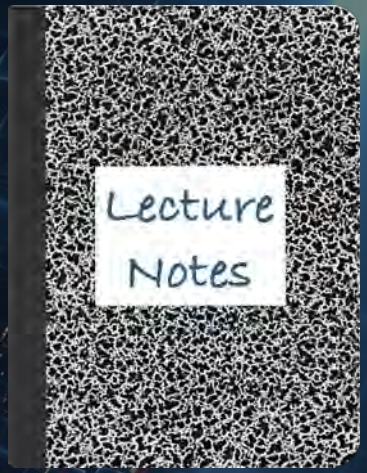
CS 417 – DISTRIBUTED SYSTEMS

Week 5: Consensus
Part 2: Paxos & Raft



Paul Krzyzanowski

From theory to working protocols

Consensus is the tool that gives one authoritative decision history

We will look at two families of solutions

- **Paxos**: the foundation (high-level)
- **Raft**: the same goal, designed to be easier to understand & implement

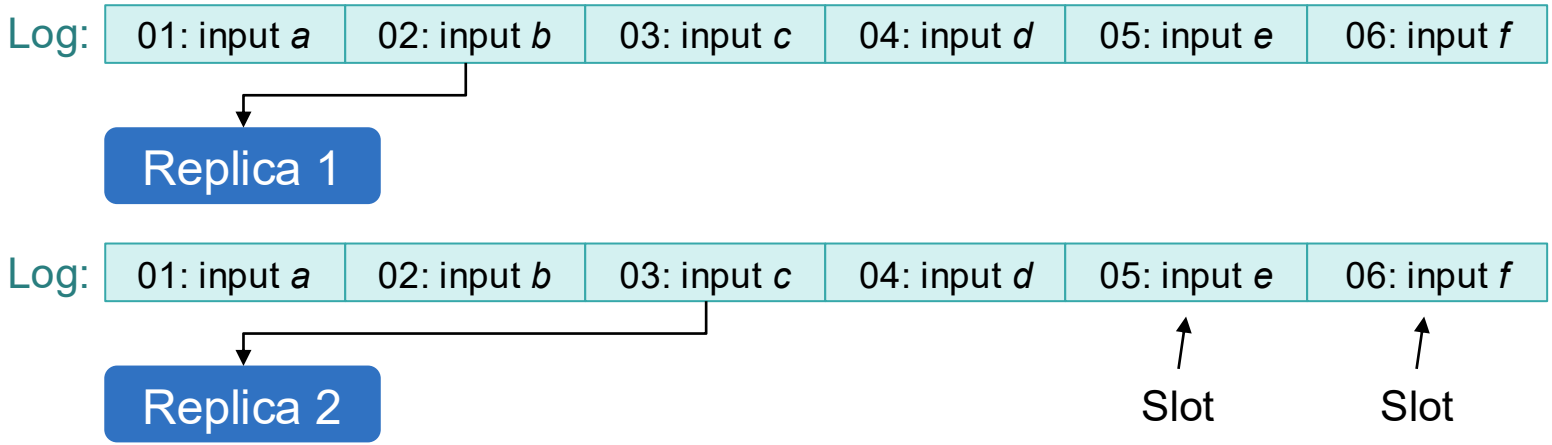
Replicated state machine via a replicated log

Replicated log = list = sequence of slots

Each slot holds an input to the process (state machine)

