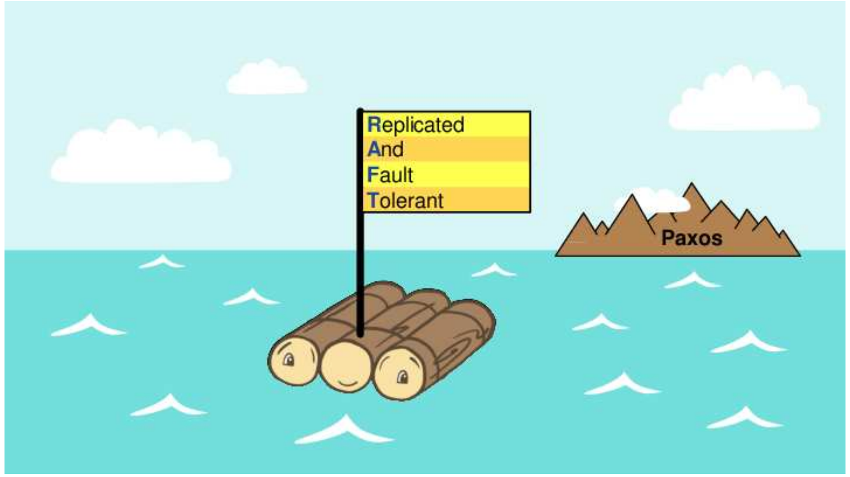
CS4224/CS5424 Lecture 9
Raft Consensus Algorithm



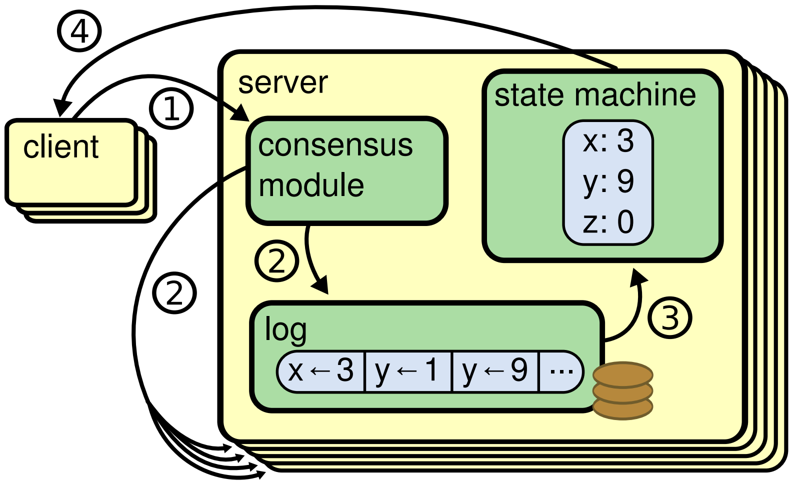
(Ongaro & Ousterhout, 2014)

Consensus Algorithms

- **Consensus Problem**: How to get multiple servers to agree on the same state
- **Consensus Algorithms**:
  - Viewstamped Replication, 1988
  - Paxos, 1990
  - Zab (Zookeeper Atomic Broadcast), 2011
  - Raft, 2014

Replicated State Machines (RSM)

- Consensus algorithm used to implement RSM to provide **fault-tolerant** distributed services
- Service is available as long as a majority of servers are operational and can communicate with each other and clients
- **Consensus algorithm** ensures each RSM receives the same sequence of inputs

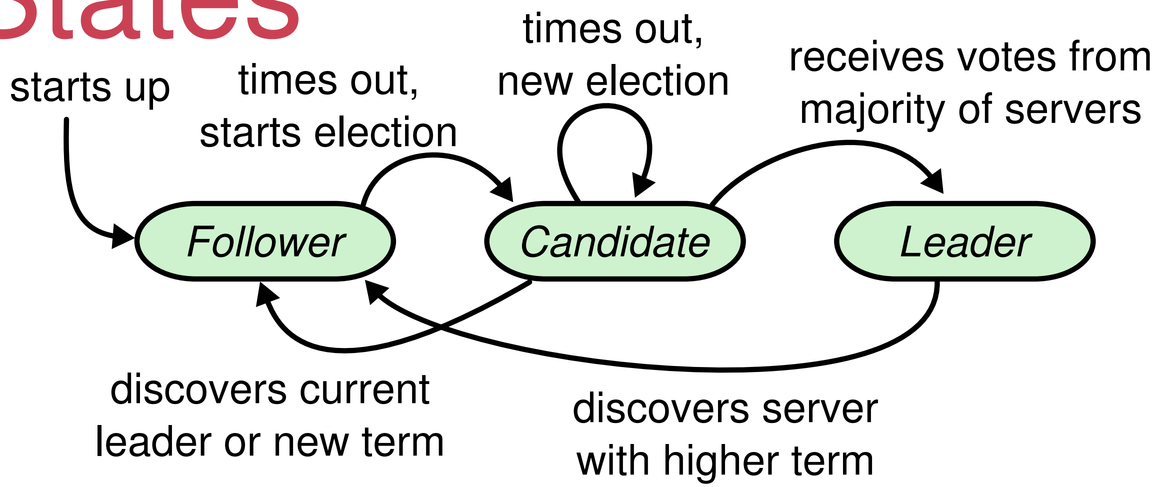


(Ongaro & Ousterhout, 2014)

