Failures, Elections, and Raft

Distributed Banking

SFO

add interest based
on current balance

PVD

deposit $1000

Synchronous vs. Asynchronous

- Execution speed
  - synchronous: bounded
  - asynchronous: unbounded
- Message transmission delays
  - synchronous: bounded
  - asynchronous: unbounded
- Local clock drift rate:
  - synchronous: bounded
  - asynchronous: unbounded

Failures

- Omission failures
  - something doesn’t happen
    - process crashes
    - data lost in transmission
    - etc.
- Byzantine (arbitrary) failures
  - something bad happens
    - message is modified
    - message received twice
    - any kind of behavior, including malicious
- Timing failures
  - something takes too long

Detecting Crashes

- Synchronous systems
  - timeouts
- Asynchronous systems
  - ?
- Fail-stop
  - an oracle lets us know

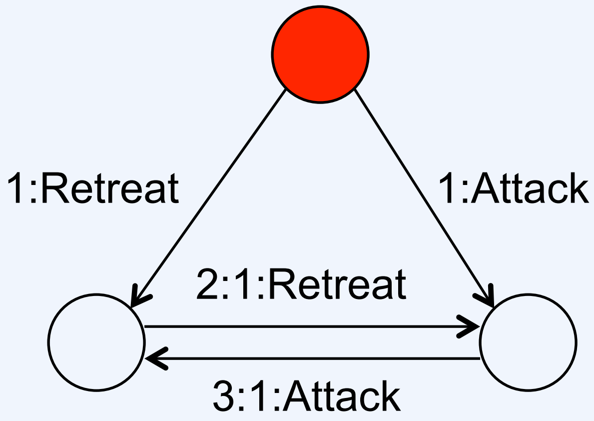
Consensus

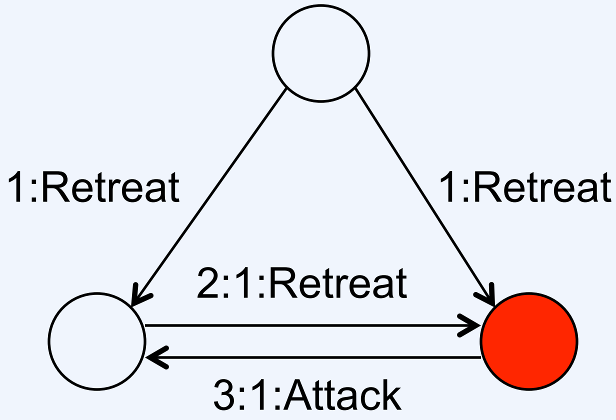
- Setting: a group of processes, some *correct*, some faulty
- Two primitives, *propose*(*v*) and *decide(v)*
- If each correct process proposes a value:
  - Termination: every correct process eventually decides on a value
  - Agreement: if a correct process decides *v*, then all processes eventually decide *v*
  - Integrity: if a correct process decides *v*, then *v* was previously proposed by some process

Consensus

- Variations on the problem, depending on assumptions
  - Synchronous vs asynchronous
  - Fail-stop vs crash/omission failures vs Byzantine failures
- Equivalent to reliable, totally- and causally-ordered broadcast (Atomic broadcast)

Byzantine Agreement Problem



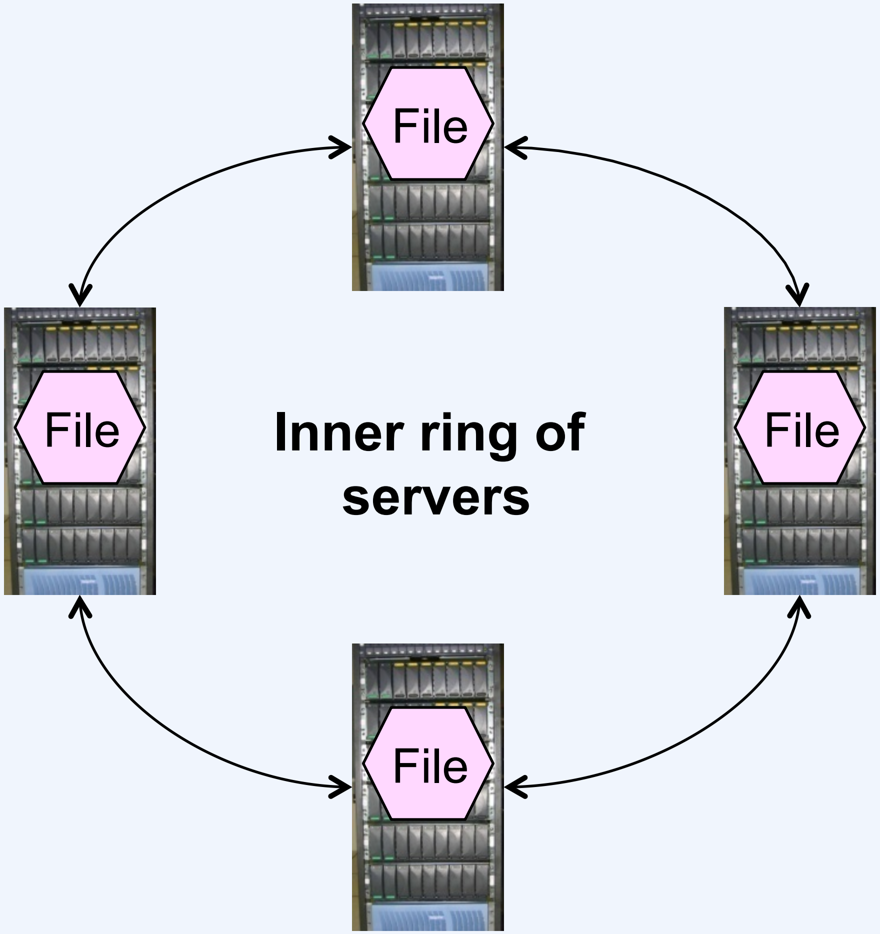


- Will cover in a future lecture

Impossibility of Consensus

- There is no deterministic protocol that solves consensus in a message-passing asynchronous system in which at most one process may fail by crashing.
  - Solve means to satisfy safety and liveness
  - Famous result from Fisher, Lynch, Paterson (FLP), 1985
- There is hope, though
  - Introducing (some) synchrony – timeouts
  - Introducing randomization
  - Introducing failure detectors

