RAFT[1] Consensus Protocol

Week 6, Lecture 2

Onur Ascigil

[1] Ongaro et al. “In Search of an Understandable Consensus Algorithm”

Paxos vs RAFT

Team Paxos

Amazon ECS

cassandra

PaxosStore:
High-availability Storage Made Practical in WeChat

Jianjun Zheng[†] Qian Lin[†*] Jiatao Xu[†] Cheng Wei[†]
Chuwei Zeng[†] Pingan Yang[†] Yunfan Zhang[†]
[†]Tencent Inc. [‡]National University of Singapore
[†]{rockzheng, sunnyxu, dengoswei, eddyzeng, ypaapyyang, ifanzhangj}@tencent.com
[‡]linqian@comp.nus.edu.sg

Windows Azure Storage: A Highly Available
Cloud Storage Service with Strong Consistency

Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu,
Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju,
Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal,
Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand,
Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, Leonidas Rigas

Microsoft

Clustrix

ceph

infinit

Megastore: Providing Scalable, Highly Available
Storage for Interactive Services

Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson,
Jean-Michel Léon, Yawei Li, Alexander Lloyd, Vadim Yushprakh
Google, Inc.

Large-scale cluster management at Google with Borg

Abhishek Verma[†] Luis Pedrosa[‡] Madhukar Korupolu
David Oppenheimer Eric Tune John Wilkes

Google Inc.

Doozer

LogDevice

The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc.*

neo4j

Team Raft

LogCabin

KV

HashiCorp
Consul

neo4j

RethinkDB

hazelcast

Atomix

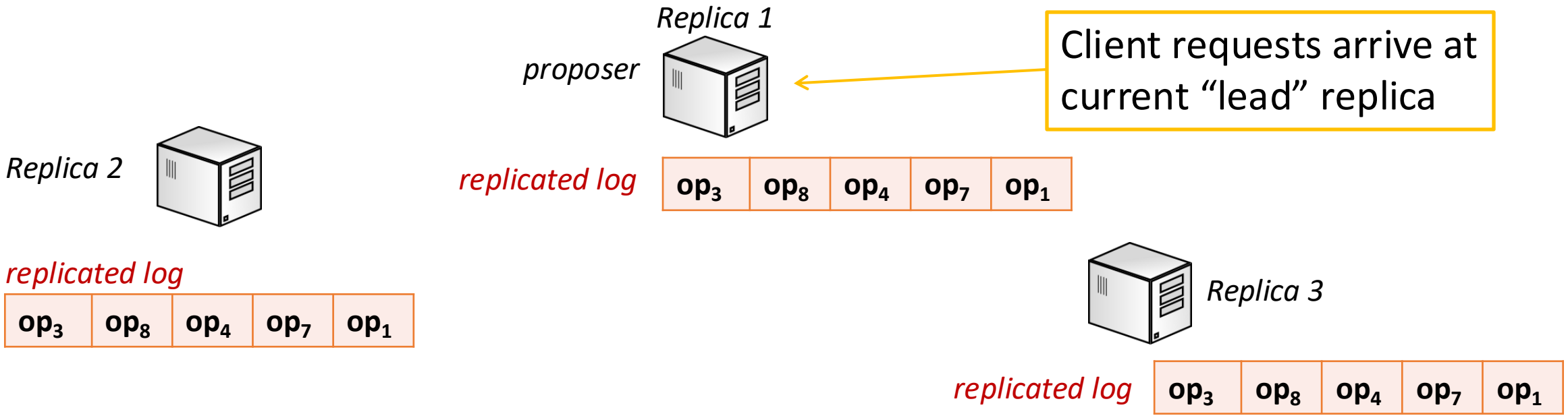
etcd

Cockroach DB

Distributed Consensus

- Implementing a *replicated state machine* requires applying the same sequence of (i.e., **totally ordered**) client requests (operations) to the replicas
- Previously, we studied Paxos (Week 5, Lecture 2) to understand how distributed agreement can take place for **a single operation (i.e., a single client request)**
- The extension of Paxos for agreeing on a sequence of operations (i.e. Multi-Paxos) is not explained in sufficient details by its designer (Leslie Lamport) which leaves certain implementation details and edge cases open to interpretation

Multi-Paxos: Unanswered questions



- How is the current "lead" replica (proposer) selected?
  - What if that replica fails?
  - What if a replica with an out-dated log becomes the lead?
- What happens if a replica's log becomes out-dated (due to failures)?
  - How can a replica catch up and update its log?

…

RAFT Consensus Protocol

- RAFT aims to have a clear and detailed description that can be easily converted to an implementation
  - This is not the case with (Multi-)Paxos
- Works under non-Byzantine, crash-only faults, similar to Paxos

"There are significant gaps between the description of the (Multi-)Paxos algorithm and the needs of a real-world system. . . . the final system will be based on an unproven protocol "

The Designers of **Chubby** at Google

RAFT System Model

- Same as Paxos!
- **Timing, Synchrony and Networking:** Partially synchronous and fair loss links
  - Messages may be lost, re-ordered, and may take arbitrarily long to deliver but are eventually delivered (after re-trying enough times)
  - Nodes operate at **arbitrary speeds (can be very slow to respond)**
- **Only crash failures:** crash-recovery
  - Nodes can fail and may sebsequently recover
  - Nodes **do not attempt to subvert** the protocol, and messages are never corrupted
    - This means RAFT (similar to Paxos) does NOT consider Byzantine failures

RAFT Basics

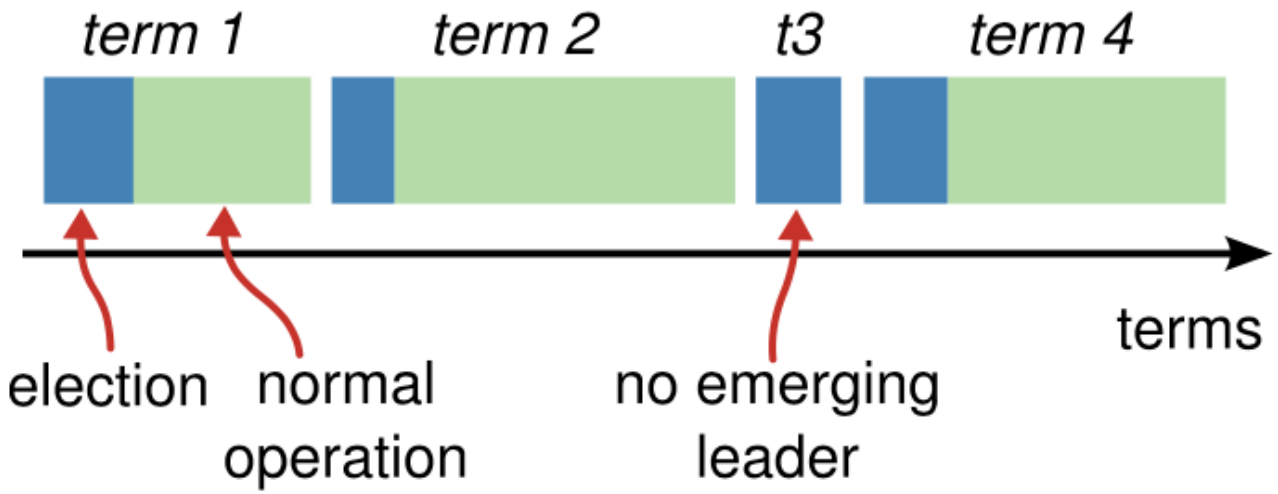
- RAFT is a leader-driven consensus protocol
  - The leader accepts client requests and replicates them on other replicas
  - The leader tells other replicas when it is safe to execute (commit) log entries
- In the normal operation, one of the replicas is a **leader**
  - A leader is chosen through an **election** mechanism

