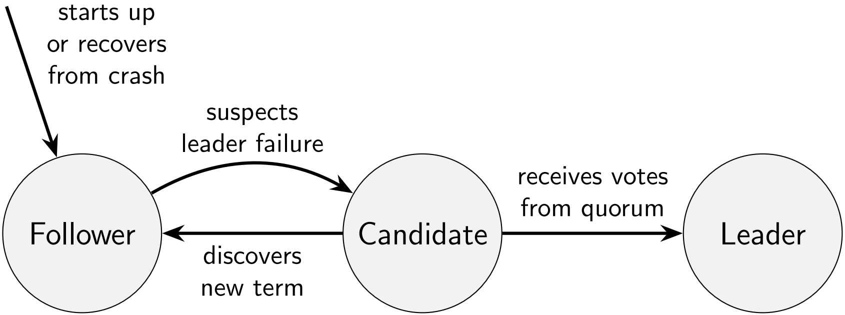
Node state transitions in Raft



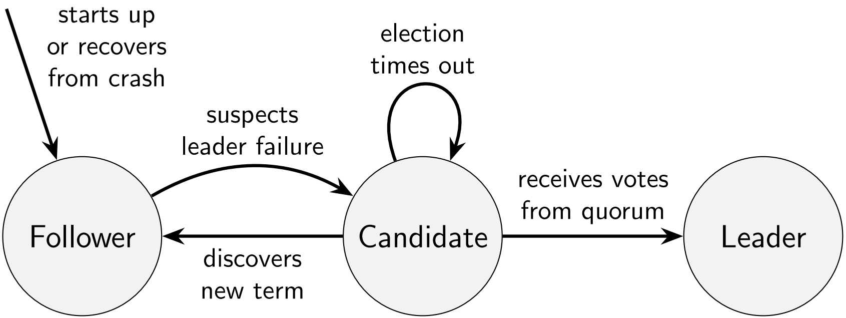
Node state transitions in Raft



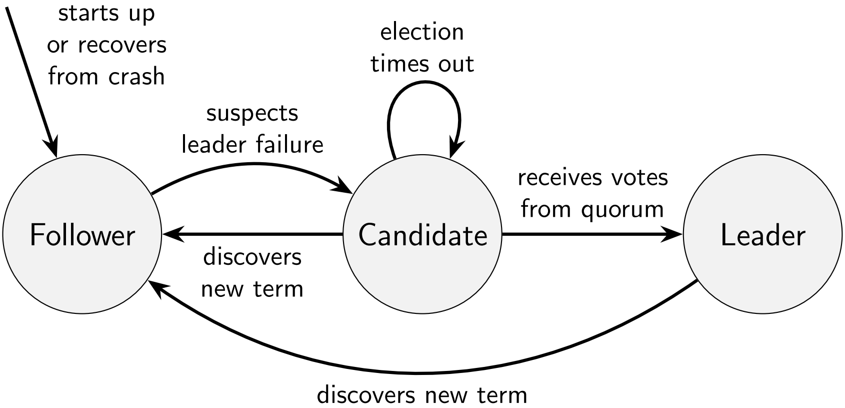
Node state transitions in Raft



Node state transitions in Raft



Node state transitions in Raft



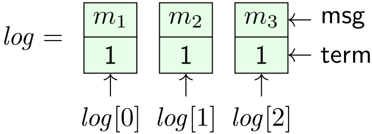
Raft (1/9): initialisation

on initialisation **do**
  $currentTerm := 0$; $votedFor :=$ null
  $log := \langle\rangle$; $commitLength := 0$
  $currentRole :=$ follower; $currentLeader :=$ null
  $votesReceived := \{\}$; $sentLength := \langle\rangle$; $ackedLength := \langle\rangle$
end on

on recovery from crash **do**
  $currentRole :=$ follower; $currentLeader :=$ null
  $votesReceived := \{\}$; $sentLength := \langle\rangle$; $ackedLength := \langle\rangle$
end on

on node $nodeId$ suspects leader has failed, or on election timeout **do**
  $currentTerm := currentTerm + 1$; $currentRole :=$ candidate
  $votedFor := nodeId$; $votesReceived := \{nodeId\}$; $lastTerm := 0$
  if $log.\text{length} > 0$ **then** $lastTerm := log[log.\text{length} - 1].\text{term}$; **end if**
  $msg := (\text{VoteRequest}, nodeId, currentTerm, log.\text{length}, lastTerm)$
  for each $node \in nodes$: **send** msg to $node$
  start election timer
end on