Raft Concepts

- Server states: follower, candidate, leader
- Term
- Log
  - Log entry: (index, term, command)
  - Log comparison - determine which log is more up-to-date (a.k.a. more complete)
  - Committed log entry
- Remote Procedure Calls (RPCs):
  - RequestVote
  - AppendEntries
- Leader election

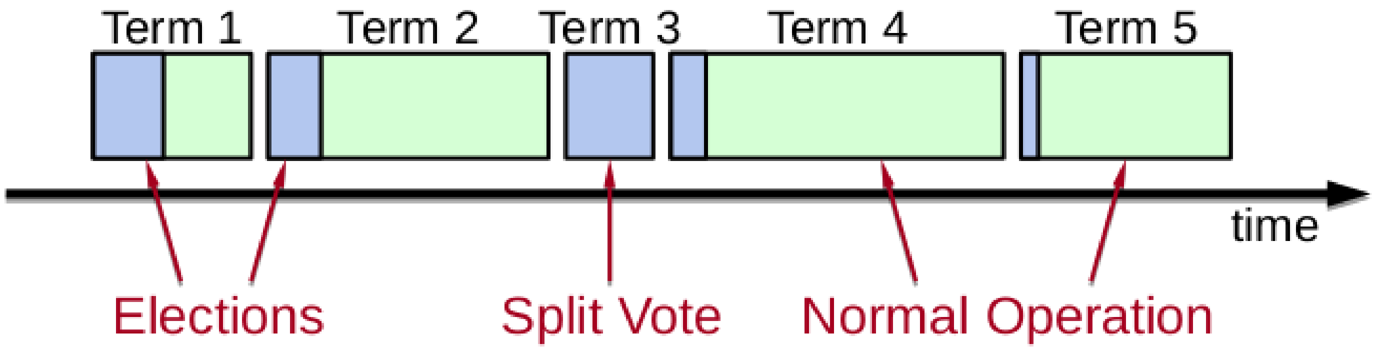
Server States



(Ongaro & Ousterhout, 2014)

- **Follower** - Passive but expects regular heartbeats from leader
- **Candidate** - Issues RequestVote RPCs to get elected as leader
- **Leader** - handles client interactions & issues AppendEntries RPCs
  - Replicate its log
  - Sends heartbeats to maintain leadership

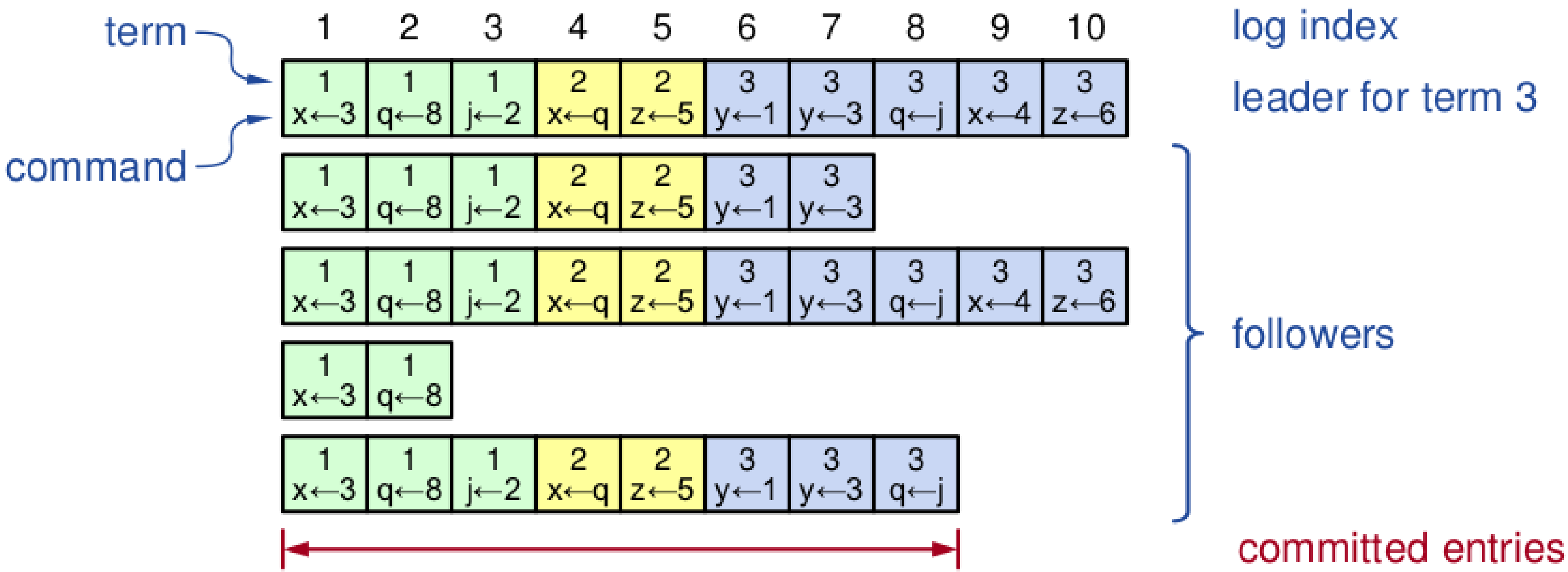
Terms



(Ongaro & Ousterhout, 2014)