Consensus: Each process agrees on the contents of each slot

Majority quorum prevents split-brain



Paxos: The Goal

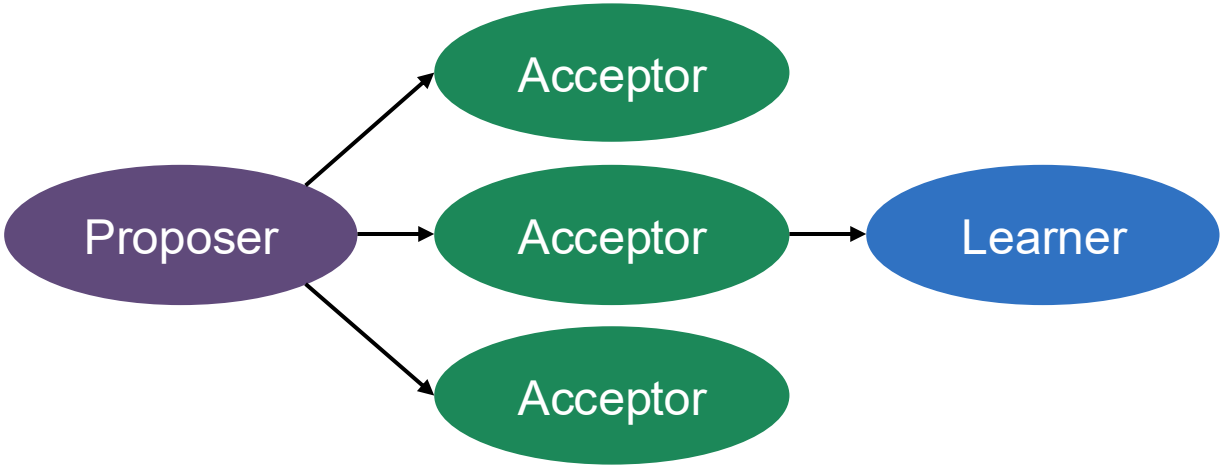
Pick one value

- Even if nodes fail and messages are delayed
- Do it **safely**: never pick two different values
- Do it with a **majority** so only one side can make decisions during disruption

Paxos roles

- **Proposer**: suggests a value and tries to get it chosen
- **Acceptor**: votes on proposals and remembers its vote
- **Learner**: finds out which value was chosen

In implementations, each server usually plays all roles



Paxos in one sentence

A value is **chosen** when a **majority** of acceptors accept the same proposal

- **Proposal numbers** tell *later attempts* to respect earlier accepted values
- This prevents conflicting choices for the same decision

Reminder: a *value* is arbitrary and application-specific:

Examples of "values" nodes may need to agree on

- The next entry in the replicated log is "SET account_balance = 500"
- "This transaction is committed"
- "Server 3 is the new leader for this epoch"
- "The new cluster configuration includes servers 1, 2, and 4"

Paxos Phase 1 (idea): claim a turn to try to decide

Goal: get a majority to recognize your attempt as the newest one

Use an attempt number (**proposal number**) ***n*** that is unique and increasing

1. Ask acceptors to promise: *"I will ignore older attempts"*
2. Acceptors will report any value they have already accepted

Paxos Phase 1 (mechanics): *Prepare and Promise*

- ***Prepare(n)*** sent to all available acceptors
- ***Promise*** reply from each acceptor means:
  - "I will not accept proposals numbered < *n*"
  - "The highest proposal I accepted is (n_a, v_a)" (if any)

Proposer proceeds if it receives promises from a majority

Paxos Phase 2 (idea): propose a value that is safe

Goal: ask acceptors to accept a specific value for attempt *n*

- If any acceptor reported an accepted value,
  reuse the value with highest number
- Otherwise, propose your own value

This rule preserves any value that might already be chosen

Paxos Phase 2 (mechanics): *Accept and Accepted*

Accept(n, v) sent to acceptors

- Acceptor accepts if it has not promised a higher number
- Acceptor records acceptance and replies ***Accepted***

The value *v* is chosen when a majority accept the same *(n, v)*

Paxos Safety

Majorities overlap is the backbone of Paxos

- **Phase 1:**
  Makes the proposer ask a majority what they have already accepted
- **Phase 2:**
  Makes the proposer continue the most recent accepted value it hears about

Result:
Once a value could be chosen, later successful attempts cannot switch to a different value

Multi-Paxos: making Paxos practical for logs

Single-decree Paxos decides one value.