RAFT Properties

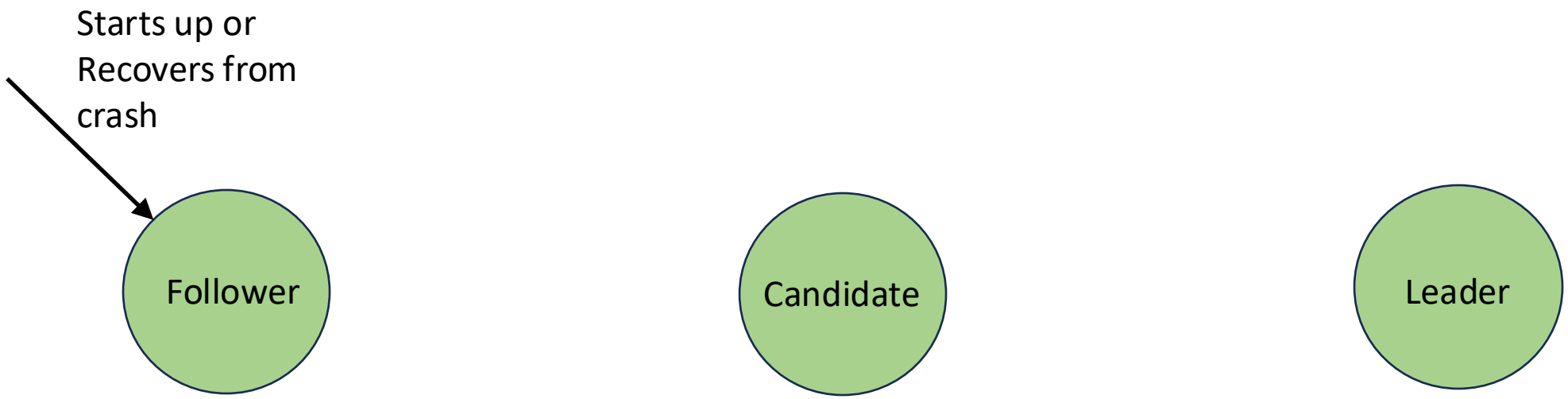
- **Leader Election**: a leader must be chosen when the existing one fails
  - VoteRequest RPC from the leader to other replicas
- **Log Replication**: the leader must accept *operation requests* from clients and replicate them across the cluster – forcing the other logs to agree with its own!
  - The leader sends ReplicateLog RPC to other replicas
- **Safety**: if any replica has committed a particular log entry to its state machine, then no other replica may apply (i.e. commit) a different command for the same log index

RAFT Basics: Terms

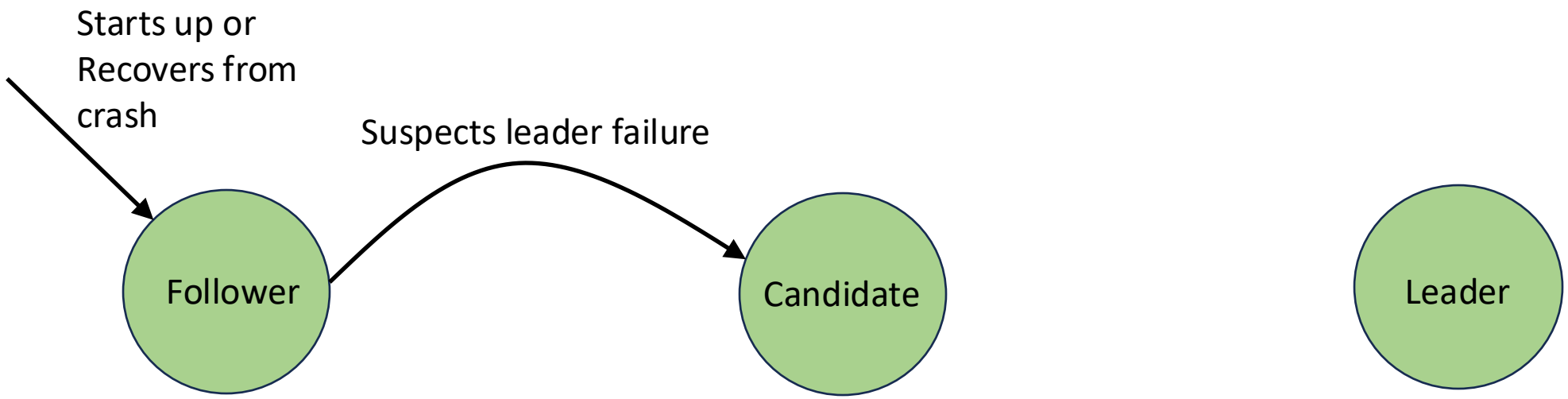
- Raft divides time into **terms** of arbitrary length, each is numbered with consecutive integers
- Term is used as a *logical clock*
  - allow replicas to detect obsolete information such as stale leaders