- Each term starts with an election
- At most one leader elected in each term
- Each server maintains current term value
- RPCs/replies include sender's current term
- Server updates its current term number if it receives a message with larger term number

Logs



(Ongaro & Ousterhout, 2014)

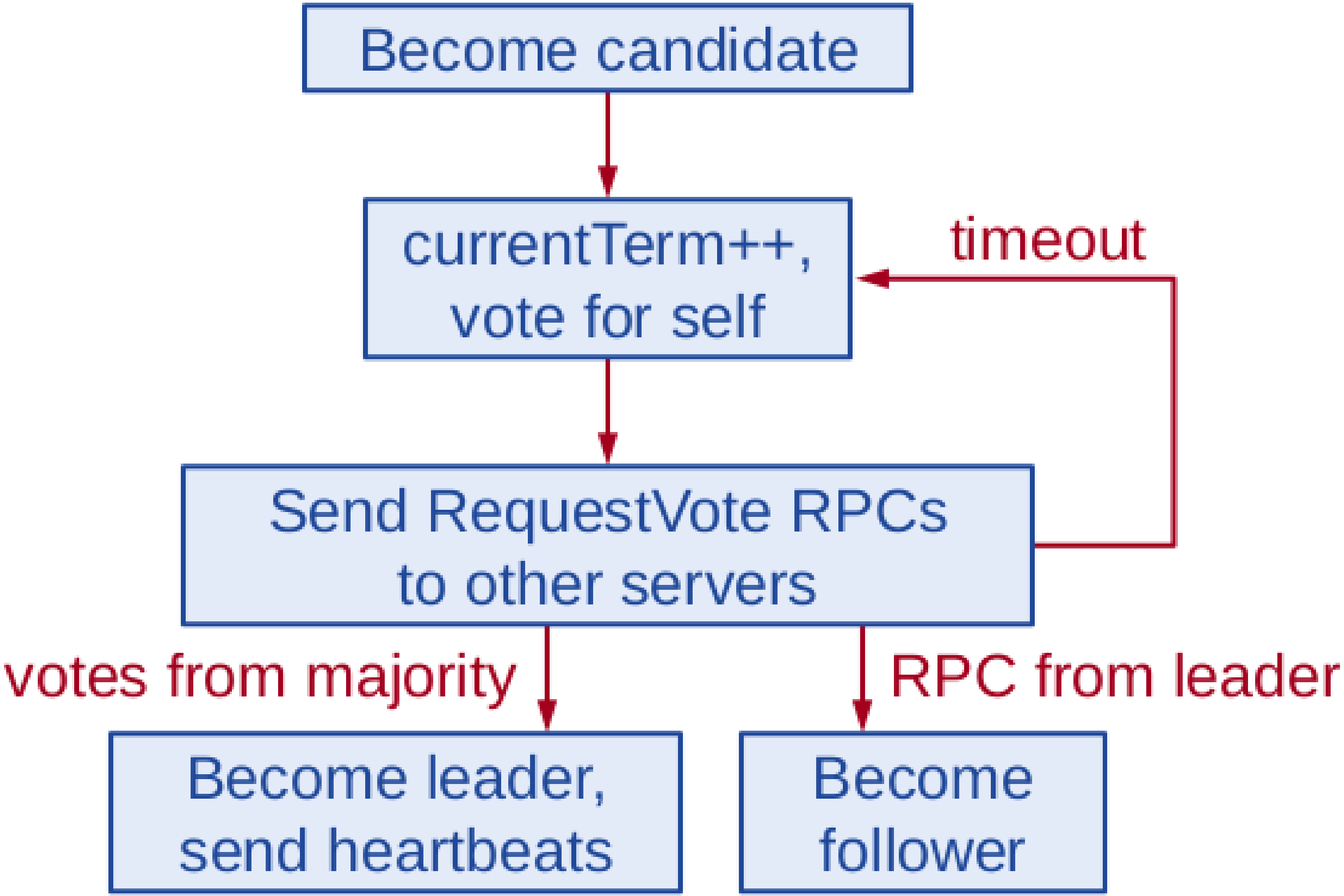
Remote Procedure Calls (RPCs)

- **RequestVote RPC**
  - Send by candidate to request for votes to be elected as leader
- **AppendEntries RPC**
  - Send by leader to replicate its log or as a heartbeat message
- A RPC is resent to server R if leader didn't receive R's response when leader timer timeouts

Timers

- **Election timer** - follower becomes a candidate or candidate restarts a new election if it didn’t receive any RPC
- **Leader timer** - leader resends RPC to follower F if leader didn’t receive F’s response
- **Client timer** - client resends command if it didn’t receive leader’s response to command

Leader Election



(Ongaro & Ousterhout, 2014)

Election Properties

- Election Safety Property: at most one leader can be elected in any term
  - Each server gives out only one vote each term
  - A candidate becomes elected as a leader it if receives a majority of votes
- Election Liveness Property: some leader must eventually be elected
  - Each server chooses a election timeout duration randomly from $[T, 2T]$
  - Works well if $T >> $ broadcast time
    - Broadcast time = average time for server to send RPCs and receive their responses
  - One server usually wins election before other election timers timeout

Persistent State on All Servers

- **currentTerm** - latest term that server has seen
- **votedFor** - candidate that received vote in current term (null if none)
- **log[]** - log entries of the form (index,term,command). First index is 1.