A replicated log needs a sequence

Multi-Paxos:
Use a leader to avoid running Phase 1 each time

- Overhead drops to one round trip per log entry
- But need to handle election of new leader and recovery of partially filled slots

Paxos in production: why it got a reputation

- Correctness depends on careful persistent state
- Leader changes and partial progress create many corner cases
- Membership changes are not built into basic Paxos

Result: easy to misunderstand, easy to implement incorrectly

> "The dirty little secret of the NSDI community is that at most five people really, truly understand every part of Paxos"
>
> – Diego Ongaro (author of Raft)

Raft

Goal: same safety and performance as Multi-Paxos

Design focus: understandability and a focus on replicated logs

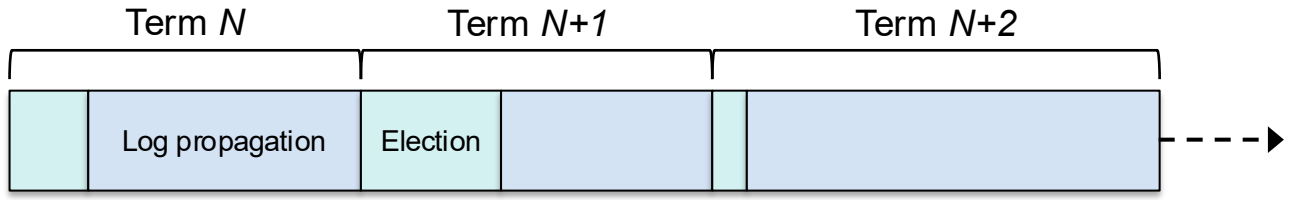
Decomposes the problem:

1. Leader election
2. Log replication
3. Safety rules

Strong leader model: *all writes go through the leader*

Raft terminology: terms and server roles

- Time is divided into **terms**: 1, 2, 3, ...
- Each term has at most one leader
- Server roles: ***follower***, ***candidate***, ***leader***
- All messages include the sender's term



Raft leader election

Leaders periodically send *AppendEntries* RPCs

- A *leader* must send this to all followers at least every ***heartbeat*** interval
- These might contain no entries but act as a heartbeat

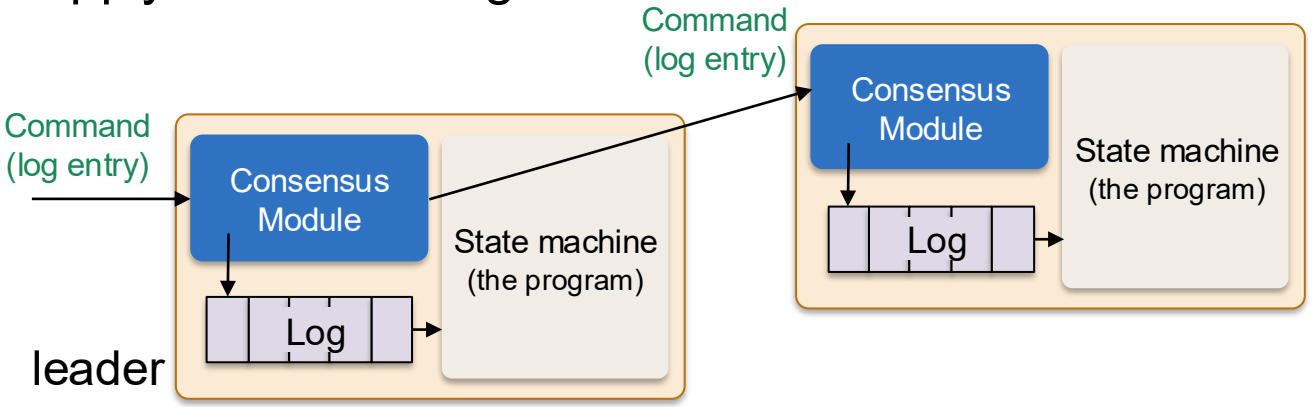
If a follower times out waiting for a heartbeat, it starts an election