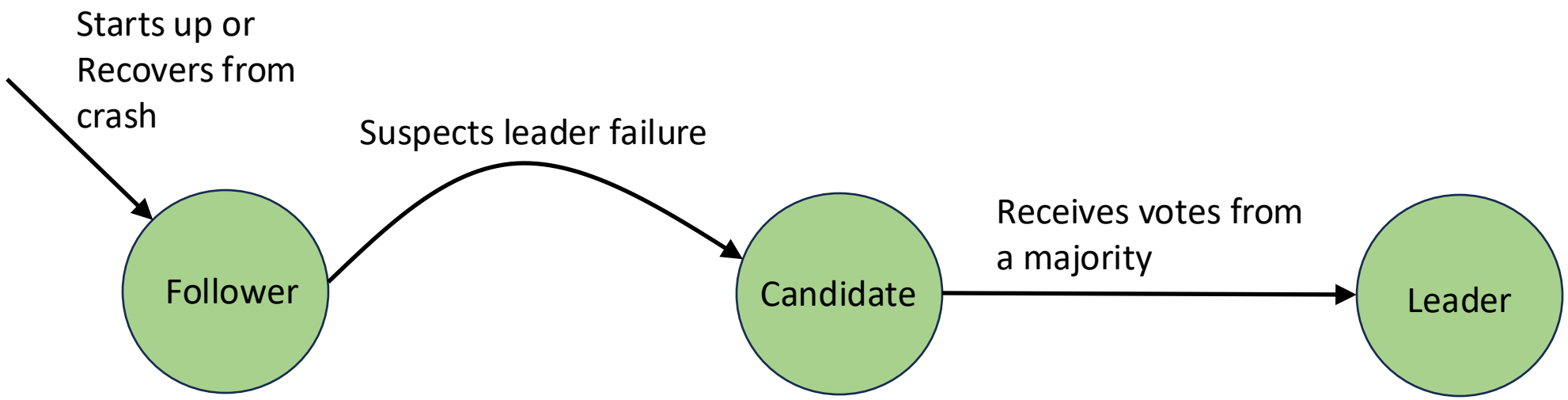
RAFT Basics: Replica States



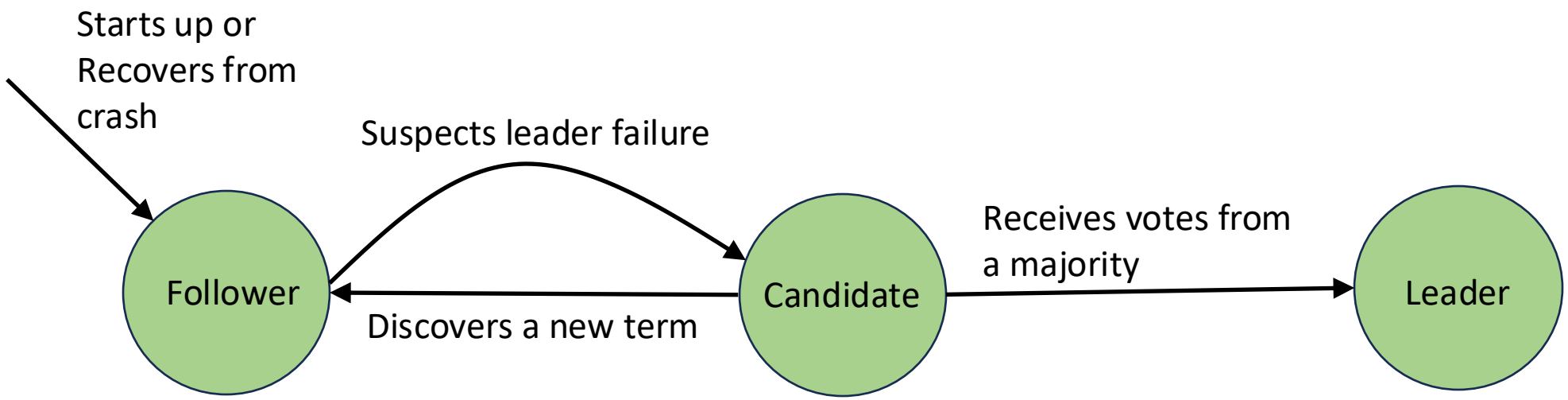
RAFT Basics: Replica States



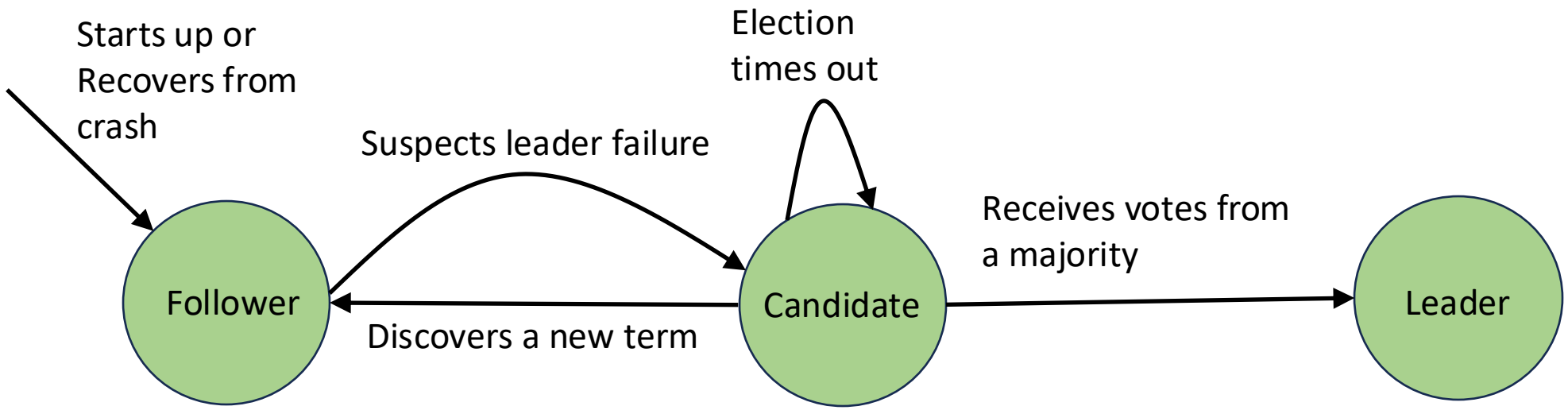
RAFT Basics: Replica States



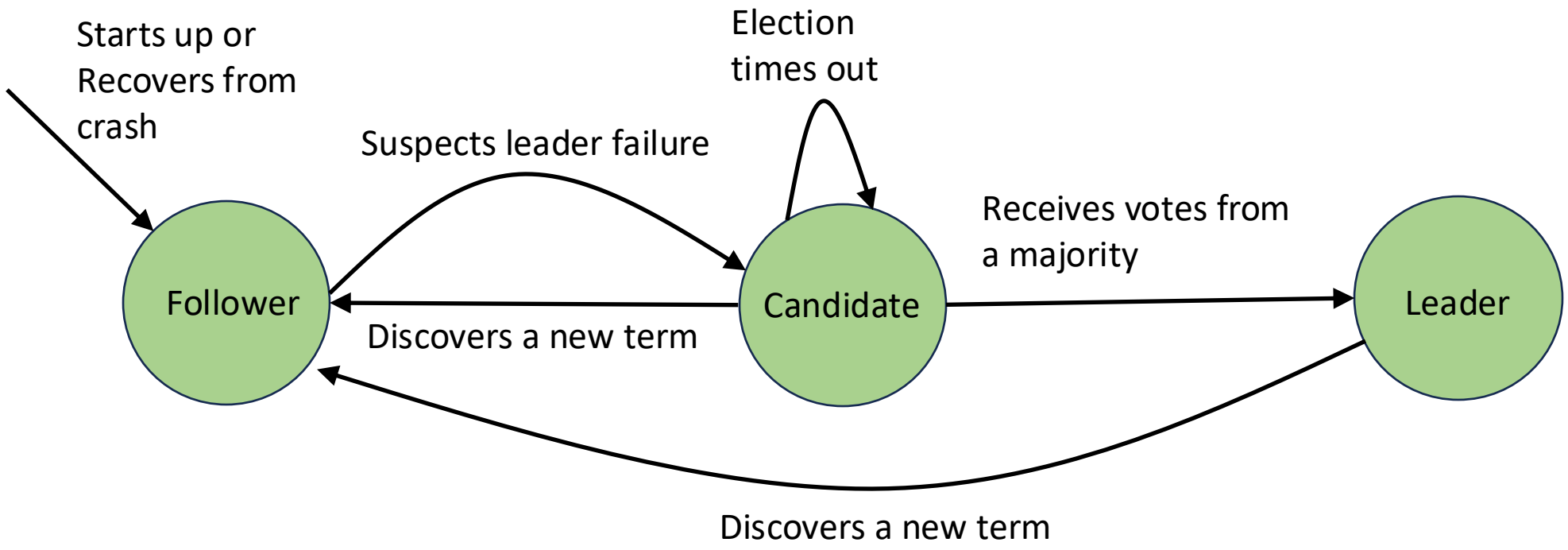
RAFT Basics: Replica States



RAFT Basics: Replica States



RAFT Basics: Replica States



RAFT Basics: Leader election

- Replicas start up as followers
- Followers expect to receive RPCs from leaders or candidates
- If **election timeout** elapses with no RPCs:
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically 150-500ms
- Leaders must send **heartbeats** to maintain authority

