Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements for the materials: Indy Gupta and Nikita Borisov

Logistics

- Informal Early Feedback form: shared on Campuswire, post #257.
- HW3 due date extended to Monday, March 30th.
- Hope you all have started working on MP2!
  - Due on April 10th.

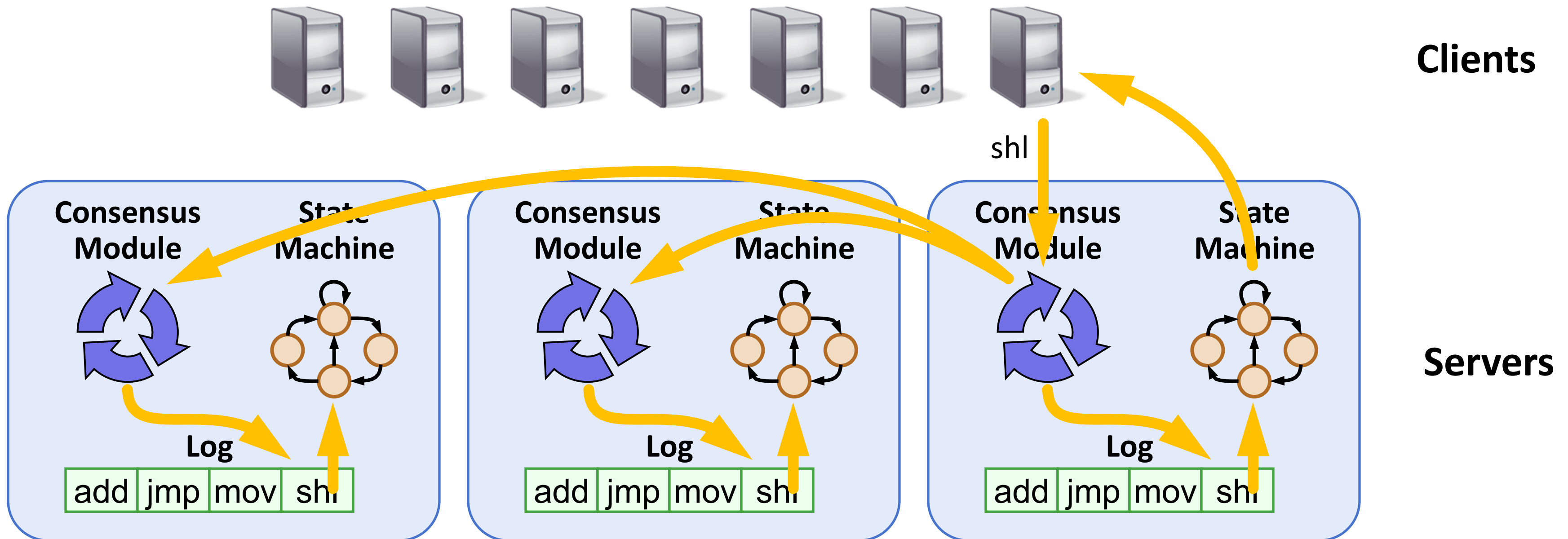
Agenda for today

- 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: A Consensus Algorithm for Replicated Logs

Slides from Diego Ongaro and John Ousterhout, Stanford University

Goal: Replicated Log

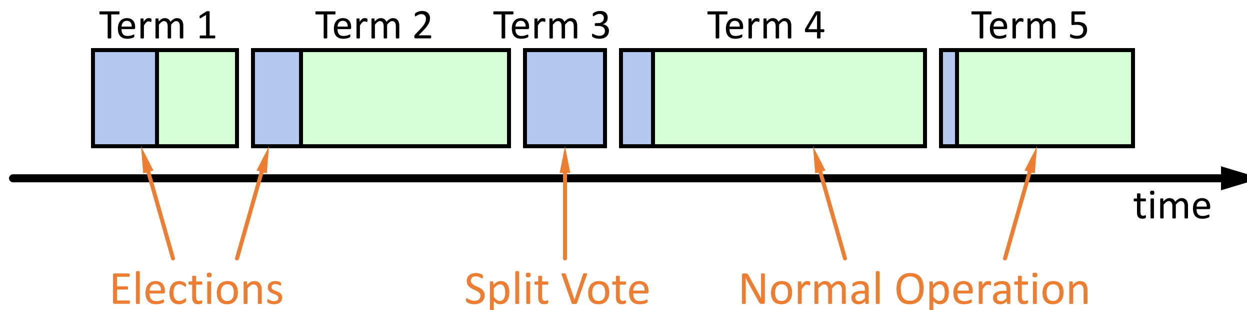


- Replicated log => replicated state machine
  - All servers execute same commands in same order
- Consensus module ensures proper log replication
- **System makes progress as long as any majority of servers are up**
- **Failure model: fail-stop (not Byzantine), delayed/lost messages**