Raft (1/9): initialisation



on initialisation **do**
    $currentTerm := 0$; $votedFor :=$ null
    $log := \langle\rangle$; $commitLength := 0$
    $currentRole :=$ follower; $currentLeader :=$ null
    $votesReceived := \{\}$; $sentLength := \langle\rangle$; $ackedLength := \langle\rangle$
end on

on recovery from crash **do**
    $currentRole :=$ follower; $currentLeader :=$ null
    $votesReceived := \{\}$; $sentLength := \langle\rangle$; $ackedLength := \langle\rangle$
end on

on node $nodeId$ suspects leader has failed, or on election timeout **do**
    $currentTerm := currentTerm + 1$; $currentRole :=$ candidate
    $votedFor := nodeId$; $votesReceived := \{nodeId\}$; $lastTerm := 0$
    if log.length > 0 **then** $lastTerm := log[log$.length $- 1]$.term; **end if**
    $msg :=$ (VoteRequest, $nodeId$, $currentTerm$, log.length, $lastTerm$)
    for each $node \in nodes$: **send** msg to $node$
    start election timer
end on

Raft (2/9): voting on a new leader

on receiving $(\text{VoteRequest}, cId, cTerm, cLogLength, cLogTerm)$
at node $nodeId$ **do**

$myLogTerm := log[log.\text{length} - 1].\text{term}$
$logOk := (cLogTerm > myLogTerm) \lor$
$(cLogTerm = myLogTerm \land cLogLength \geq log.\text{length})$

$termOk := (cTerm > currentTerm) \lor$
$(cTerm = currentTerm \land votedFor \in \{cId, \text{null}\})$

if $logOk \land termOk$ **then**
$currentTerm := cTerm$
$currentRole := \text{follower}$
$votedFor := cId$
send $(\text{VoteResponse}, nodeId, currentTerm, \text{true})$ to node cId
else
send $(\text{VoteResponse}, nodeId, currentTerm, \text{false})$ to node cId
end if
end on

Raft (2/9): voting on a new leader

c for candidate

on receiving (VoteRequest, cId, $cTerm$, $cLogLength$, $cLogTerm$)
    at node $nodeId$ **do**
  $myLogTerm := log[log.\text{length} - 1].\text{term}$
  $logOk := (cLogTerm > myLogTerm) \lor$
    $(cLogTerm = myLogTerm \land cLogLength \geq log.\text{length})$

  $termOk := (cTerm > currentTerm) \lor$
    $(cTerm = currentTerm \land votedFor \in \{cId, \text{null}\})$

  if $logOk \land termOk$ **then**
    $currentTerm := cTerm$
    $currentRole :=$ follower
    $votedFor := cId$
    send (VoteResponse, $nodeId$, $currentTerm$, true) to node cId
  else
    send (VoteResponse, $nodeId$, $currentTerm$, false) to node cId
  end if
end on

Raft (3/9): collecting votes

on receiving (VoteResponse, $voterId, term, granted$) at $nodeId$ **do**
    if $currentRole = \text{candidate} \wedge term = currentTerm \wedge granted$ **then**
        $votesReceived := votesReceived \cup \{voterId\}$
        if $|votesReceived| \geq \lceil (|nodes| + 1)/2 \rceil$ **then**
            $currentRole := \text{leader};\ currentLeader := nodeId$
            cancel election timer
            for each $follower \in nodes \setminus \{nodeId\}$ **do**
                $sentLength[follower] := log.\text{length}$
                $ackedLength[follower] := 0$
                $\textsc{ReplicateLog}(nodeId, follower)$
            end for
        end if
    else if $term > currentTerm$ **then**
        $currentTerm := term$
        $currentRole := \text{follower}$
        $votedFor := \text{null}$
        cancel election timer
    end if
end on

Raft (4/9): broadcasting messages

on request to broadcast msg at node $nodeId$ **do**
  if $currentRole$ = leader **then**
    append the record (msg : msg, term : $currentTerm$) to log
    $ackedLength[nodeId]$:= log.length
    for each $follower \in nodes \setminus \{nodeId\}$ **do**
      $\textsc{ReplicateLog}(nodeId, follower)$
    end for
  else
    forward the request to $currentLeader$ via a FIFO link
  end if
end on

periodically at node $nodeId$ **do**
  if $currentRole$ = leader **then**
    for each $follower \in nodes \setminus \{nodeId\}$ **do**
      $\textsc{ReplicateLog}(nodeId, follower)$
    end for
  end if
end do

Raft (5/9): replicating from leader to followers

Called on the leader whenever there is a new message in the log, and also periodically. If there are no new messages, $entries$ is the empty list. LogRequest messages with $entries = \langle\rangle$ serve as heartbeats, letting followers know that the leader is still alive.

function $\textsc{ReplicateLog}(leaderId, followerId)$
  $i := sentLength[followerId]$
  $entries := \langle log[i],\ log[i+1],\ \ldots,\ log[log.\text{length} - 1]\rangle$
  $prevLogTerm := 0$
  if $i > 0$ **then**
    $prevLogTerm := log[i-1].\text{term}$
  end if
  send $(\text{LogRequest}, leaderId, currentTerm, i, prevLogTerm,$
      $commitLength, entries)$ to $followerId$
end function