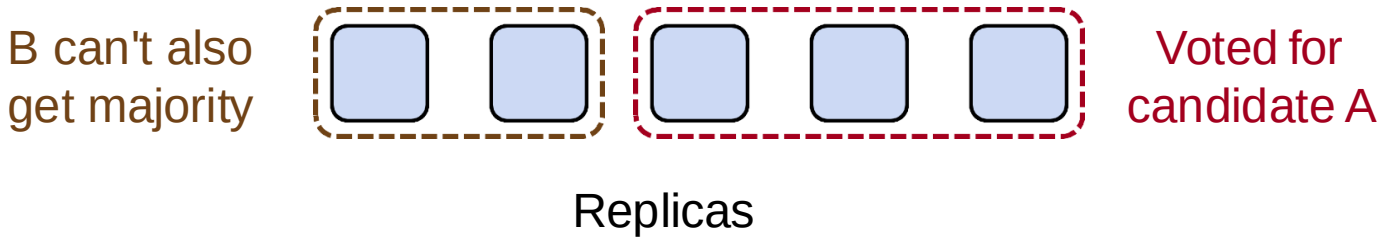
RAFT Basics: Leader election

Upon election timeout:

- **Increment current term**
- **Change to Candidate state**
- **Vote for self**
- **Send RequestVote RPCs to all other replicas, wait until either:**
  1. Receive votes from majority of replicas:
     - Become leader
     - Send ReplicateLog heartbeats to all other replicas
  2. Receive RPC from valid leader:
     - Return to follower state
  3. No-one wins election (election timeout elapses):
     - Increment term, start new election after a random timeout

RAFT Basics: Leader election

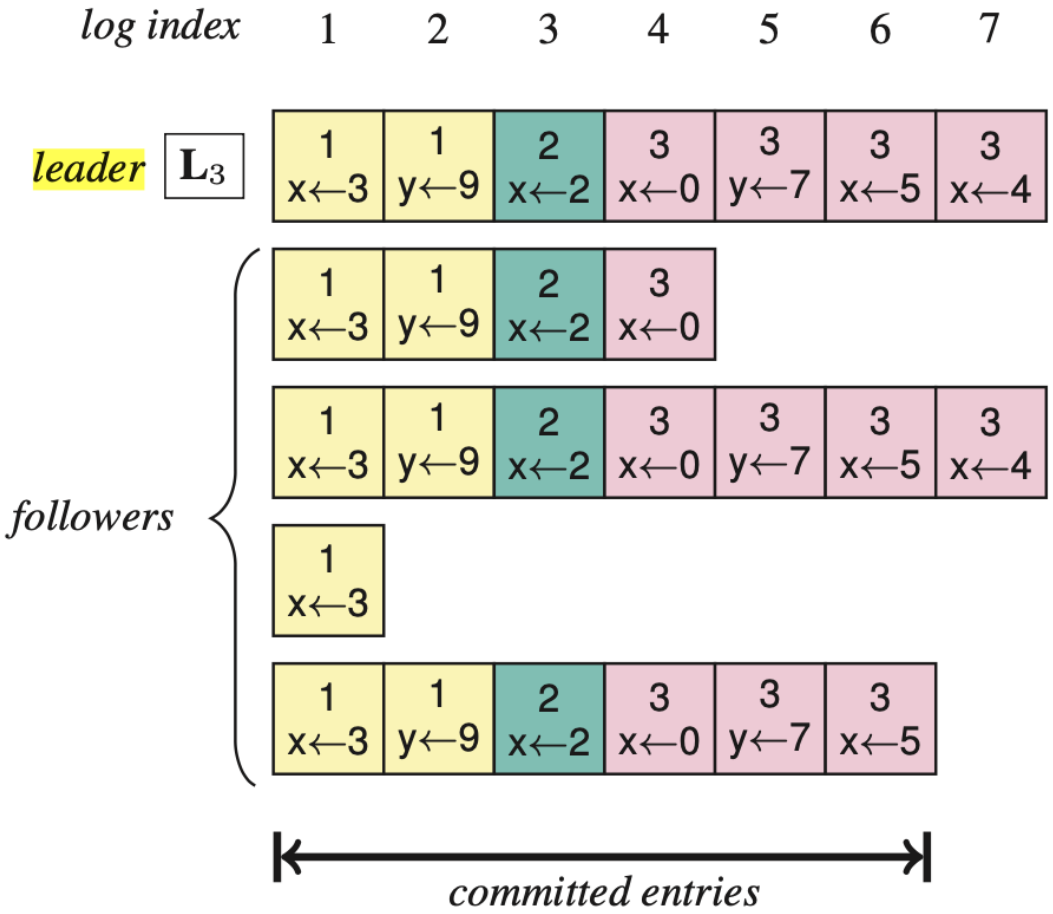
- **Safety: allow at most one winner per term**
  - Each replica gives out only one vote per term (persist on disk) – on a first-come-first-served basis
  - Two different candidates can't accumulate majorities in same term



- **Liveness: a candidate must eventually win**
  - It is possible that multiple (two or more) replicas announce their candidacy simultaneously and none of them achieves a majority:
    - In that case, the replicas each chooses an election timeout **randomly** from, e.g., 150-300ms range
    - One replica usually times out first and wins the election before others wake up

RAFT Basics: Log Replication

- At any given time, the log of a replica contains both committed and uncommitted entries



- **Log entry = index, term, operation**
- **Log stored on persistent storage (disk); so it survives crashes**

RAFT Basics: Normal Operation

- Client sends operations
- Leader appends operations to its log
- Leader sends ReplicateLog RPCs to followers
- Once a new entry safely committed:
  - Leader applies command to its state machine, returns result to client
- Catch up followers in background:
  - Leader notifies followers of committed entries in subsequent ReplicateLog RPCs
  - Followers apply committed commands to their state machines
- Performance is optimal in common case:
  - One successful RPC to any majority of replicas