RequestVote RPC

- RequestVote RPC Arguments:
  - **candidateId** = identifier of candidate
  - **term** = candidate's term
  - **lastLogIndex** = index of candidate's last log entry
  - **lastLogTerm** = term of candidate's last log entry
- Response of the form (term, voteGranted)
  - **term** = current term of responding server R
  - **voteGranted** = *true* if R votes for sender; *false*, otherwise

Server R's Response to RequestVote RPC

- **Case 1**: RPC.term $<$ R.currentTerm
  - R replies (R.currentTerm, false)

- **Case 2**: (RPC.term $>$ R.currentTerm) and **p**
  - R.currentTerm = RPC.term & R.votedFor = RPC.candidateId
  - R replies (R.currentTerm, true)

- **Case 3**: (RPC.term = R.currentTerm) and (R.votedFor = null) and **p**
  - R.votedFor = RPC.candidateId
  - R replies (R.currentTerm, true)

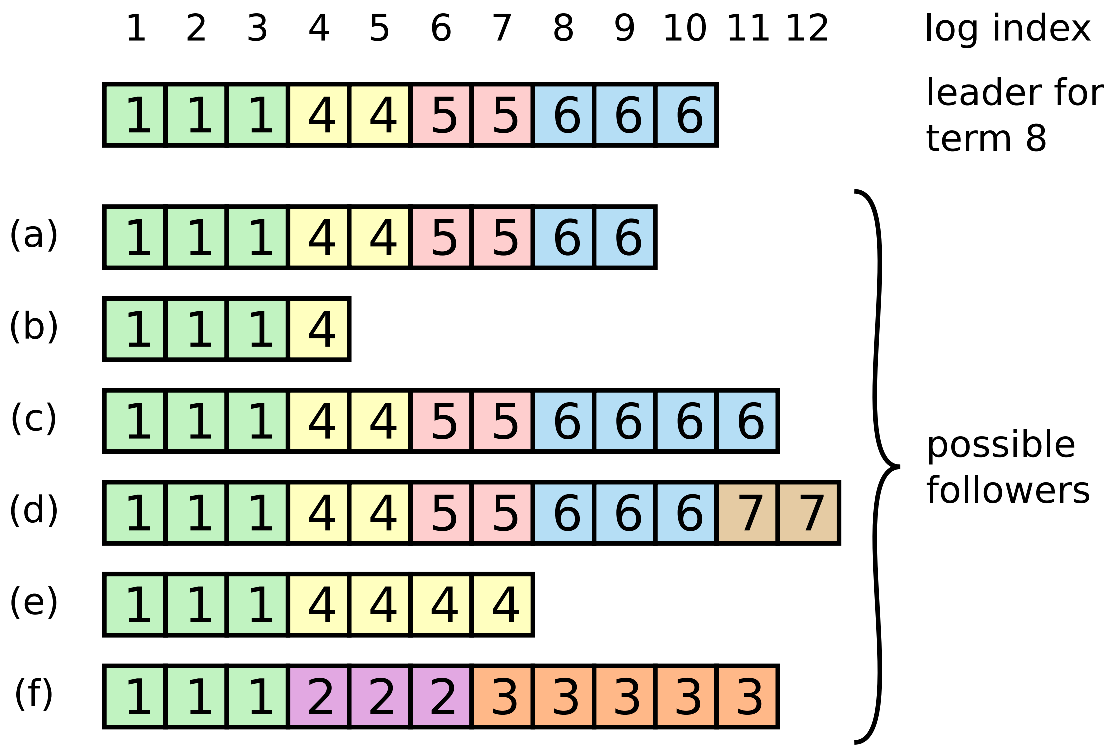
- **Case 4**: (RPC.term = R.currentTerm) and (R.votedFor = RPC.candidateId)
  - R replies (R.currentTerm, true)

- **Case 5**: In all other cases,
  - if RPC.term $>$ R.currentTerm then
    R.currentTerm = RPC.term & R.votedFor = null
  - R replies (R.currentTerm, false)

p: R's log is not more complete than sender's log

Comparing Log's Completeness

- X's log is more complete than Y's log if
  1. either X.lastLogTerm > Y.lastLogTerm
  2. or (X.lastLogTerm = Y.lastLogTerm) and (X.lastLogIndex > Y.lastLogIndex)



(Ongaro & Ousterhout, 2014)