Raft (6/9): followers receiving messages

$$
\begin{array}{l}
\textbf{on } \text{receiving } (\text{LogRequest}, \mathit{leaderId}, \mathit{term}, \mathit{logLength}, \mathit{logTerm}, \\
\qquad\qquad \mathit{leaderCommit}, \mathit{entries}) \text{ at node } \mathit{nodeId} \textbf{ do} \\
\quad \textbf{if } \mathit{term} > \mathit{currentTerm} \textbf{ then} \\
\qquad \mathit{currentTerm} := \mathit{term};\ \mathit{votedFor} := \text{null} \\
\quad \textbf{end if} \\
\quad \mathit{logOk} := (\mathit{log}.\text{length} \geq \mathit{logLength}) \\
\quad \textbf{if } \mathit{logOk} \wedge (\mathit{logLength} > 0) \textbf{ then} \\
\qquad \mathit{logOk} := (\mathit{logTerm} = \mathit{log}[\mathit{logLength} - 1].\text{term}) \\
\quad \textbf{end if} \\
\\
\quad \textbf{if } \mathit{term} = \mathit{currentTerm} \wedge \mathit{logOk} \textbf{ then} \\
\qquad \mathit{currentRole} := \text{follower};\ \mathit{currentLeader} := \mathit{leaderId} \\
\qquad \textsc{AppendEntries}(\mathit{logLength}, \mathit{leaderCommit}, \mathit{entries}) \\
\qquad \mathit{ack} := \mathit{logLength} + \mathit{entries}.\text{length} \\
\qquad \textbf{send } (\text{LogResponse}, \mathit{nodeId}, \mathit{currentTerm}, \mathit{ack}, \text{true}) \text{ to } \mathit{leaderId} \\
\quad \textbf{else} \\
\qquad \textbf{send } (\text{LogResponse}, \mathit{nodeId}, \mathit{currentTerm}, 0, \text{false}) \text{ to } \mathit{leaderId} \\
\quad \textbf{end if} \\
\textbf{end on}
\end{array}
$$

Raft (7/9): updating followers' logs

function $\textsc{AppendEntries}(logLength, leaderCommit, entries)$
    if $entries$.length $> 0 \ \wedge \ log$.length $> logLength$ **then**
        if $log[logLength]$.term $\neq entries[0]$.term **then**
            $log := \langle log[0], \ log[1], \ \ldots, \ log[logLength - 1]\rangle$
        end if
    end if
    if $logLength + entries$.length $> log$.length **then**
        for $i := log$.length $- \ logLength$ **to** $entries$.length $- 1$ **do**
            append $entries[i]$ to log
        end for
    end if
    if $leaderCommit > commitLength$ **then**
        for $i := commitLength$ **to** $leaderCommit - 1$ **do**
            deliver $log[i]$.msg to the application
        end for
        $commitLength := leaderCommit$
    end if
end function

Raft (8/9): leader receiving log acknowledgements

```
on receiving (LogResponse, follower, term, ack, success) at nodeId do
    if term = currentTerm ∧ currentRole = leader then
        if success = true ∧ ack ≥ ackedLength[follower] then
            sentLength[follower] := ack
            ackedLength[follower] := ack
            CommitLogEntries()
        else if sentLength[follower] > 0 then
            sentLength[follower] := sentLength[follower] − 1
            ReplicateLog(nodeId, follower)
        end if
    else if term > currentTerm then
        currentTerm := term
        currentRole := follower
        votedFor := null
    end if
end on
```

Raft (9/9): leader committing log entries

Any log entries that have been acknowledged by a quorum of nodes are ready to be committed by the leader. When a log entry is committed, its message is delivered to the application.

define $\mathrm{acks}(\mathit{length}) = |\{n \in \mathit{nodes} \mid \mathit{ackedLength}[n] \geq \mathit{length}\}|$

function $\textsc{CommitLogEntries}$
    $\mathit{minAcks} := \lceil (|\mathit{nodes}| + 1)/2 \rceil$
    $\mathit{ready} := \{\mathit{len} \in \{1, \dots, \mathit{log}.\mathsf{length}\} \mid \mathrm{acks}(\mathit{len}) \geq \mathit{minAcks}\}$
    if $\mathit{ready} \neq \{\} \ \wedge\ \max(\mathit{ready}) > \mathit{commitLength}\ \wedge$
            $\mathit{log}[\max(\mathit{ready}) - 1].\mathsf{term} = \mathit{currentTerm}$ **then**
        for $i := \mathit{commitLength}$ **to** $\max(\mathit{ready}) - 1$ **do**
            deliver $\mathit{log}[i]$.msg to the application
        end for
        $\mathit{commitLength} := \max(\mathit{ready})$
    end if
end function