Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Logistics

- HW2 is due today.

- HW3 has been released.
  - you should be able to solve all questions.

- Midterm 1 scores have been released (where you lost points yet to be disclosed).

Agenda for today

- Consensus
  - Consensus in synchronous systems
    - *Chapter 15.4*
  - Impossibility of consensus in asynchronous systems
    - *We will not cover the proof in details*
  - Good enough consensus algorithm for asynchronous systems:
    - *Paxos made simple, Leslie Lamport, 2001*
  - Other forms of consensus algorithm
    - Raft (log-based consensus)
    - Block-chains (distributed consensus)

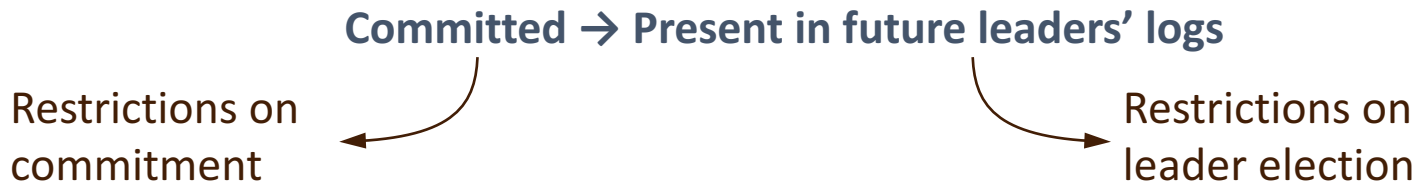
Raft Overview

1. Leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. Neutralizing old leaders
3. Normal operation (basic log replication)
4. Safety and consistency after leader changes

Safety Requirement for log consensus

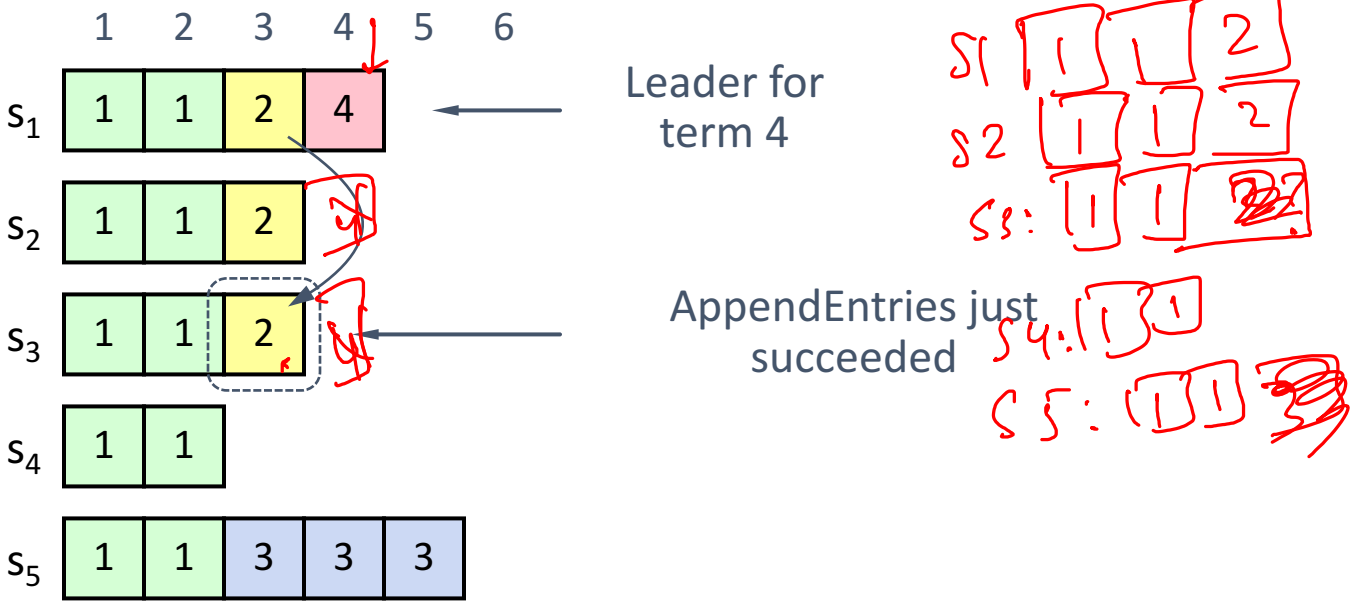
Once a log entry has been applied to a state machine, no other state machine must apply a different value for that log entry.