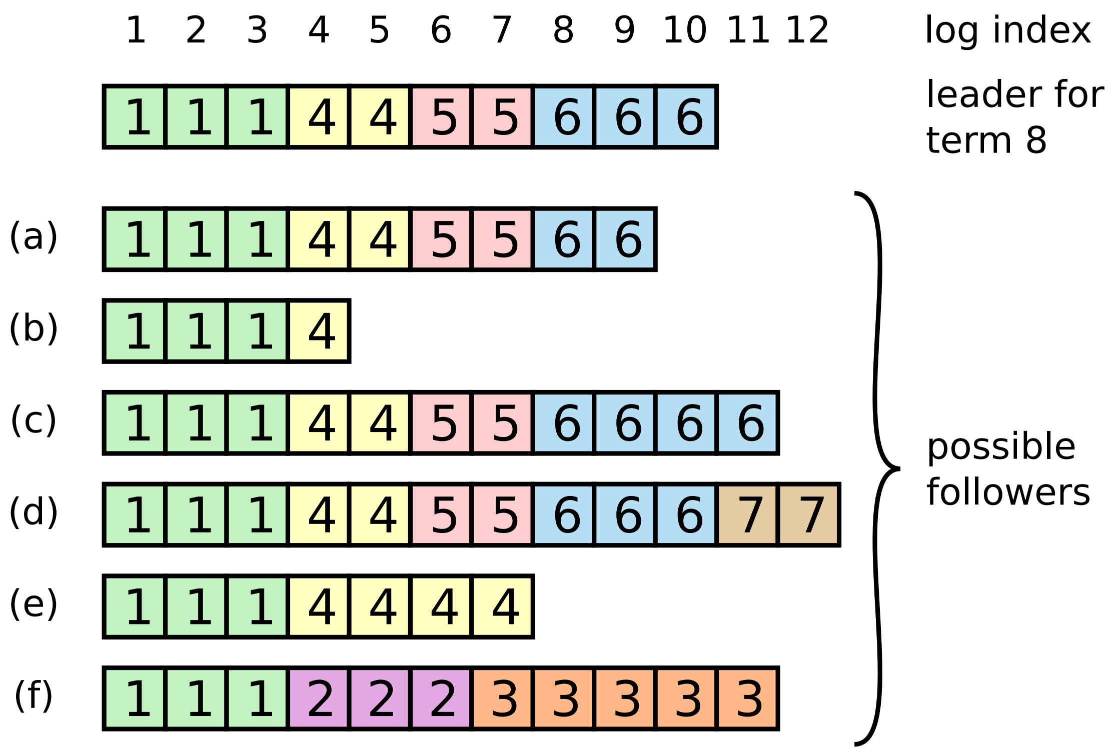
Normal Operations

- Client sends command to leader
- Leader appends command to its log
- Leader sends **AppendEntries RPCs** to all followers
- Once new log entry is committed
  - Leader executes command in its state machine & returns result to client
  - Leader notifies followers of committed entries in subsequent **AppendEntries RPCs**
  - Followers execute committed commands in their state machines
- **Leader Append-Only Property**: a leader never overwrites or deletes entries in its log; it only appends new entries

Log Matching Property

(1) If two entries in different logs have the same index and term, then they store the same command

(2) If two entries in different logs have the same index and term, then the logs are identical in all preceding entries