RAFT Details: Announcing Candidacy

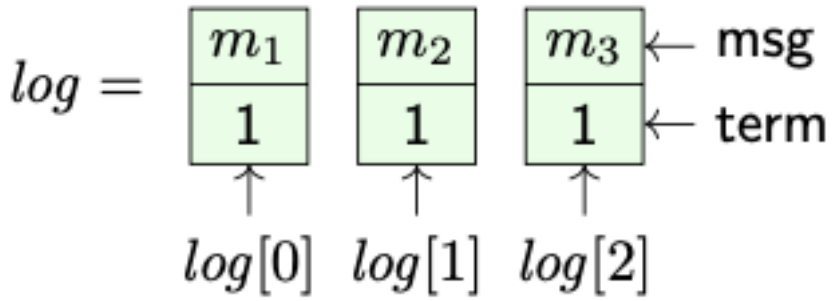
- Each node maintains a set of state in persistent storage:
  - CurrentTerm, lastVoted, Log
- A follower announces its candidacy if:
  - It detects that the leader has crashed (Remember, failure detection is not perfect!)
- A follower first becomes a candidate, then:
  - Picks the next higher term number (one larger than any term it has seen)
  - Votes for itself (just like any rational leader candidate would do!)
  - Sends a VoteRequest RPC: VoteRequest (nodeID , currentTerm , logLength, lastTerm)
    - nodeID: candidate's node ID
    - currentTerm: term number picked by the candidate
    - logLength: candidate's log length (why?)
    - lastTerm: the term number of the last entry in the candidate's log

RAFT Details: Initialisation & Announcing Candidacy

```
on initialisation do
  currentTerm := 0; votedFor := null                          // stored on the disk – to preserve in case of a crash
  log := ⟨⟩;   commitLength := 0                              //  stored on the disk
  currentRole := follower;   currentLeader := null            // in-memory storage – no need to recover after a crash
  votesReceived := {}; sentLength := ⟨⟩; ackedLength := ⟨⟩    // in-memory storage
end

on recovery from crash do
  currentRole := follower; currentLeader := null
  votesReceived := {}; sentLength := ⟨⟩; ackedLength := ⟨⟩
end

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 := RequestVote(nodeId , currentTerm , log.length, lastTerm)
  for each node in nodes:
    send msg to node
  start election timer
end
```



RAFT Details: Follower Voting

- A follower either sends a positive (accepts candidacy) or a negative vote (rejects candidacy) as a response to VoteRequest from a candidate.
- A candidate C is rejected by a follower F:
  - C's term number is smaller than F's
  - C's log is behind (shorter than F's) or its last log entry is from an older term than F's last log entry
- Otherwise, F sends a positive vote (on a first-come-first-served basis)
- In a response, the follower sends its own term number (which can be larger than the candidate's if the vote is negative)
- A follower sends at most one positive vote during a given term number
- Why reject a candidate whose log is behind?

RAFT Details: Follower Voting

```
VoteRequest(cId, cTerm, cLogLength, cLogTerm) at node nodeId do
   if cTerm > currentTerm then
      currentTerm := cTerm; currentRole := follower
      votedFor := null
   end if

   lastTerm := 0
   if log.length > 0 then
      lastTerm := log[log.length-1].term;
   end if

   logOk := (cLogTerm > lastTerm) OR (cLogTerm == lastTerm AND cLogLength ≥ log.length)
   if cTerm == currentTerm AND logOk AND votedFor in {cId , null} then
      votedFor := cId
      return (nodeId, currentTerm, true) to node cId    // send positive vote
   else
      return (nodeId, currentTerm, false) to node cId   // send negative vote
   end if
end
```

RAFT Details: Collecting Votes

- If a candidate receives positive votes from **a majority of replicas** (including the candidate's own vote) *for the same term*, then the candidate becomes leader
- Multicasts a message to announce its leadership to all replicas
- If any of the followers respond with a larger term number than the candidate's, then the candidate backs off immediately and sets itself as a follower
- If there is no winner for a term, then the candidate times out for some random amount of time
  - Randomness is needed to ensure that multiple candidate do not simultaneously send VoteRequests

RAFT Details: Collecting Votes

```
on receiving (VoteResponse, voterId, term, granted) at nodeId do
  if currentRole == candidate AND term == currentTerm AND granted then   // if the vote is granted and the term is correct
    votesReceived := votesReceived ∪ {voterId }                  // Add the voter to the received voted
    if |votesReceived| ≥ [(|nodes| + 1)/2] then                  // Do we have a majority of peers voting for our candidacy ?
      currentRole := leader; currentLeader := nodeId
      cancel election timer
      for each follower in nodes except {nodeId} do    // Tell everyone that this node is the new leader
        sentLength[follower] := log.length
        ackedLength[follower] := 0
        replicateLeaderLog(nodeId,follower)              // Send ReplicateLog messages
      end for
    end if
  else if term > currentTerm then                 // Is there another replica with a higher term out there ?
    currentTerm := term                            // Update the current term to be the higher term number
    currentRole := follower                        // Forget about the candidacy (and the election) and become a follower
    votedFor := null                               // Forget any vote casted in a previous term
    cancel election timer
  else set election timer                       // No winners, so timeout for a random period and try again
  end if
end
```

RAFT Details: Processing Client Requests

- If a client message (requesting for an operation) arrives at the leader, the leader then:
  - adds the message to its own log
  - sends a ReplicateLog() RPC to other replicas
- The leader must periodically send a ReplicateLog() RPC to all replicas even when there are no new client messages (as heartbeat messages)
- If a client message arrives at a non-leader replica, it must forward it to the current leader

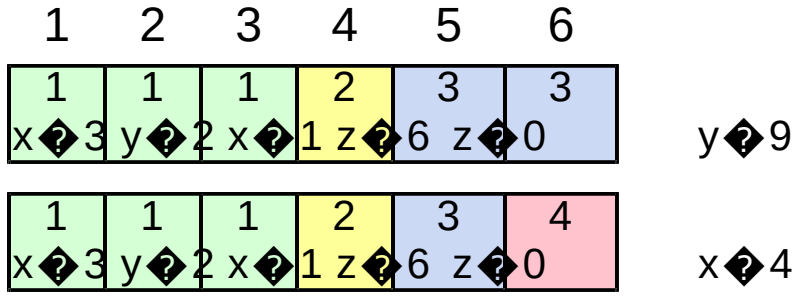
RAFT Details: Processing Client Requests

```
on receiving client request msg at node nodeId do
  if currentRole == leader then
    append the record (msg : msg, term : currentTerm) to log     // append the message to the log
    ackedLength[nodeId] := log.length                            // set myself as having acknowledged to msg
    for each follower in nodes except {nodeId} do
      replicateLeaderLog(nodeId,follower)    // local function (defined later)
    end for
  else
      forward the request to currentLeader
  end if
end on

periodically at node nodeId do     // periodically send ReplicateLog RPCs (as a heartbeat message)
  if currentRole = leader then
    for each follower in nodes except {nodeId} do
      replicateLeaderLog(nodeId,follower)
    end for
  end if
end do
```

RAFT Details: Leader Replicating its Log

- **If log entries on different replicas have same index and term:**
  - They store the same command
  - The logs are identical in all preceding entries