- 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



Committing Entry from Earlier Term

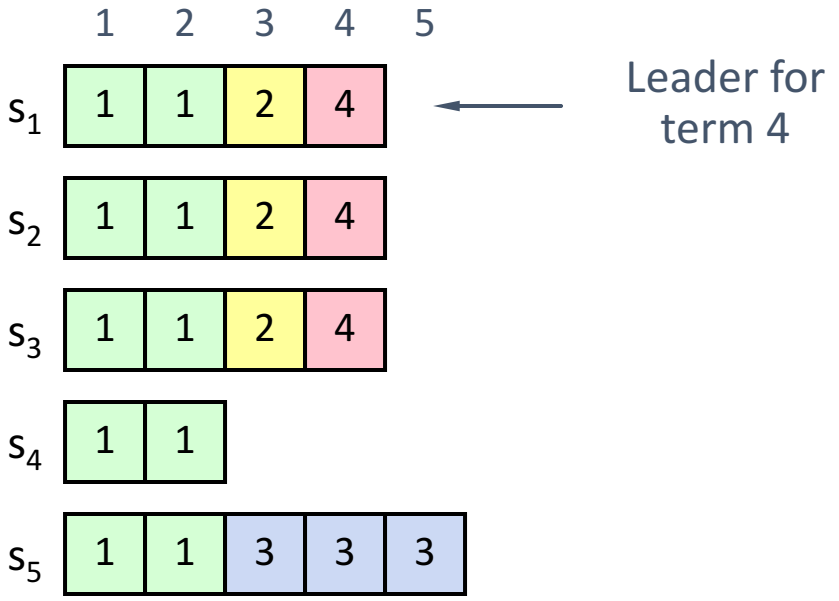
- Leader is trying to finish committing entry from an earlier term



- 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

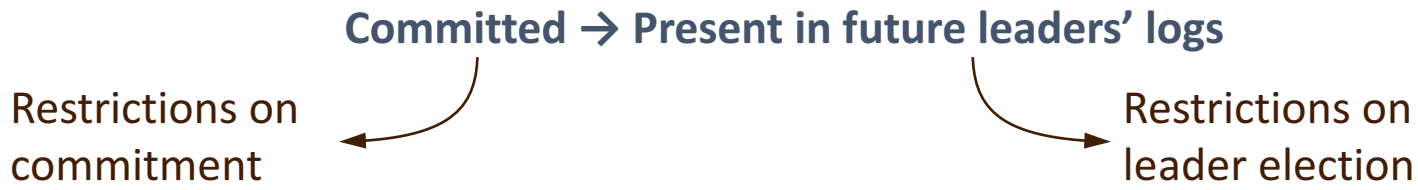
- For a leader to decide an entry is committed:
  - Must be stored on a majority of servers
  - *At least one new entry from leader's current term must also be stored on majority of servers*
- Once entry 4 committed:
  - s_5 cannot be elected leader for term 5
  - Entries 3 and 4 both safe



Safety Requirement for log consensus

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



Raft Protocol Summary

Followers

- Respond to RPCs from candidates and leaders.
- Convert to candidate if election timeout elapses without either:
  - Receiving valid AppendEntries RPC, or
  - Granting vote to candidate

Candidates

- Increment currentTerm, vote for self
- Reset election timeout
- Send RequestVote RPCs to all other servers, wait for either:
  - Votes received from majority of servers: become leader
  - AppendEntries RPC received from new leader: step down
  - Election timeout elapses without election resolution: increment term, start new election
  - Discover higher term: step down

Leaders

- Initialize nextIndex for each to last log index + 1
- Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeouts
- Accept commands from clients, append new entries to local log
- Whenever last log index ≥ nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successful
- If AppendEntries fails because of log inconsistency, decrement nextIndex and retry
- Mark log entries committed if stored on a majority of servers and at least one entry from current term is stored on a majority of servers
- Step down if currentTerm changes

Persistent State

Each server persists the following to stable storage synchronously before responding to RPCs:

currentTerm	latest term server has seen (initialized to 0 on first boot)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries

Log Entry

term	term when entry was received by leader
index	position of entry in the log
command	command for state machine

RequestVote RPC

Invoked by candidates to gather votes.