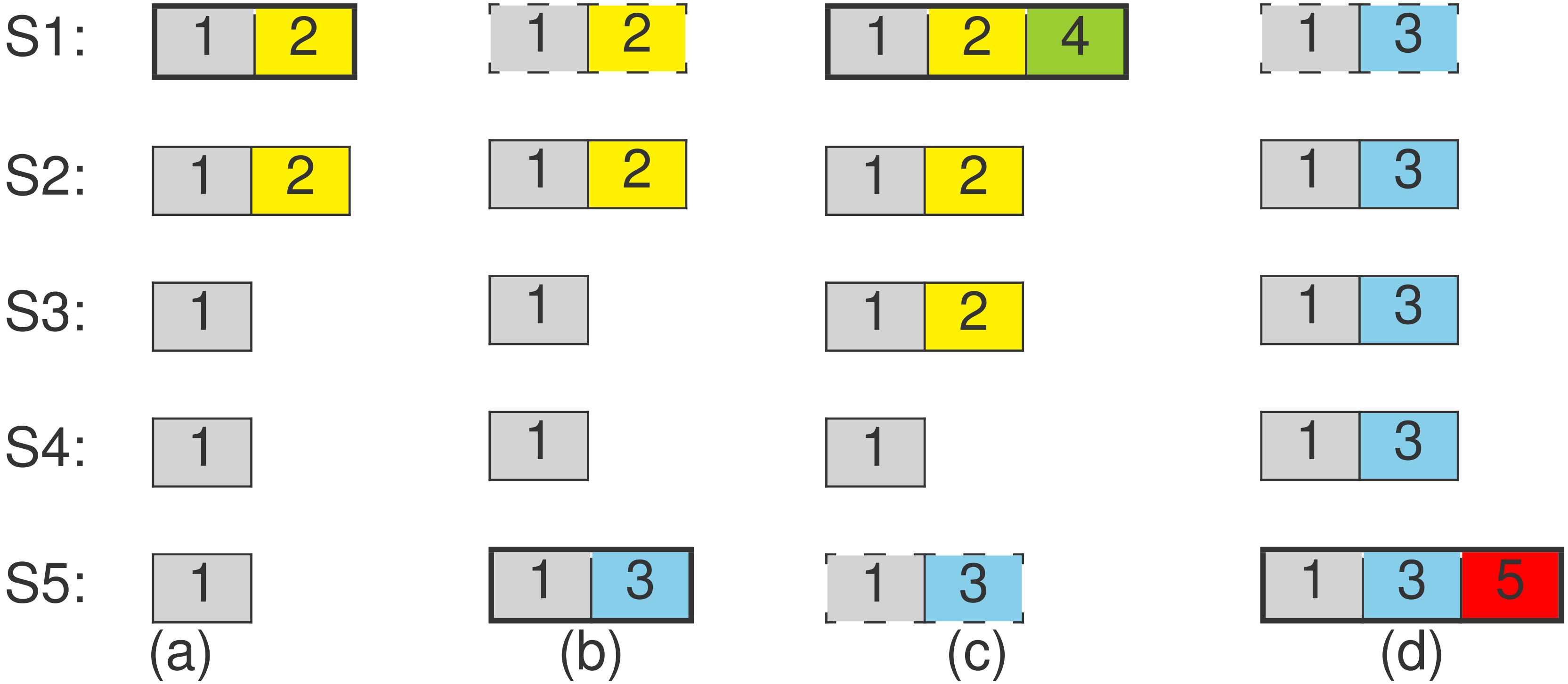


(Ongaro & Ousterhout, 2014)

Committed Log Entries

- A log entry is directly committed once the leader that created the entry has replicated it to a majority of servers
- All log entries preceding a directly committed entry are indirectly committed
- A log entry is committed if the entry is directly or indirectly committed

Committed Log Entries: Example 1



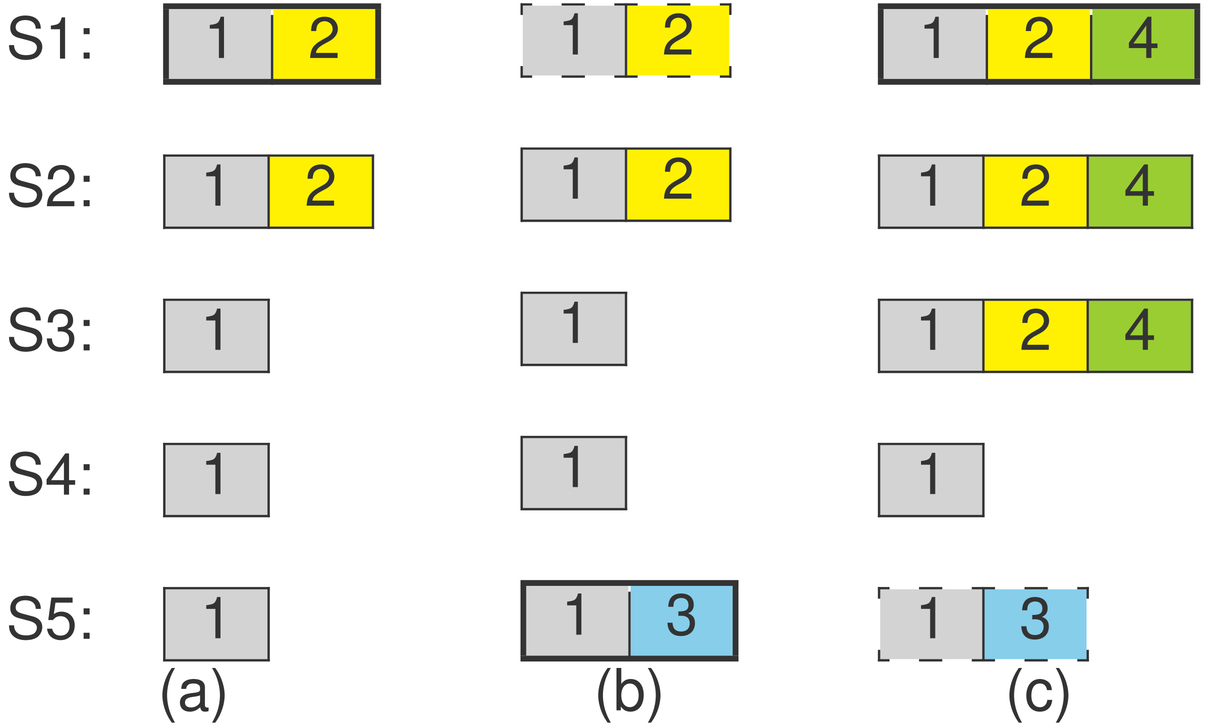
(a) S1 is leader & partially replicates new entry (2,2)

(b) S1 fails, S5 becomes leader & appends new entry (2,3)

(c) S5 fails, S1 becomes leader, appends new entry (3,4) & replicates entry (2,2) to S3

(d) S1 fails, S5 becomes leader, appends new entry (3,5) & replicates entry (2,3)