Oceanstore: Replicated Primary Replica



Inner ring of servers

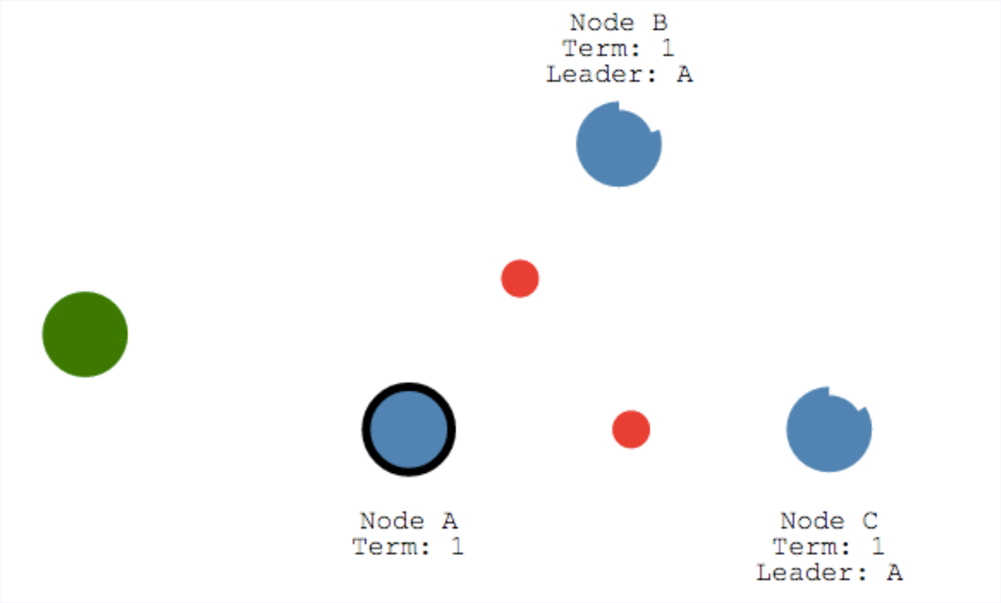
State-Machine Replication

- An approach to dealing with failures by replication
- A data-structure with deterministic operations replicated among servers
- State consistent if every server sees the same sequence of operations
  - Multiple rounds of consensus
- Common algorithms (non-Byzantine):
  - Paxos [Lamport], Viewstamped Replication [Oki and Liskov], Zab [Junqueira et al.], Raft [Ongaro and Ousterhout]

Raft

- Proposed by Ongaro and Ousterhout in 2014
- Four components
  - Leader election
  - Log replication
  - Safety
  - Membership changes
- Assumes crash failures
- No dependency on time for safety
  - But depends on time for availability

Demo



http://thesecretlivesofdata.com/raft

Server States

- At any given time, each server is either:
  - Leader: handles all client interactions, log replication
    - At most 1 viable leader at a time
  - Follower: completely passive (issues no RPCs, responds to incoming RPCs)
  - Candidate: used to elect a new leader
- Normal operation: 1 leader, N-1 followers

Slides based on Raft slides by John Ousterhout

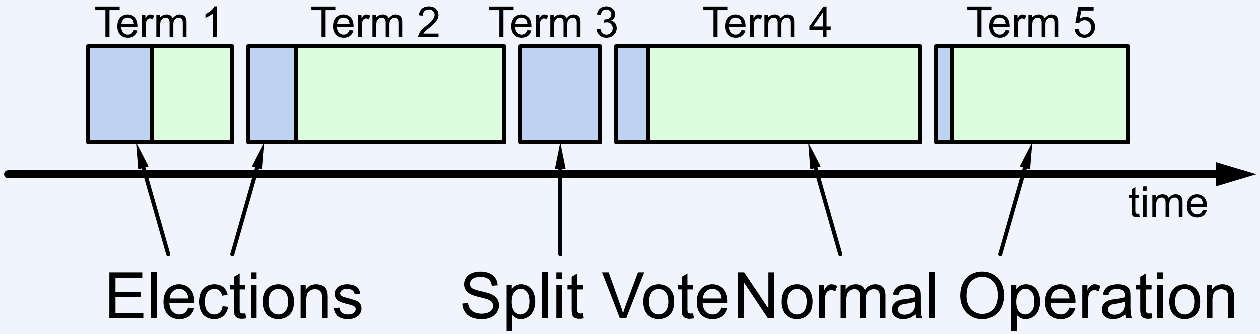
Life as a Leader

- Client sends command to leader
- Leader appends command to its log
- Leader sends AppendEntries RPCs to followers
- 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
- Crashed/slow followers?
  - Leader retries RPCs until they succeed
- Performance is optimal in common case:
  - One successful RPC to any majority of servers

Heartbeats and Timeouts

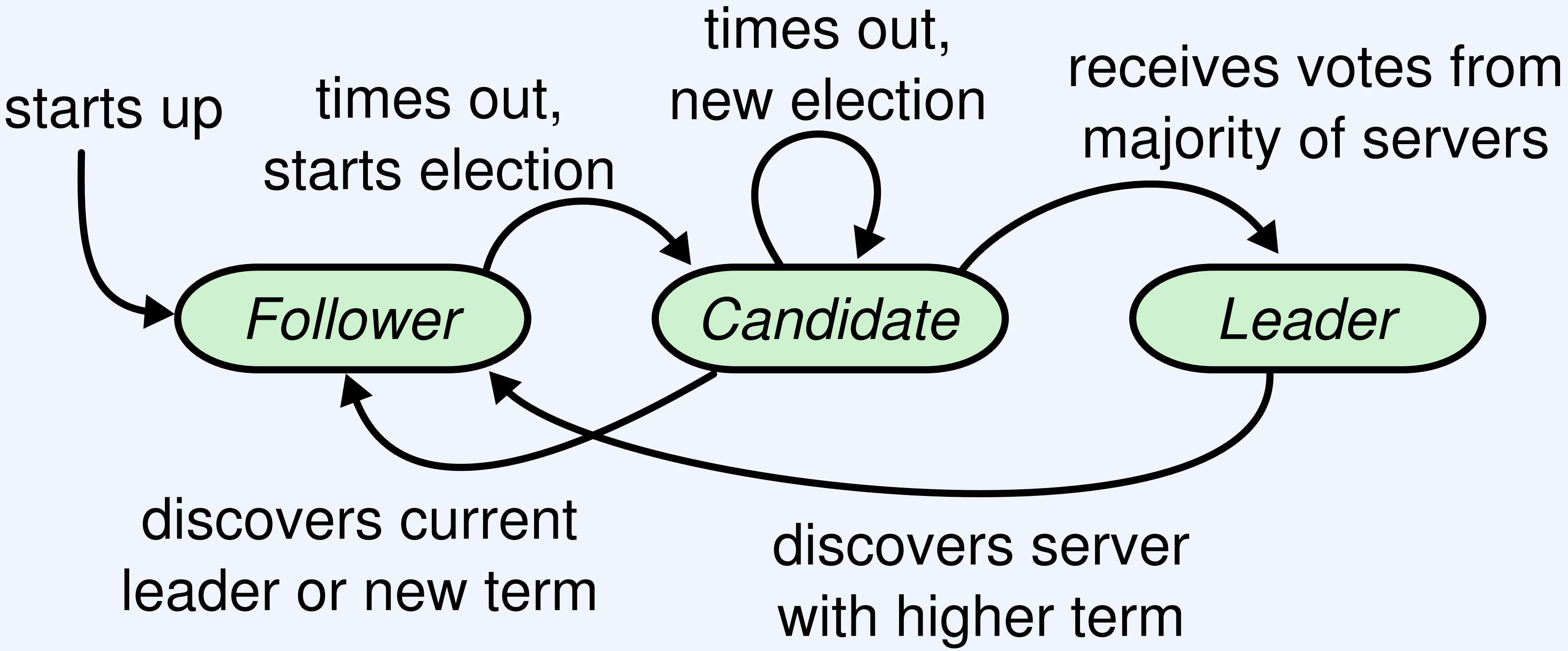
- Servers start up as followers
- Followers expect to receive RPCs from leaders or candidates
- Leaders must send heartbeats (empty AppendEntries RPCs) to maintain authority
- If electionTimeout elapses with no RPCs:
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically 100-500ms