Arguments:

candidateId	candidate requesting vote
term	candidate's term
lastLogIndex	index of candidate's last log entry
lastLogTerm	term of candidate's last log entry

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Implementation:

1. If term > currentTerm, currentTerm ← term (step down if leader or candidate)
2. If term == currentTerm, votedFor is null or candidateId, and candidate's log is at least as complete as local log, grant vote and reset election timeout

AppendEntries RPC

Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat .

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat)
commitIndex	last entry known to be committed

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Implementation:

1. Return if term < currentTerm
2. If term > currentTerm, currentTerm ← term
3. If candidate or leader, step down
4. Reset election timeout
5. Return failure if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
6. If existing entries conflict with new entries, delete all existing entries starting with first conflicting entry
7. Append any new entries not already in the log
8. Advance state machine with newly committed entries

More details in Raft paper

- Link on the course website.
- Play with the visualization at raft.github.io
- The concepts covered Section 6 and beyond are not in your syllabus.

Agenda for today

- Consensus
  - Consensus in synchronous systems
    - *Chapter 15.4*
  - Impossibility of consensus in asynchronous systems
    - *We will not cover the proof in details*
  - Good enough consensus algorithm for asynchronous systems:
    - *Paxos made simple, Leslie Lamport, 2001*
  - Other forms of consensus algorithm
    - Raft (log-based consensus)
    - Block-chains (distributed consensus)

Bitcoins

- Implement a *distributed* replicated state machine that maintains an *account ledger (= bank).*
  - No user should be able to "double-spend".
  - Need to know of all transactions to validate this.
  - Who does this validation? Cannot trust a single central authority.
    - Any participant (replica) should be able to validate.
  - All replicas must agree on the single history on transaction ordering.
- Scale to thousands of replicas distributed across the world.
- Allow old replicas to fail, new replicas to join seamlessly.
- Withstand various types of attacks.

Uses Blockchains for Consensus

- Why not use Paxos / Raft?
  - Need to scale to thousands of replicas across the world.
  - May not even know of all replicas a priori.
  - Participants may leave / join dynamically.
  - Paxos/Raft are ill-suited for such a setup.
    - Leader election in Raft or proposals in Paxos require communication with at least a majority of servers.
    - Require knowing the number of replicas.
    - ….

- *So how does blockchain work?*
  - Focus of today's class. Only a high-level discussion.

Basic Idea

Transactions grouped into a *block* that gets added to the *chain* (history of transactions) by the "leader of that block".

Lottery Leader Election

- Every node chooses a random number

- Leader = "closest to 0"

Lottery Leader Election

- Every node chooses a random number
  - The method for choosing the number in blockchains enables log consensus (with a high probability).
  - Requires the leader to expend CPU (as *proof-of-work*).

- Leader = "closest to 0"
  - Defined such that a replica can determine this independently without coordination

Choosing the random number

- Cryptographic hash function:
  - H(x) -> { 0, 1, ..., 2^{256}-1}
- Hard to *invert:*
  - Given y, find x such that H(x) = y
  - E.g., SHA256, SHA3, ...
- Every node picks random number x and computes H(x)
- Node with H(x) "closest to 0" wins
  - Finding such an x requires expending CPU (*proof-of-work*).

- *But once we have found an 'x', we can always be the leader for all blocks, or even share it with colluding parties. How to prevent that?*