Committed Log Entries: Example 2



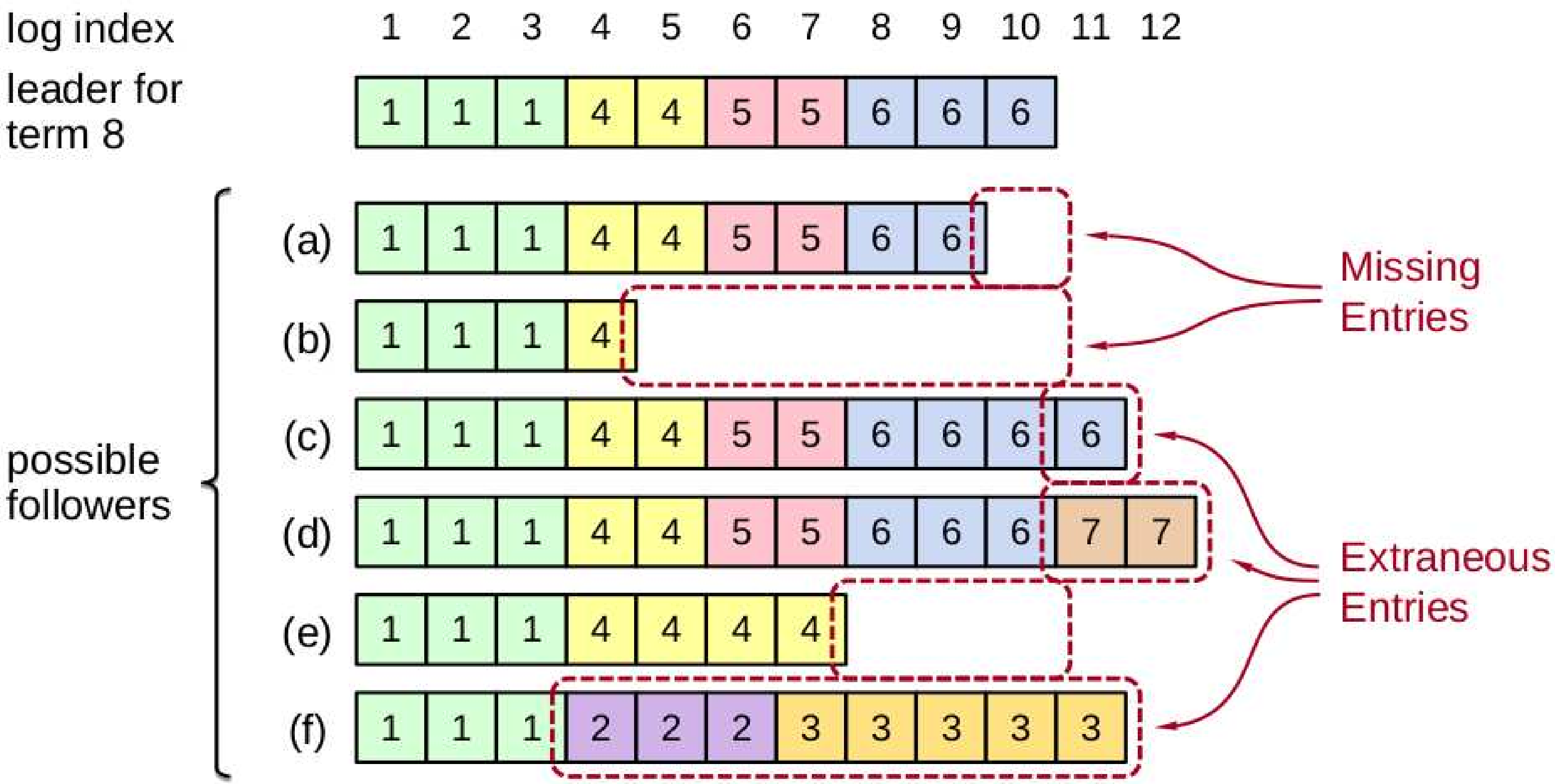
(a) S1 is leader & partially replicates new entry (2,2)

(b) S1 fails, S5 becomes leader & appends new entry (2,3)

(c) S5 fails, S1 becomes leader, appends new entry (3,4) & partially replicates entry (3,4)