Terms



- Time divided into terms:
  - – Election
  - – Normal operation under a single leader
- **At most 1 leader per term**
- Some terms have no leader (failed election)
- Each server maintains current term value
- Key role of terms: identify obsolete information

Server State Machine



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

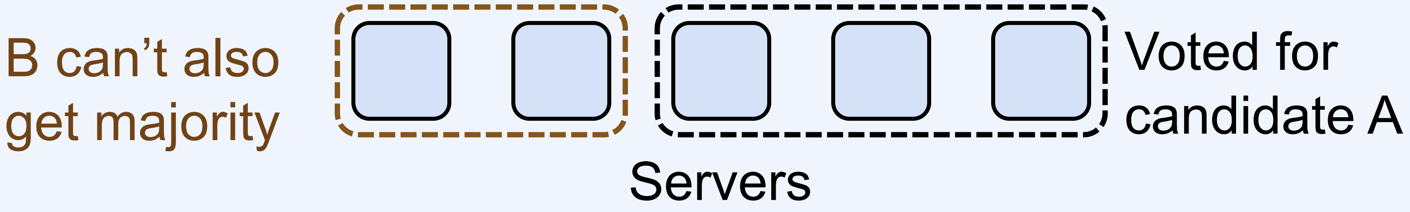
discovers server with higher term

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

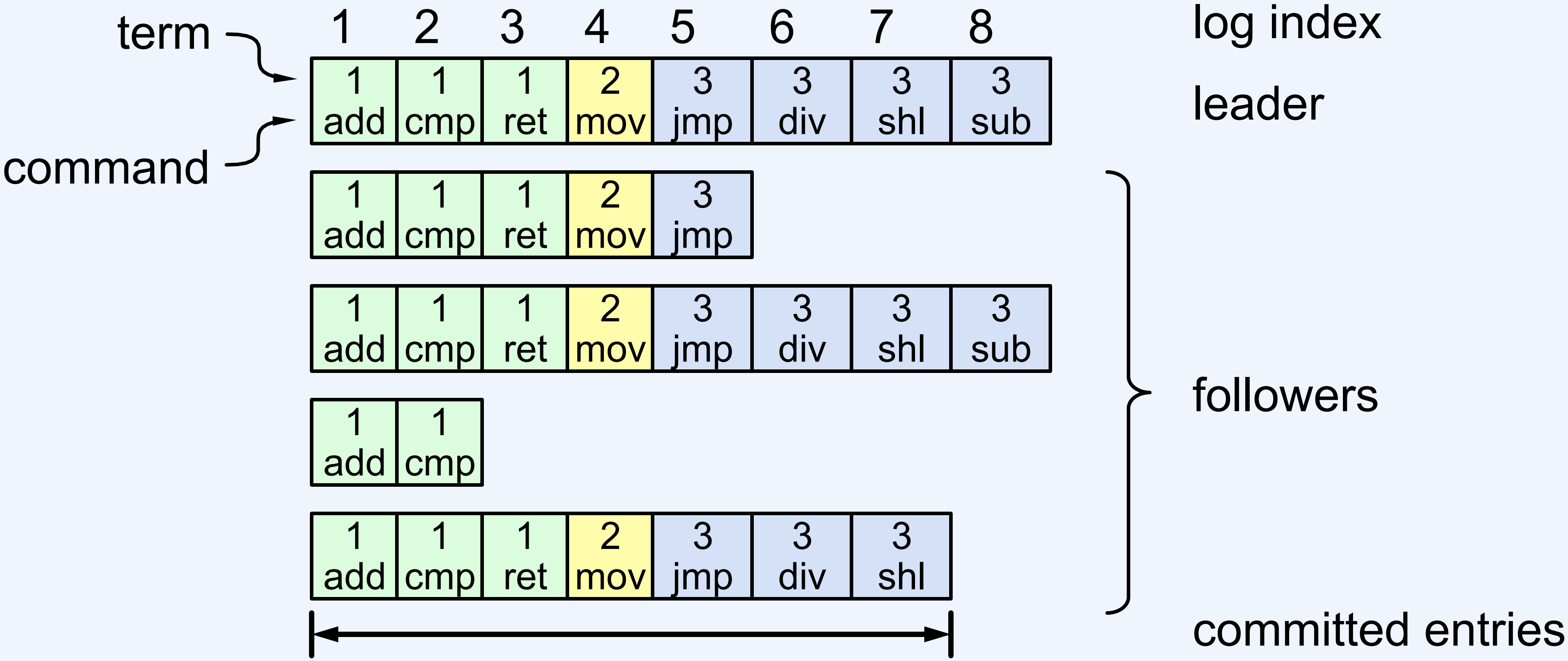
Elections, cont'd

- **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 >> broadcast time

Log Structure

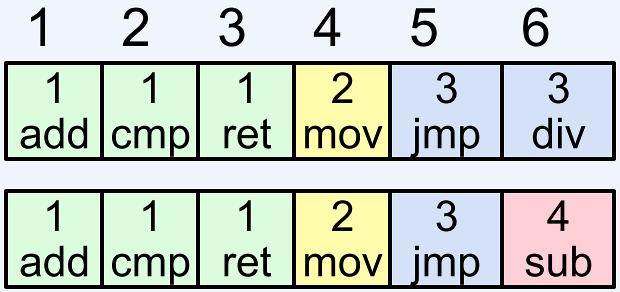


1	2	3	4	5	6	7	8
1 add	1 cmp	1 ret	2 mov	3 jmp	3 div	3 shl	3 sub
1 add	1 cmp	1 ret	2 mov	3 jmp			
1 add	1 cmp	1 ret	2 mov	3 jmp	3 div	3 shl	3 sub
1 add	1 cmp						
1 add	1 cmp	1 ret	2 mov	3 jmp	3 div	3 shl	