Using a seed

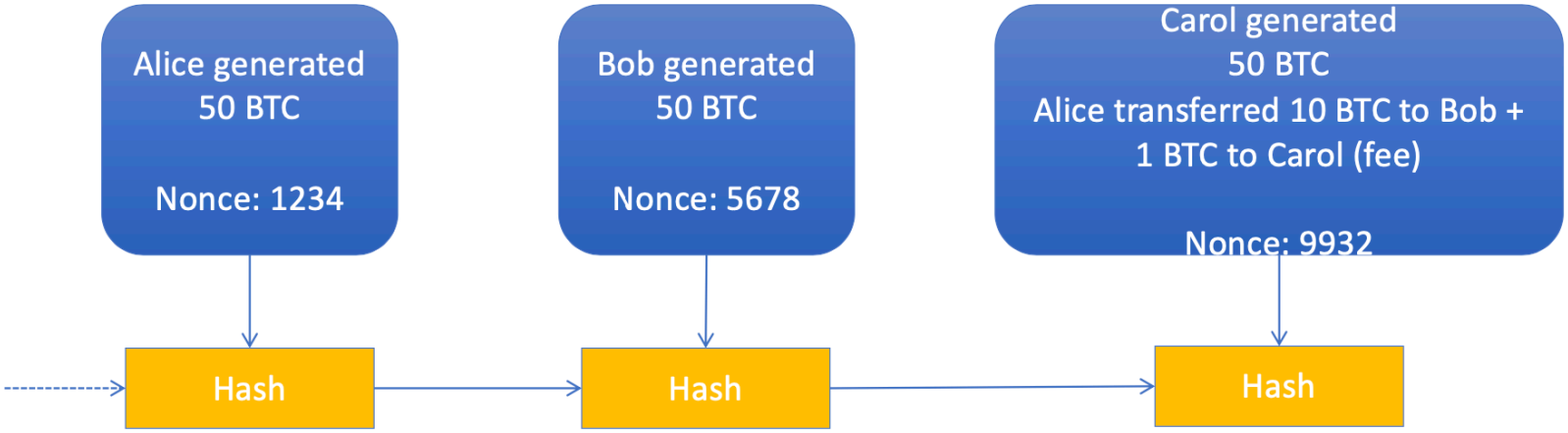
- Every node picks x, computes H(seed || x)
- Closest to 0 wins
- What to use as a seed?
- Hash of:
  - Previous log
  - Node identifier
  - New messages to add to log

- *How to find "closest to 0"?*

Iterated Hashing / Proof of work

- Repeat:
  - Pick random x, compute y = H(seed || x)
  - If y<T, you win!
- Set threshold T so that on average, one winner every few minutes
- Given a *solution*, x such that H(seed || x) < T, anyone can *verify* the solution in constant time (microseconds).

Chaining the blocks



Account	Balance
Alice	39 BTC
Bob	60 BTC
Carol	51 BTC

Protocol Overview

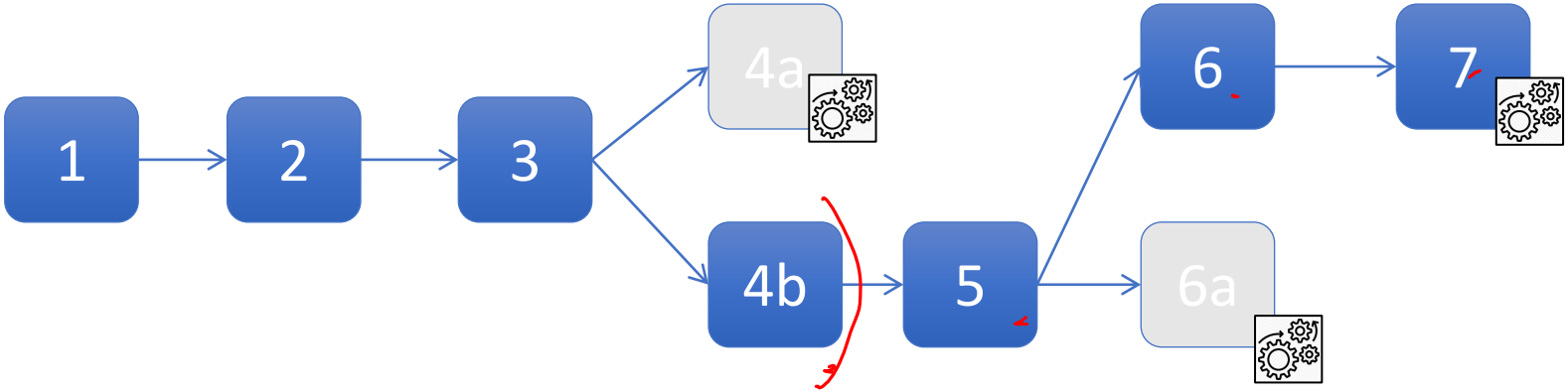
- New transactions broadcast to all nodes.
- Each node collects new transactions into a block.
- Each node works on finding a proof-of-work for its block to become its leader and get it appended to a chain.
  - i.e. finds x, such that $H(\text{seed} \,||\, x) < T$.
- When a node finds a proof-of-work, it broadcasts it to all nodes.
- Nodes accept a block only if all transactions in it are valid.
- Nodes express their acceptance by working on creating the next block in the chain, using the hash of accepted block as previous hash.