Log Inconsistencies

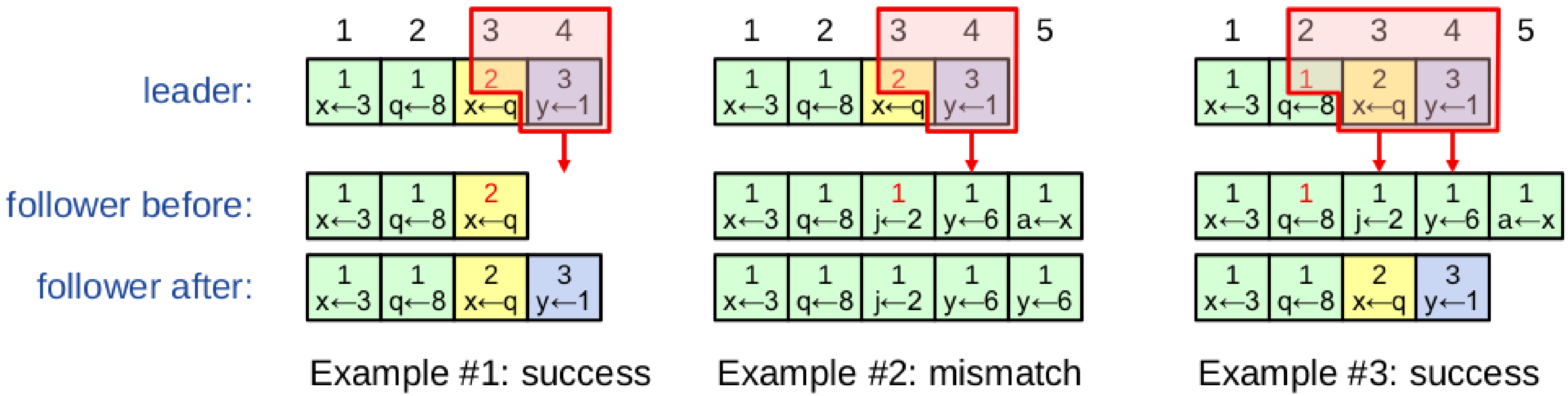
- Server failures can cause log inconsistencies



(Ongaro & Ousterhout, 2014)

AppendEntries Consistency Check

- AppendEntries RPCs include (index,term) of entry preceding new one(s)
- Follower F must contain matching entry; otherwise F rejects AppendEntries RPC request & leader retries with lower log index



(Ongaro & Ousterhout, 2014)

Volatile State (assumes log index starts at 1)

Volatile state on all servers (both values are initialized to 0):

- **commitIndex** - index of highest log entry known to be committed
- **lastApplied** - index of highest log entry applied to state machine

Volatile state on leaders (reinitialized after election):

- **nextIndex[]** - for each server, index of next log entry to send to that server
  - Initialized to index of leader's last log entry + 1
- **matchIndex[]** - for each server, index of highest log entry known to be replicated on server
  - Initialized to 0

AppendEntries RPC

- AppendEntries RPC Arguments:
  - **leaderId** = identifier of leader
  - **leaderTerm** = leader's term
  - **leaderCommit** = leader's commitIndex
  - **prevLogIndex** = index of log entry immediately preceding new log entries
  - **prevLogTerm** = term of *prevLogIndex* log entry
  - **entries[]** = log entries to store
    - entries[] is empty if AppendEntries RPC is used for heartbeat message
- Response of the form (term, success)
  - **term** = current term of responding follower F
  - **success** = *true* if F contains entry matching prevLogIndex & prevLogTerm; *false*, otherwise

Processing AppendEntries RPC (by follower F)

1. If leaderTerm $<$ F.currentTerm then reply (F.currentTerm, false)
2. If leaderTerm $>$ F.currentTerm then **F.currentTerm** = leaderTerm
3. If entries[] is not empty then
   - 3.1 If F's log doesn't contain (prevLogIndex, prevLogTerm), then reply (F.currentTerm, false)
   - 3.2 If an entry e in **F's log** conflicts with a new entry (i.e., same index but different term), then delete e & all entries that follow e
   - 3.3 Append any new entries not already in **F's log**
4. If leaderCommit $>$ F.commitIndex then set **F.commitIndex** = min (leaderCommit, index of last entry in F's log)
5. Reply (F.currentTerm, true)

Leader Completeness Property

- **Leader Completeness Property**: 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
- Leader election ensures that leader's log is at least as complete as a majority of servers' logs
  - This guarantees Leader Completeness Property

Rules for all Servers

- If commitIndex $>$ lastApplied, then
  - Increment **lastApplied** by one
  - Apply log[lastApplied] to state machine
- If RPC request or response contains term $T >$ currentTerm, then
  - Set **currentTerm** = T
  - Convert server to follower

Rules for Leaders

- Upon election:
  - Send initial empty AppendEntries RPC to each server
  - Repeat during idle period to prevent timeouts of election timer
- If received command from client, then
  - Append entry to **local log**
  - Respond to client after entry has been applied to state machine
- If index of last log entry $\geq$ nextIndex for a follower F, then
  - Send AppendEntries RPC to F with log entries starting at nextIndex
  - If successful, then update **nextIndex** & **matchIndex** for F
  - Otherwise, if AppendEntries RPC fails because of log inconsistency, then decrement nextIndex & retry
- If there exists an N such that $N >$ commitIndex, a majority of matchIndex[i] $\geq N$, and log[N].term = currentTerm, then
  - Set **commitIndex** = N

State Machine Safety Property

- **State Machine Safety Property**: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index

Raft topics that are not covered

- Managing cluster membership
- Log compaction
- Client interaction

Summary of Raft Properties

- **Election Safety**: at most one leader can be elected at a given term
- **Election Liveness**: some leader must eventually be elected
- **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, no other server will ever apply a different log entry for the same index

References

- D. Ongaro, J. Ousterhout, In Search of an Understandable Consensus Algorithm, USENIX Annual Technical Conference 2014
- https://raft.github.io/