- **If a given entry is committed, all preceding entries are also committed**

RAFT Details: Leader Replicating its Log

- The leader replicates its log entries to other replicas by sending a **ReplicateLog** RPC message to each replica
- The leader keeps track of a *prefix* and *suffix* for each follower:
  - prefix points to the last entry that is common to the leader and the follower
    - The protocol guarantees that the logs are identical in all preceding entries
  - suffix is the list of entries that the leader contributes to the follower

RAFT Details: Leader Replicating its Log

replicateLeaderLog is called on the leader whenever there is a new message in the log, and also periodically. If there are no new messages, *suffix* is the empty list. LogRequest messages with *suffix* = ⟨⟩ serve as heartbeats, letting followers know that the leader is still alive.

```
function replicateLeaderLog(leaderId, followerId)
    prefixLen := sentLength[followerId]
    suffix := ⟨log[prefixLen], log[prefixLen + 1], . . . , log[log.length − 1]⟩
    prefixTerm := 0

    if prefixLen  > 0 then
        prefixTerm := log[prefixLen – 1].term
    end if

    send ReplicateLog(leaderId, currentTerm, prefixLen, prefixTerm, commitLength, suffix) to followerId
end function
```

RAFT Details: Followers processing ReplicateLog

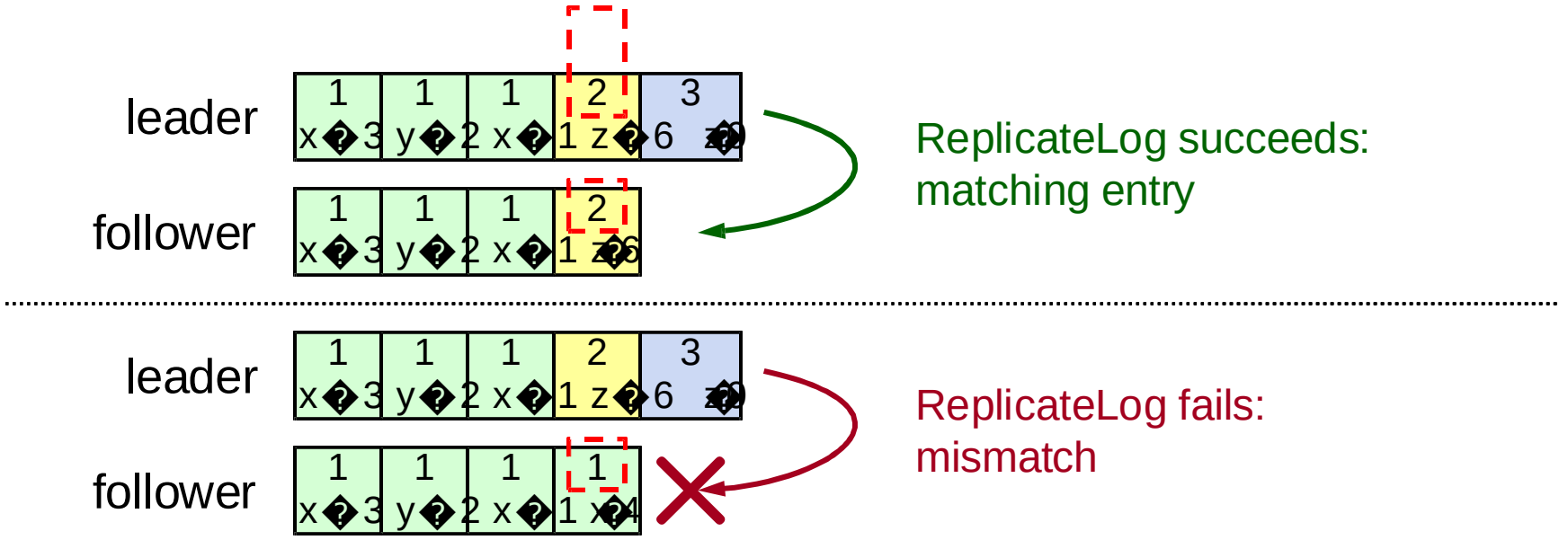
```
on receiving ReplicateLog(leaderId,term,prefixLen,prefixTerm, leaderCommitIndex, entries) at node nodeId do
  if term > currentTerm then
    currentTerm := term; votedFor := null
    cancel election timer
  end if

  if term == currentTerm then
    currentRole := follower; currentLeader := leaderId
  end if

  logOk := (log.length ≥ prefixLen) AND (prefixLen == 0 OR log[prefixLen-1].term == prefixTerm)   // Log consistency check
  if term == currentTerm AND logOk then
    appendEntries(prefixLen, leaderCommitIndex , suffix )
    ack := prefixLen + suffix.length
    return ReplicateLogResponse(nodeId, currentTerm, ACK=true) to leaderId    // Logs are consistent so return ACK
  else
    return ReplicateLogResponse(nodeId, currentTerm, ACK=false) to leaderId   // Logs are inconsistent, we need repairing!
  end if
end
```

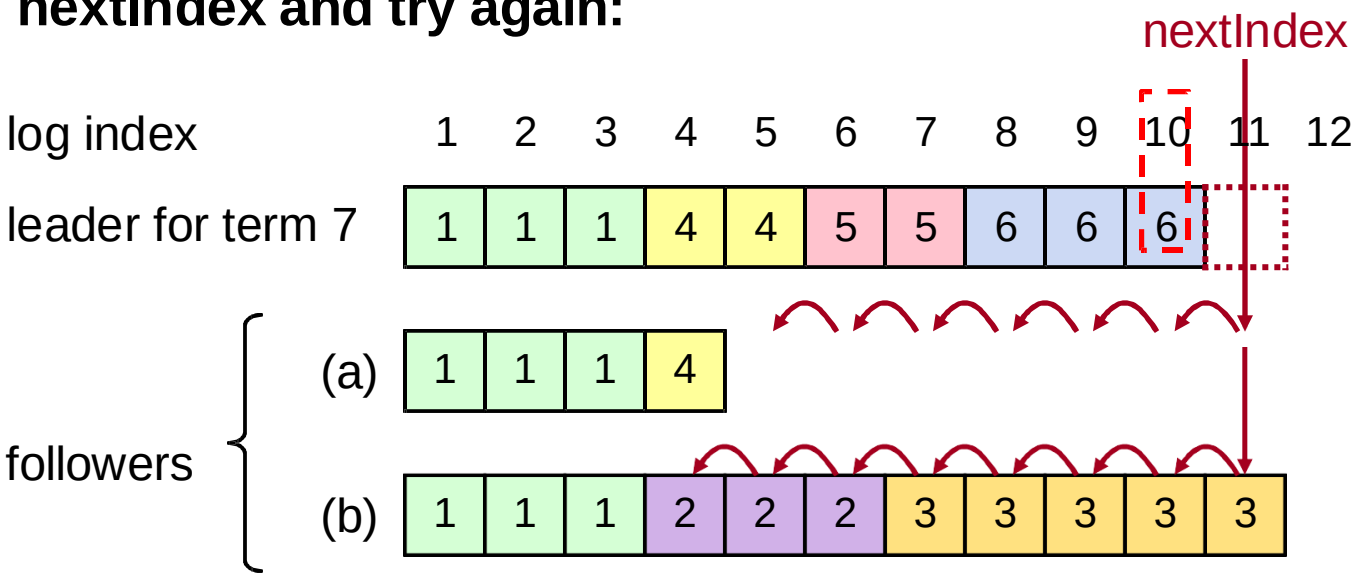
RAFT Details: ReplicateLog Consistency Check