- Log entry = index, term, command
- Log stored on stable storage (disk); survives crashes
- Entry committed if known to be stored on majority of servers*
  - Durable, will eventually be executed by state machines

Log Consistency

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

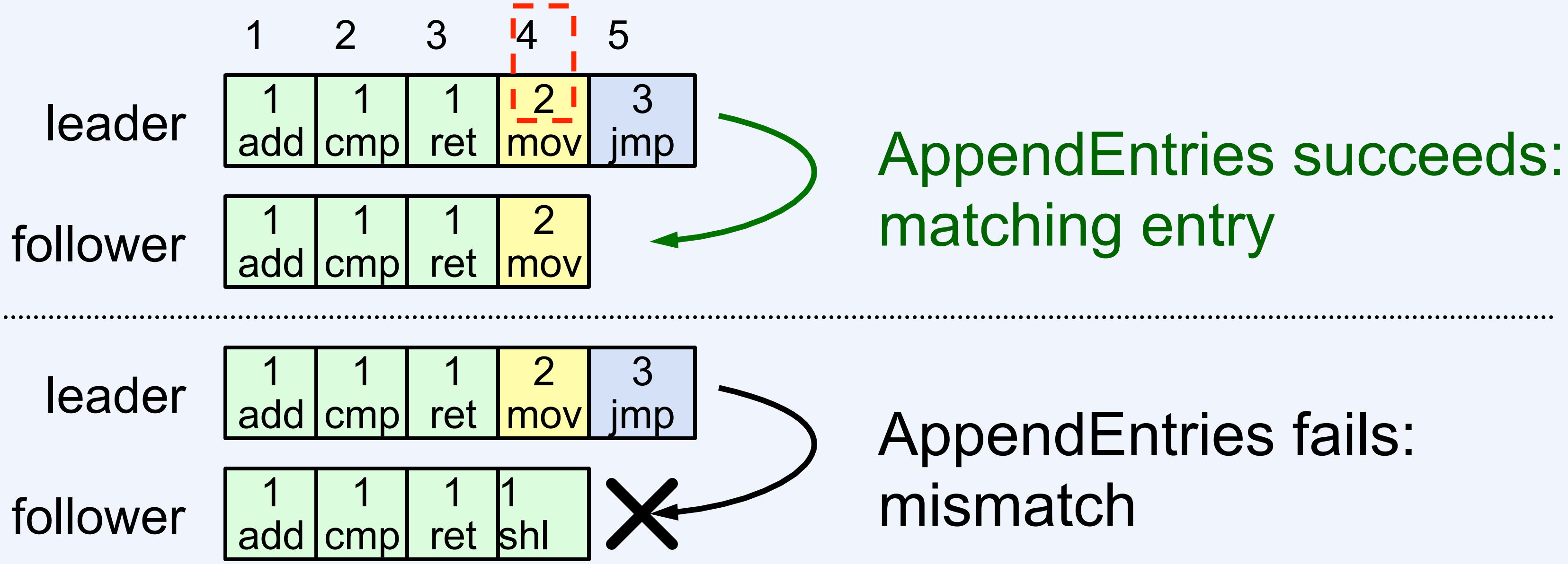


1	2	3	4	5	6
1 add	1 cmp	1 ret	2 mov	3 jmp	3 div
1 add	1 cmp	1 ret	2 mov	3 jmp	4 sub

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

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



	1	2	3	4	5
leader	1 add	1 cmp	1 ret	2 mov	3 jmp
follower	1 add	1 cmp	1 ret	2 mov	

AppendEntries succeeds: matching entry

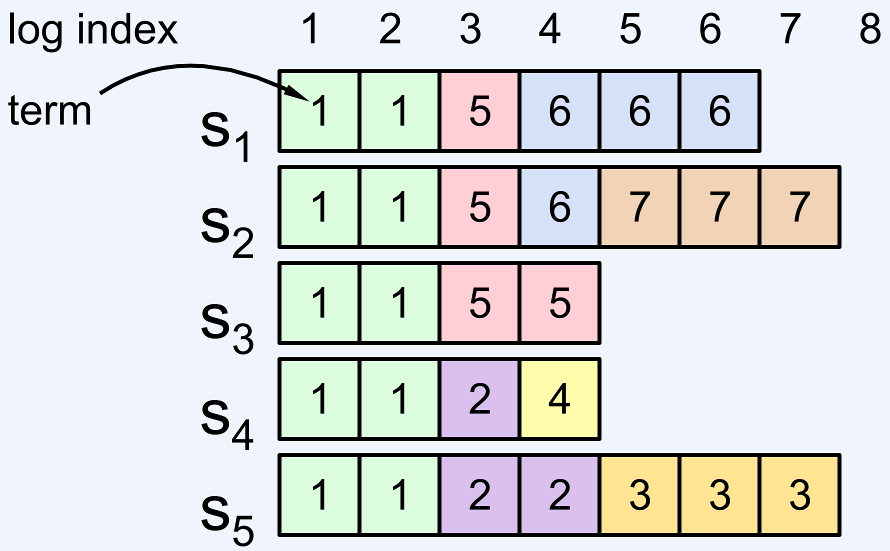
leader	1 add	1 cmp	1 ret	2 mov	3 jmp
follower	1 add	1 cmp	1 ret	1 shl	

AppendEntries fails: mismatch

Slides based on Raft slides by John Ousterhout

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:

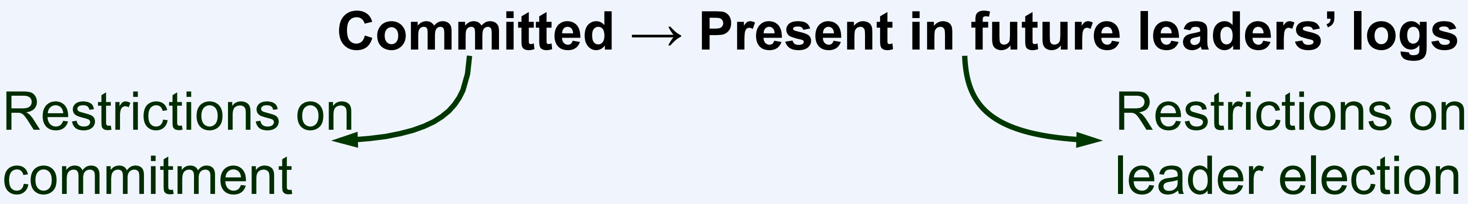


log index	1	2	3	4	5	6	7	8
s_1	1	1	5	6	6	6		
s_2	1	1	5	6	7	7	7	
s_3	1	1	5	5				
s_4	1	1	2	4				
s_5	1	1	2	2	3	3	3	

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
- This guarantees the safety requirement
  - 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



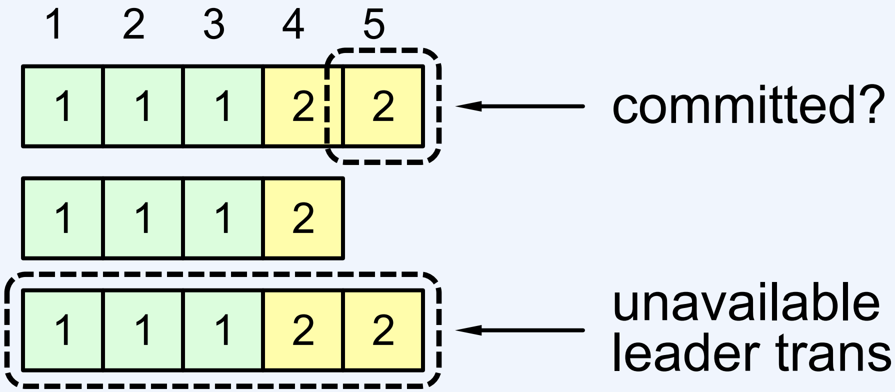
Committed → Present in future leaders’ logs

Restrictions on commitment

Restrictions on leader election

Picking the Best Leader

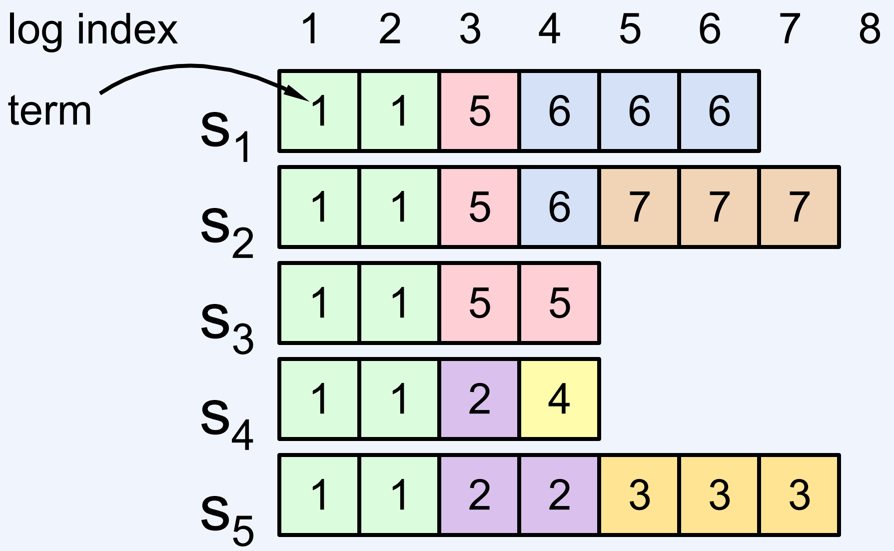
- Can't tell which entries are committed!



- 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":
    $(\text{lastTerm}_V > \text{lastTerm}_C) \,||$
    $(\text{lastTerm}_V == \text{lastTerm}_C) \,\&\&\, (\text{lastIndex}_V > \text{lastIndex}_C)$
  - **Leader will have "most complete" log among electing majority**

Picking the Best Leader

- Who can be elected for term 8?
  - – S1, S2, S3
  - – S4 and S5 cannot

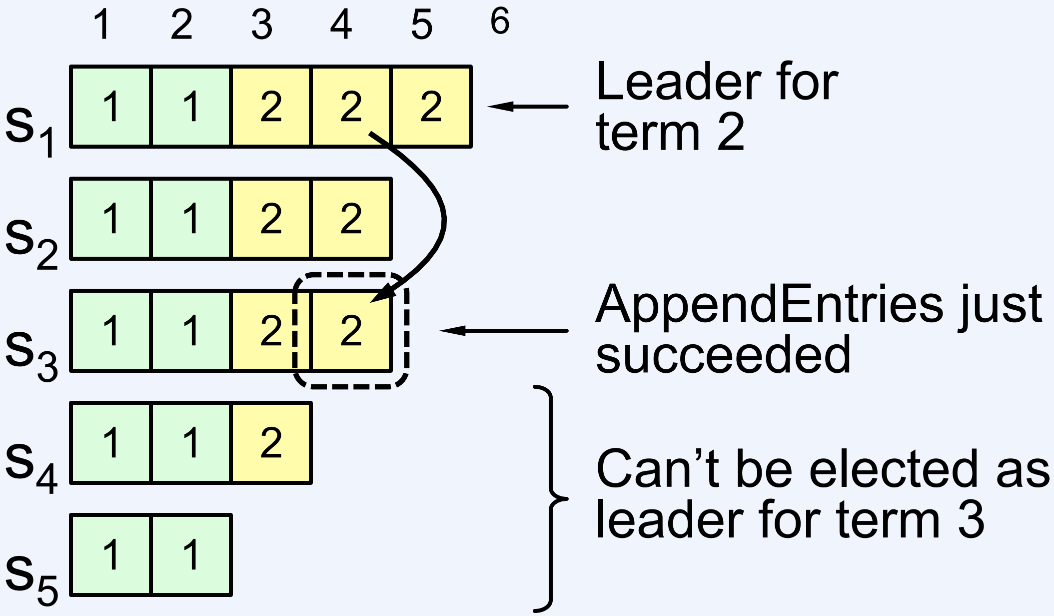


term

log index	1	2	3	4	5	6	7	8
S_1	1	1	5	6	6	6		
S_2	1	1	5	6	7	7	7	
S_3	1	1	5	5				
S_4	1	1	2	4				
S_5	1	1	2	2	3	3	3	