- Changes its state to ***candidate***; increments its term #
- Sets a random election timeout
- Votes for itself
- Sends ***RequestVote*** RPC messages to all other members

- Majority votes → leader; then leader sends heartbeats

Raft log replication

- Leader appends client command to its log
  - Sends ***AppendEntries*** to followers
  - Entry is committed once stored on a majority of followers
- Leader tells followers the commit index
- Followers apply committed log entries in order



Raft's key safety mechanism: log matching

AppendEntries RPC includes

- `prevLogIndex` : log index # of its previous log entry
- and `prevLogTerm`: term # of the previous log entry

Follower accepts the entry *only* if its log matches at that point

- The entry it receives is what it expects to be the next entry

- If mismatch: the follower rejects, leader backs up and retries
  - Example: it receives index 5 but does not have index 4
  - When the follower rejects, the leader will send the earlier entry
  - The follower keeps rejecting until the log entries match

Effect: follower logs converge to the leader's log

Committing and “leader completeness”

Commit rule: leader commits current-term entries by majority replication

- Committed entries are never overwritten
- Election restriction: voters prefer candidates with up-to-date logs
- Result: future leaders contain all committed entries

Raft in practice

- **Membership changes**: joint consensus (old + new quorums overlap)
- **Log compaction**: snapshots to bound log growth
- **Common deployment**: 3 or 5 nodes for majority quorums

Paxos vs Raft: what to remember

- **Both**:
  - Majority quorums
  - Replicated logs (multi-Paxos)
  - Safety unconditional (crashes ok)
  - Liveness conditional (progress only with majority)
- **Paxos**: proposal numbers and phases, flexible but subtle
- **Raft**: explicit leader, explicit log rules, easier to explain and implement

Paxos is the theory baseline; Raft is the common modern design

Raft deeper mechanics

Goal:

- Preserve safety under crashes and partitions
- Make progress when a majority can communicate

- Focus on:
  - Elections
  - Log repair
  - Commit rules
  - Membership change

Raft Elections: randomized timeouts

- Split votes are likely when two candidates start together
  - Each candidate collects a subset of votes – no majority
- Minimize the chance of this with **randomized timeouts**
  - Election timeout is randomized per server
  - Leaders send frequent heartbeats to reset timeouts

Elections: *up-to-date log* voting

Vote only for candidates with up-to-date logs

Up-to-date defined by:

1. `lastLogTerm`
2. `lastLogIndex`

Prevents electing a leader that is missing committed entries

Log Repair

Each *AppendEntries* RPC carries information about the previous log entry

- `prevLogIndex` and `prevLogTerm`

- A follower accepts the entry only if its log matches at that point
- If there's a mismatch → reject
  - Leader backs up and retries
  - Follower keeps rejecting until the leader backs up to the point where logs are synchronized
  - Then the leader sends updates

Log repair: why overwriting is safe

- **If mismatch**: follower deletes conflicting suffix (last entry in the log)
- Leader sends the correct suffix for that follower

Committed entries are never overwritten

Commit rule: why leaders commit carefully

Leader marks an entry committed when it is on a majority

- But leaders only ***directly commit*** entries from their *current term*
- All earlier-term entries become committed when a later current-term entry commits

Failure case: old leader accepts a write

Old leader may be isolated but still thinks it is leader

- It can append entries locally
- It cannot commit without a majority
- Client must treat uncommitted responses as not durable

Failure case: leader crashes mid-replication

Some followers have the new entry, others do not

- New leader elected based on the up-to-date rule
- New leader repairs follower logs via *AppendEntries*

Membership change: why joint consensus exists

Changing membership changes what "majority" means

- Need overlap between old and new quorums
- **Joint consensus phase**: require majorities of both configs

Prevents two disjoint groups from committing during transition

Raft: Key Points

Elections: randomized timeouts + up-to-date voting

Replication: prevLogIndex/prevLogTerm
+ overwrite conflicting suffixes

Commit: majority, with the current-term rule

Membership: joint consensus to preserve quorum overlap

The End