- Each ReplicateLog RPC contains a **prefix**: the index, term of entry preceding new ones (**suffix**)
- Follower must contain matching prefix entry; otherwise it rejects request
- Implements an induction step, ensures coherency

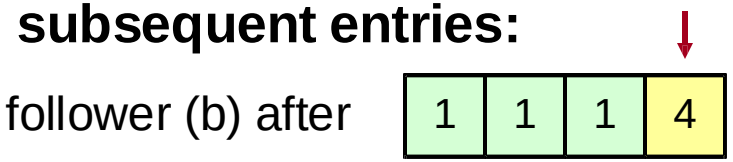


RAFT Details: Repairing Follower Logs

- **Leader keeps nextIndex for each follower:**
  - Index of next log entry to send to that follower
- **When ReplicateLog consistency check fails, decrement nextIndex and try again:**



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

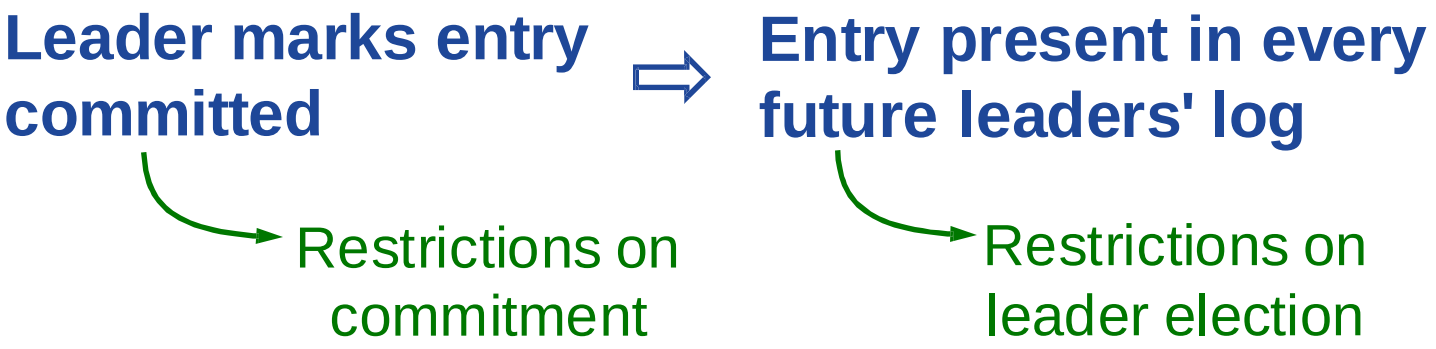


RAFT Details: Committing entries

- Leader ensures that log entries are reliably replicated across a majority of followers
- The leader tracks ACKs for each log entry and, when a majority of followers ACK a given entry, the leader can safely assume that this log entry is "committed”
  - Leader **executes the committed log entry** (i.e., the operation stored in that entry) in its local state machine and **returns the result back to the client**
- This commitment is reflected by updating the leader’s leaderCommitIndex—an index that denotes the highest log entry that is guaranteed to be committed across a majority
- Leader informs all followers of the new commit status by sending the updated leaderCommitIndex in subsequent ReplicateLog RPCs

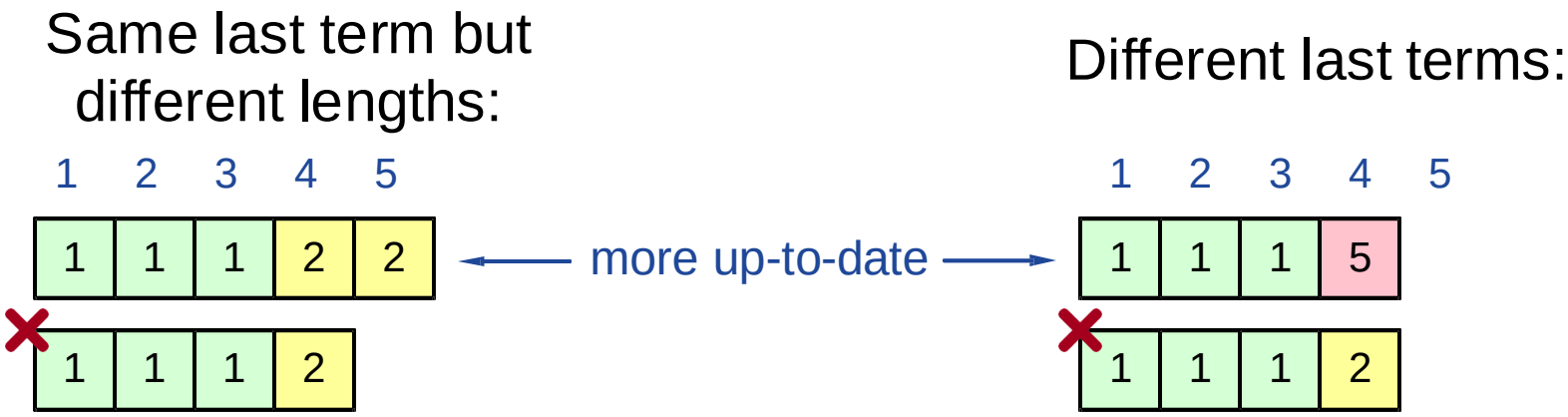
RAFT: Safety Requirements

- Any two committed entries at the same index must be the same.
- Committed entries can not be removed later by, for instance, another leader!
  - It will be eventually present in every future leaders' log