Committing Entry from Current Term

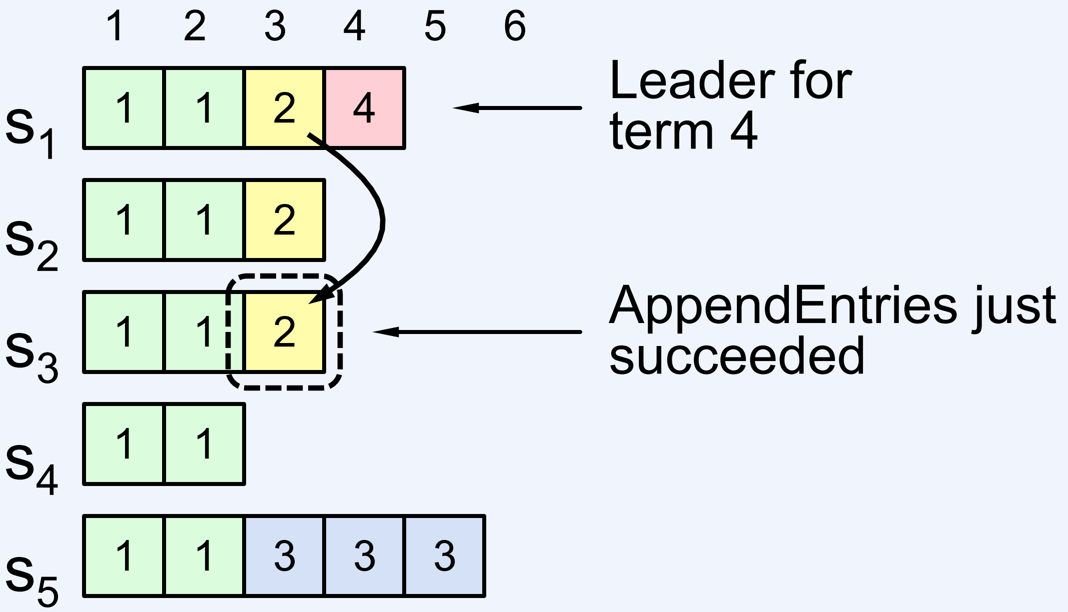
- Case #1/2: Leader decides entry in current term is committed



- Safe: leader for term 3 must contain entry 4

Committing Entry from Earlier Term

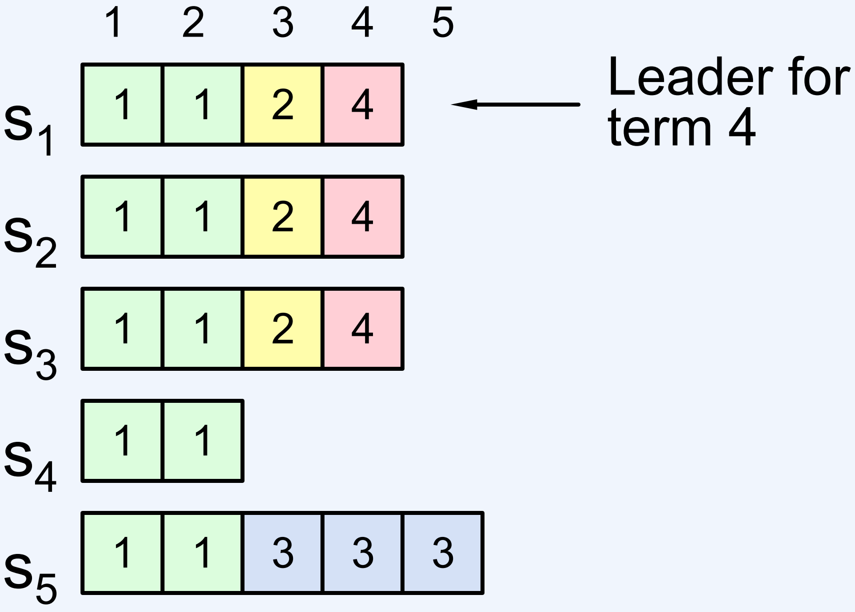
- Case #2/2: Leader is trying to finish committing entry from an earlier term



- 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

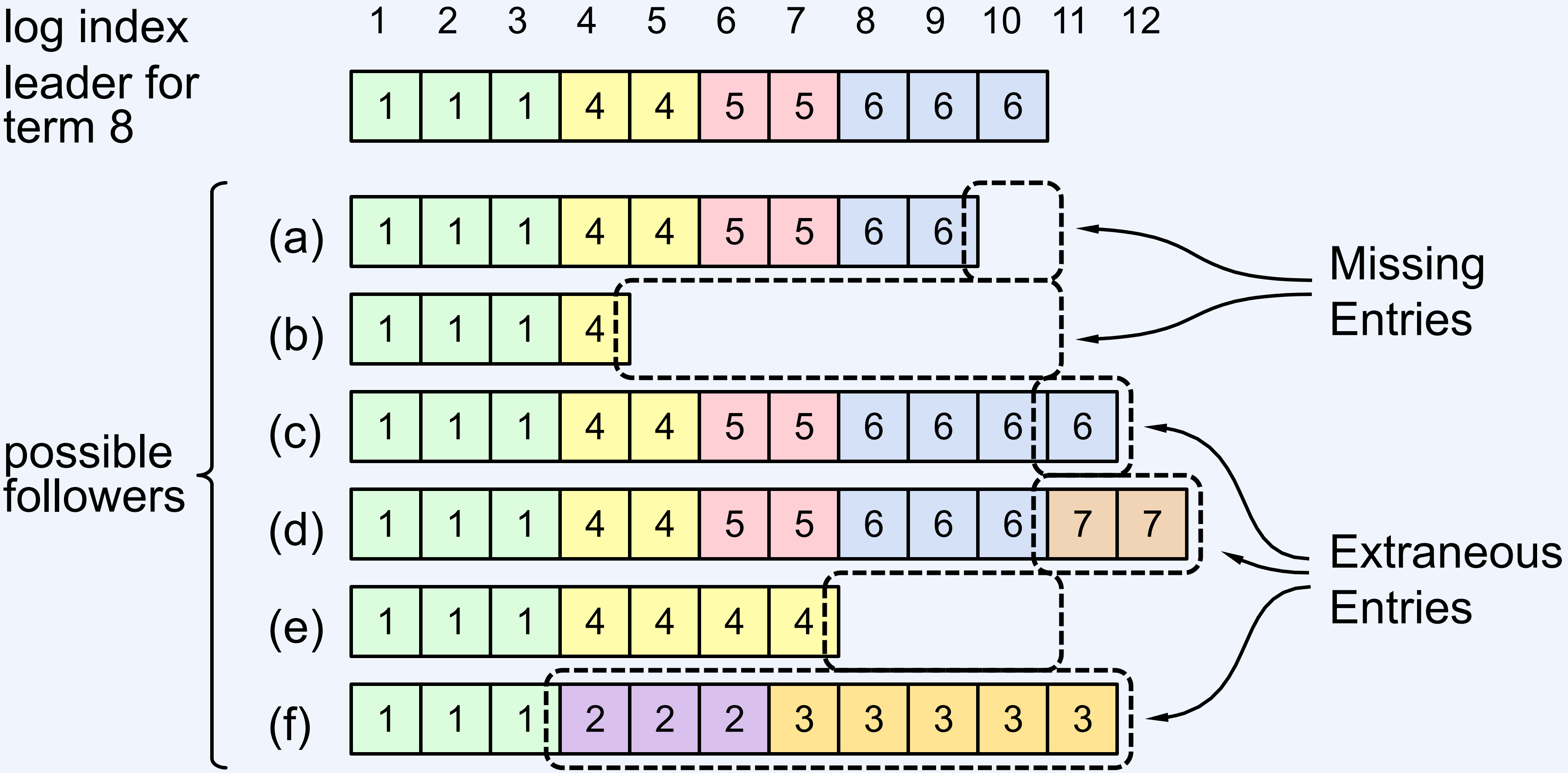
- For a leader to decide an entry is committed:
  - Must be stored on a majority of servers
  - At least one new entry from leader's term must also be stored on majority of servers
- Once entry 4 committed:
  - s_5 cannot be elected leader for term 5
  - Entries 3 and 4 both safe