What could go wrong?

- Two nodes may end up mining different versions of the next block.

Longest Chain Rule

- Two nodes may end up mining different versions of the next block.
- A node may receive two versions of the next block.
- Will store both, but work on the first one they receive.
- Over time, more blocks will be received.
- The node will switch to working on the *longest chain*.



When is a transaction committed?

- Wait for upto k more blocks to be added in the chain.
- Then commit the transaction.
- k is set to 6 for Bitcoins.

Security Property

- Majority decision is represented by the longest chain.
  - It has greatest "proof-of-work" invested in it.

- If majority of CPU power is controlled by honest nodes, the honest chain will grow fastest and outpace competing chains.

- To modify past blocks, an attacker will need to redo the proof-of-work for that block, and all blocks after it, and then surpass the work of honest nodes.

- Probability of attack reduces as more blocks get added.

Incentives for Logging

- Security better if more people participated in logging.

- Incentivize users to log *others'* transactions
  - Transaction fees: e.g. pay me x% to log your data (or some fixed fee per transaction)
  - Mining reward: each block *creates* bitcoins

Logging Speed

- How to set T?
  - Too small: slows down transactions
  - Too big: wasted effort due to chain splits

- Periodically adjust difficulty T such that one block gets added every 10 minutes.
  - Depends on hardware speed (which typically improves over time) and number of participants (which vary over time).

- Determined algorithmically based on the rate at which blocks are mined
  - Target is 1 block every 10minutes.
  - Difficulty recomputed after every 2016 blocks.

Bitcoin Broadcast

- Need to broadcast:
  - Transactions to all nodes, so they can be included in a block.
  - New blocks to all nodes, so that they can switch to longest chain.

- What if we use R-multicast?
  - Have to send O(N) messages
  - Have to *know* which nodes to send to
  - Not a suitable choice.

Gossip / Viral propagation

- Each node connects to a small set of *neighbors* (10–100).
- Nodes propagate transactions and blocks to neighbors.
- Push method: when you hear a new tx/block, resend them to all (some) of your neighbors (flooding).
- Pull method: periodically poll neighbors for list of blocks/tx's, then request any you are missing.
- Unreliable: some nodes may not receive all transactions or all blocks. But that's ok.

Maintaining Neighbors

- A *seed* service
  - Gives out a list of random or well-connected nodes
  - E.g., seed.bitnodes.io
- Neighbor discovery
  - Ask neighbors about *their* neighbors
  - Randomly connect to some of them

Bitcoin Summary

- Unreliable broadcast using gossip
- Probabilistic "leader" election for mining blocks (tx ordering)
- Longest chain rule to ensure long-term (probabilistic) consistency and security

- Compared with Paxos/Raft:
  - Scales to thousands of participants, dynamic groups
  - Tens of minutes to successfully log a transaction (vs. milliseconds)

MP2: Raft Leader Election and Log Consensus

- Lead TA: Aman Khinvasara

- Objective:
  - Implement a leader-based consensus protocol for replicated state machine, that maintains log consensus even when nodes crash or get temporarily disconnected.

- Task:
  - Beef up a skeleton code provided to you to implement Raft leader election and log consensus.
  - We provide an emulation framework and a test suite.
  - Strive to pass all the test cases provided in our test suite.

MP2: Logistics

- Due on April 8th.
  - Late policy: Can use part of your 168hours of grace period accounted per student over the entire semester.
- Must be implemented in Go.
  - The framework we provide is in Go.
- Read the specification and the comments in the provided code carefully.
- **Start early!!**