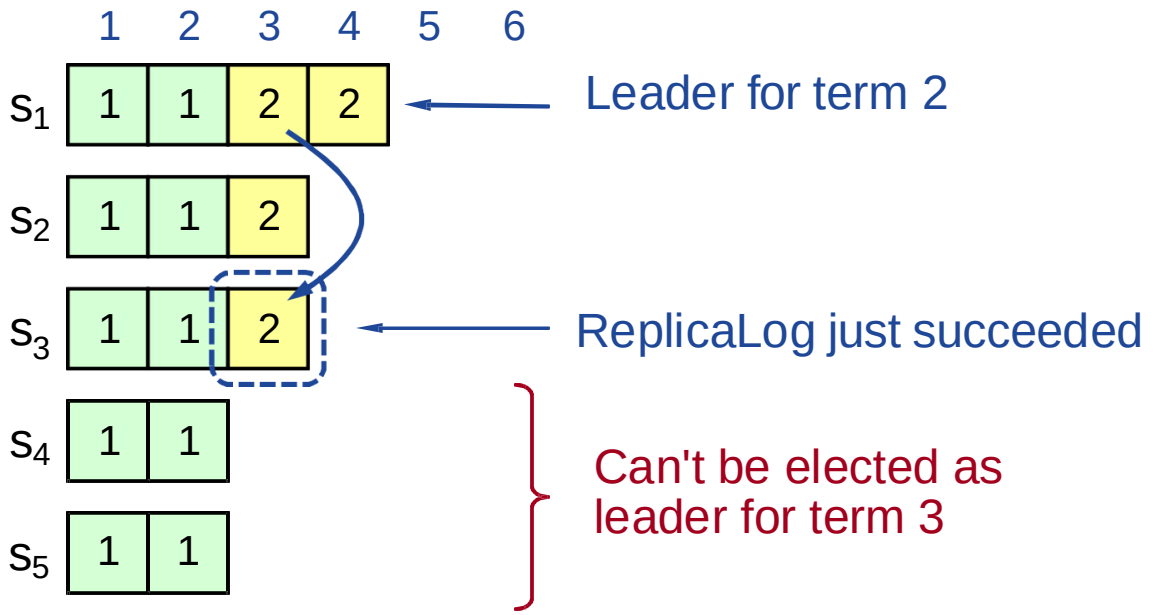


RAFT: Safety Rules

- **During elections, candidate must have most up-to- date log among electing majority:**
  - Candidates include log info in RequestVote RPCs (length of log & term of last log entry)
  - Voting replica denies vote if its log is more up-to-date:



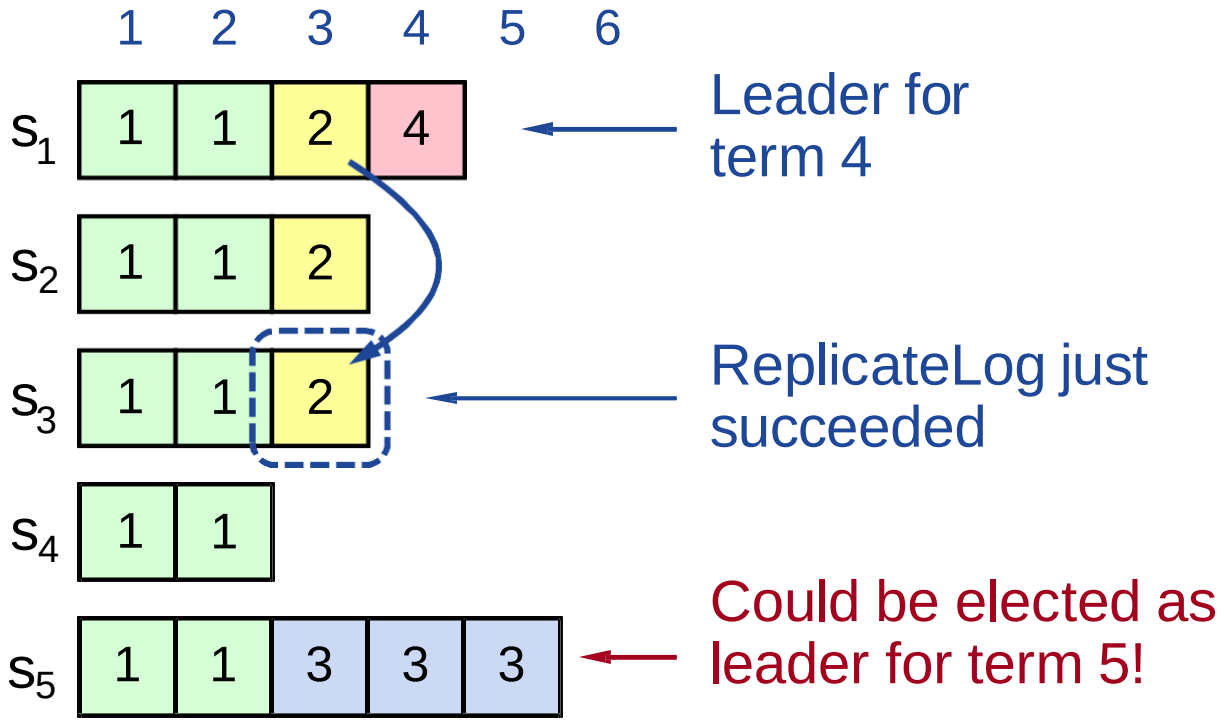
RAFT: Safety Rules



- **Majority replication makes entry 3 safe:**

Leader marks entry committed ⇨ **Entry present in every future leaders' log**

RAFT: Safety Rules

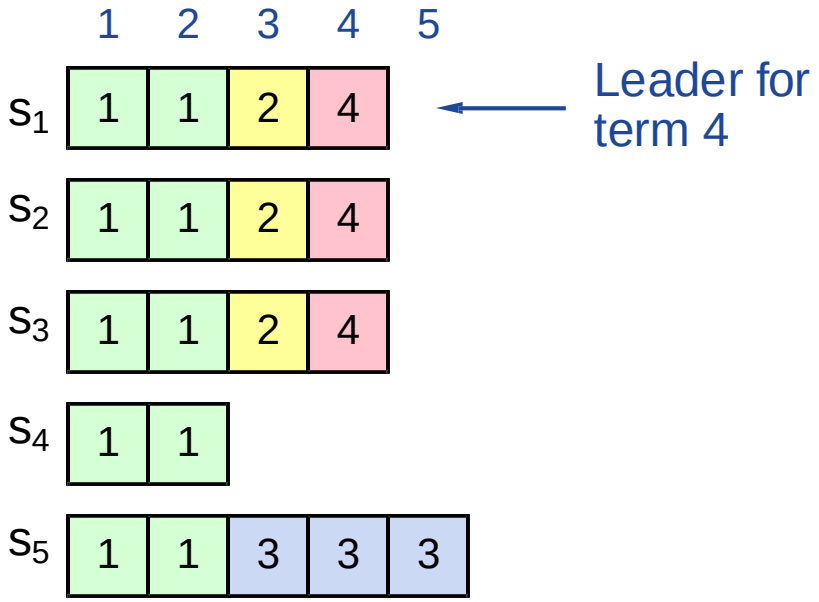


- **Entry 3 not safely committed:**

Leader marks entry committed ⇨ **Entry present in every future leaders' log**

RAFT: Safety Rules

- New leader may not mark old entries committed until it has committed an entry from its current term.
- 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

RAFT: Liveness Requirements

- Remember, liveness means the system continues to make progress (i.e., does not halt), which means:
  - A new leader is eventually elected if the current leader fails
  - Client requests are eventually processed and committed
- The system maintains liveness as long as:
  - multicastDelay $\ll$ electionTimeout $\ll$ ATBF
    - ATBF: average time between failures for a single replica
  - Liveness is lost temporarily during elections because there is no leader (client requests are rejected)

Summary

- Introduced another consensus protocol, RAFT, to implement “State Machine Replication” (SMR)
- RAFT is a leader-based consensus protocol with leader election and log replication stages
- RAFT and Paxos are equivalent in terms of safety, liveness, and fault tolerance within a non-Byzantine, crash-only model