Combination of election rules and commitment rules makes Raft safe

Log Inconsistencies

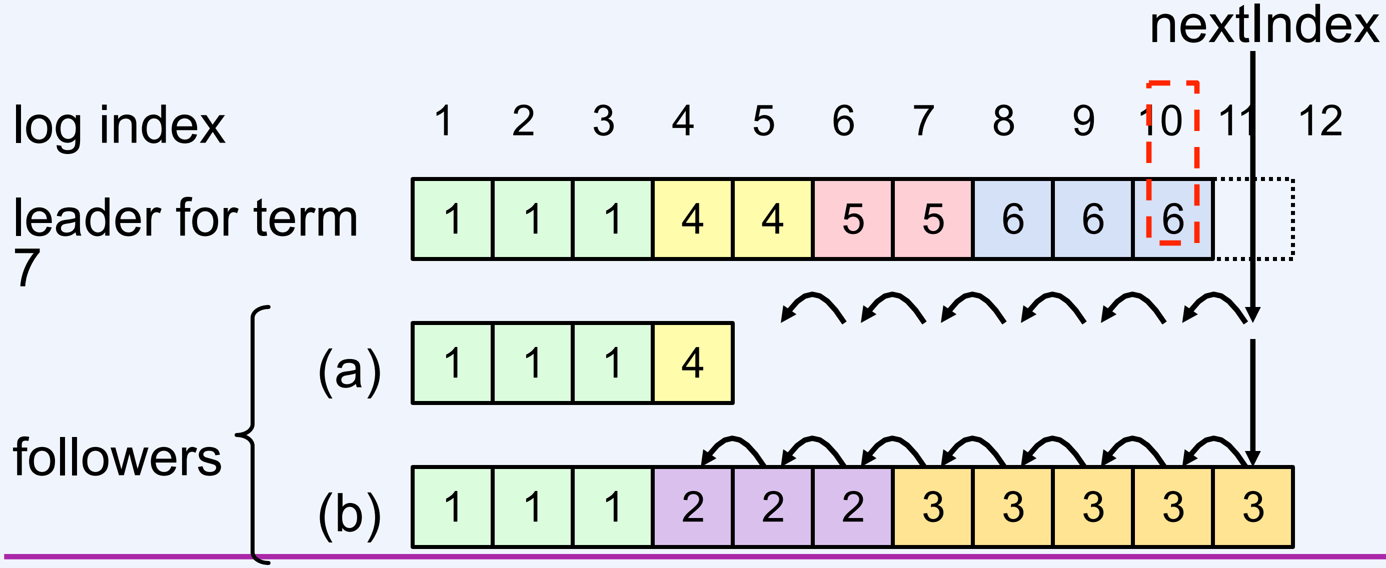
Leader changes can result in log inconsistencies:



log index	1	2	3	4	5	6	7	8	9	10	11	12
leader for term 8	1	1	1	4	4	5	5	6	6	6		
(a)	1	1	1	4	4	5	5	6	6			
(b)	1	1	1	4								
(c)	1	1	1	4	4	5	5	6	6	6	6	
(d)	1	1	1	4	4	5	5	6	6	6	7	7
(e)	1	1	1	4	4	4	4					
(f)	1	1	1	2	2	2	3	3	3	3	3	

Repairing Follower Logs

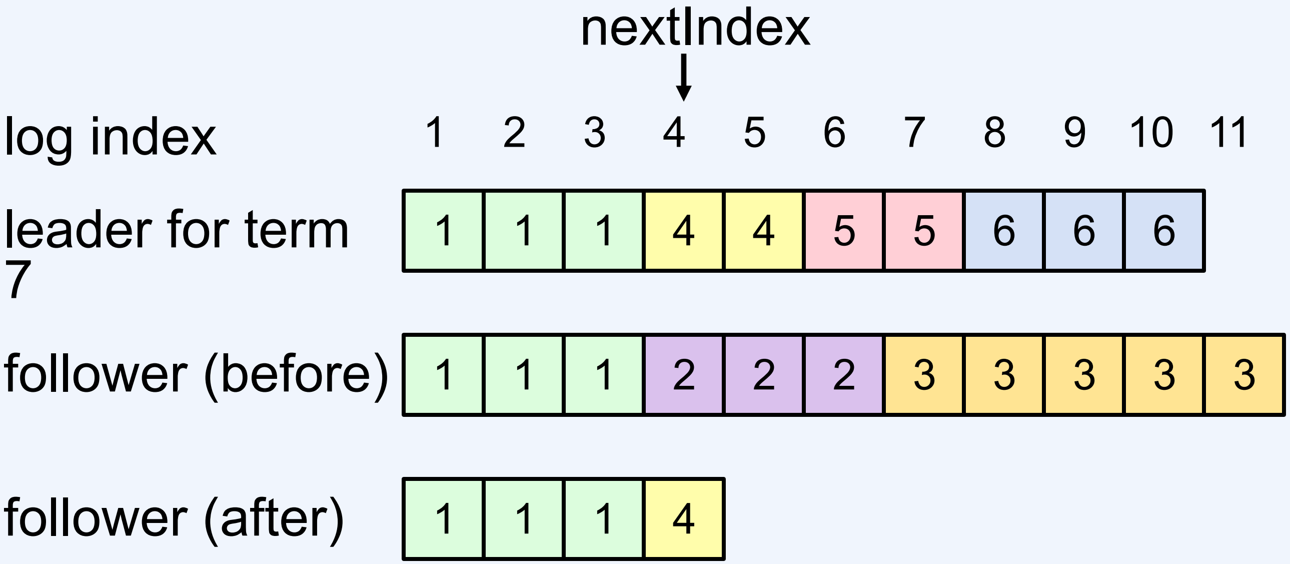
- New leader must make follower logs consistent with its own
  - Delete extraneous entries
  - Fill in missing entries
- Leader keeps nextIndex for each follower:
  - Index of next log entry to send to that follower
  - Initialized to (1 + leader's last index)
- When AppendEntries consistency check fails, decrement nextIndex and try again:



		1	2	3	4	5	6	7	8	9	10	11	12
log index		1	2	3	4	5	6	7	8	9	10	11	12
leader for term 7		1	1	1	4	4	5	5	6	6	6		
followers	(a)	1	1	1	4								
	(b)	1	1	1	2	2	2	3	3	3	3	3	

Repairing Logs, cont'd

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



nextIndex

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							

Neutralizing Old Leaders

- Deposed leader may not be dead:
  - Temporarily disconnected from network
  - Other servers elect a new leader
  - Old leader becomes reconnected, attempts to commit log entries
- Terms used to detect stale leaders (and candidates)
  - Every RPC contains term of sender
  - If sender’s term is older, RPC is rejected, sender reverts to follower and updates its term
  - If receiver’s term is older, it reverts to follower, updates its term, then processes RPC normally
- Election updates terms of majority of servers
  - Deposed server cannot commit new log entries

Client Protocol

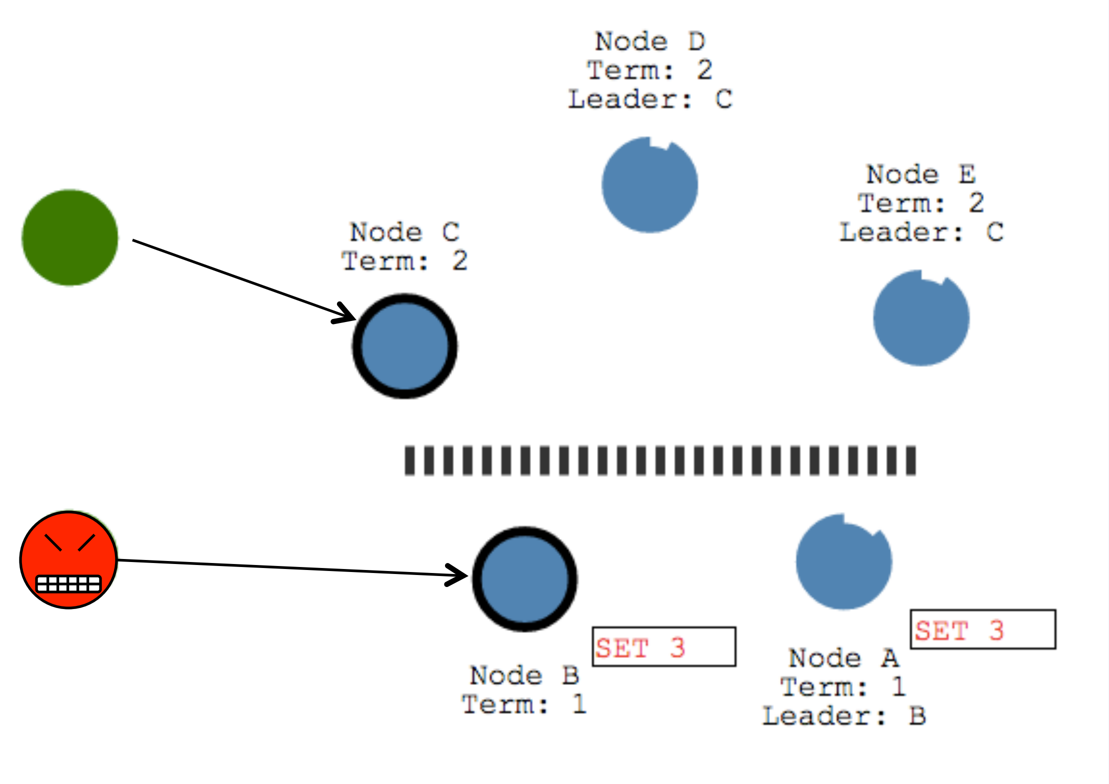
- Send commands to leader
  - If leader unknown, contact any server
  - If contacted server not leader, it will redirect to leader
- Leader does not respond until command has been logged, committed, and executed by leader’s state machine
- If request times out (e.g., leader crash):
  - Client reissues command to some other server
  - Eventually redirected to new leader
  - Retry request with new leader

Client Protocol, cont’d

- What if leader crashes after executing command, but before responding?
  - 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
- Result: exactly-once semantics as long as client doesn’t crash
- Enforces **linearizability** (will see in upcoming lecture)

Partitioned Leader

- A client talking to a partitioned leader could be delayed forever.
  - Solution: leader will step down after a number of rounds of heartbeats with no response from majority



What if clients can crash?

- Servers maintain a session for each client
  - Keep track of latest sequence number processed for client, and response
- Generalizes for multiple outstanding requests
  - Server keeps set of <seq, resp> for client
  - Client includes with request lowest seq with no response
  - Server can discard smaller sequence numbers
- Must expire sessions
  - All replicas must agree on when to do this
  - Raft uses leader timestamp, committed to log

Alive clients with expired sessions

- How to distinguish between client which exited from client which just took too long?
- Require clients to register with the leader when starting a session
  - RegisterClient RPC
  - Leader returns unique ID to the client
  - Client uses this ID in subsequent request
- If server receives request for non-existing session…
  - Return an error. Current implementation crashes the client, forcing restart

Configuration Changes

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

- See the paper for details