Raft Overview

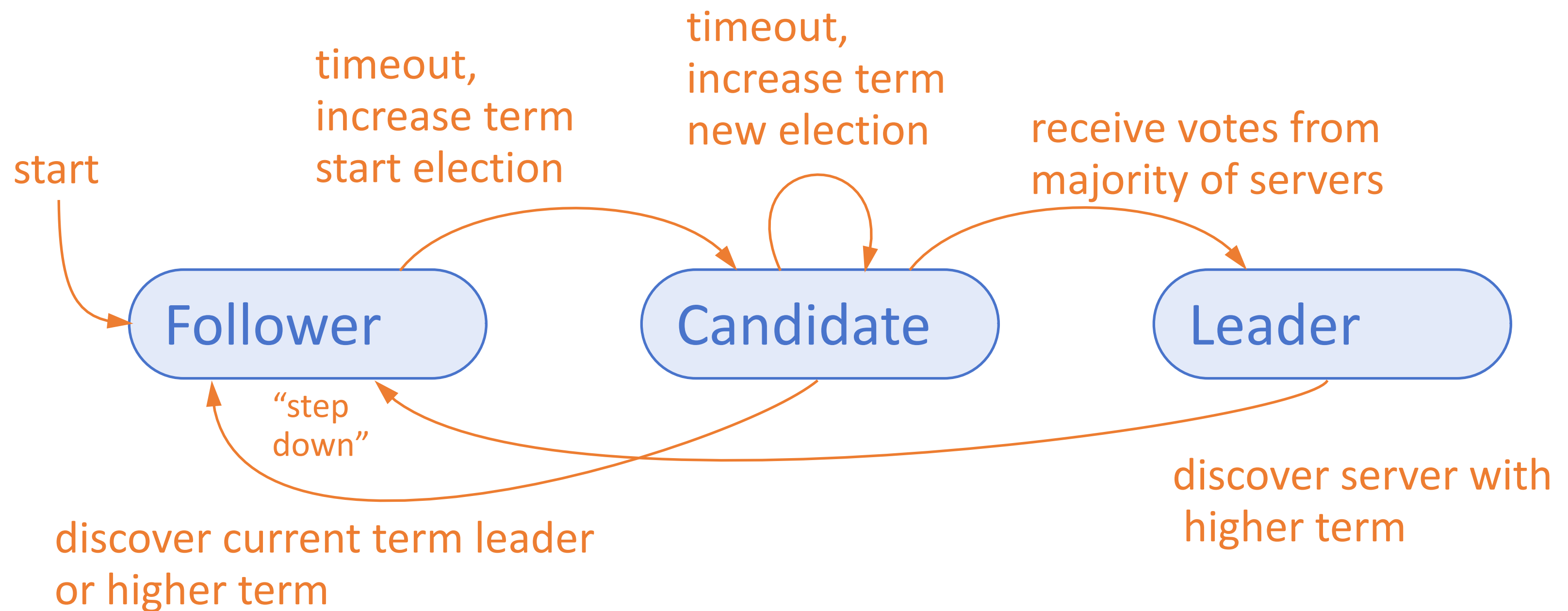
1. Leader election:
   - Select one of the servers to act as leader
   - Detect crashes, choose new leader
2. Neutralizing old leaders
3. Normal operation (basic log replication)
4. Safety and consistency after leader changes

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

State Diagram Revisit



Implication of terms

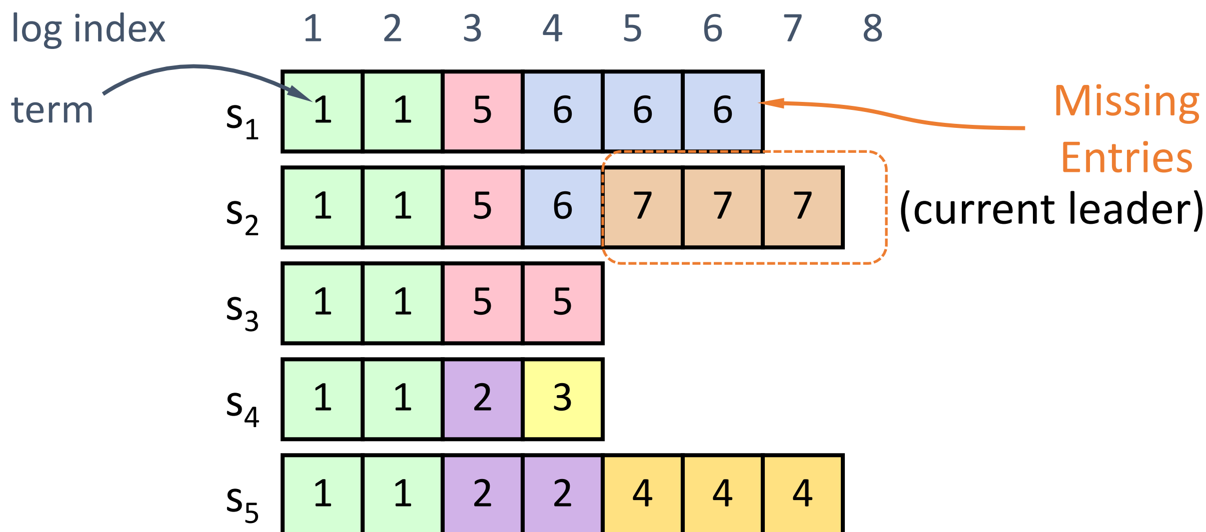
- Each term has at most one leader (safety condition).
- Terms always increase with time.
- If the latest term has an elected leader, majority of processes must have updated themselves to the latest term.
- Only the leader of the latest term can commit log entries (we will discuss this next).

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

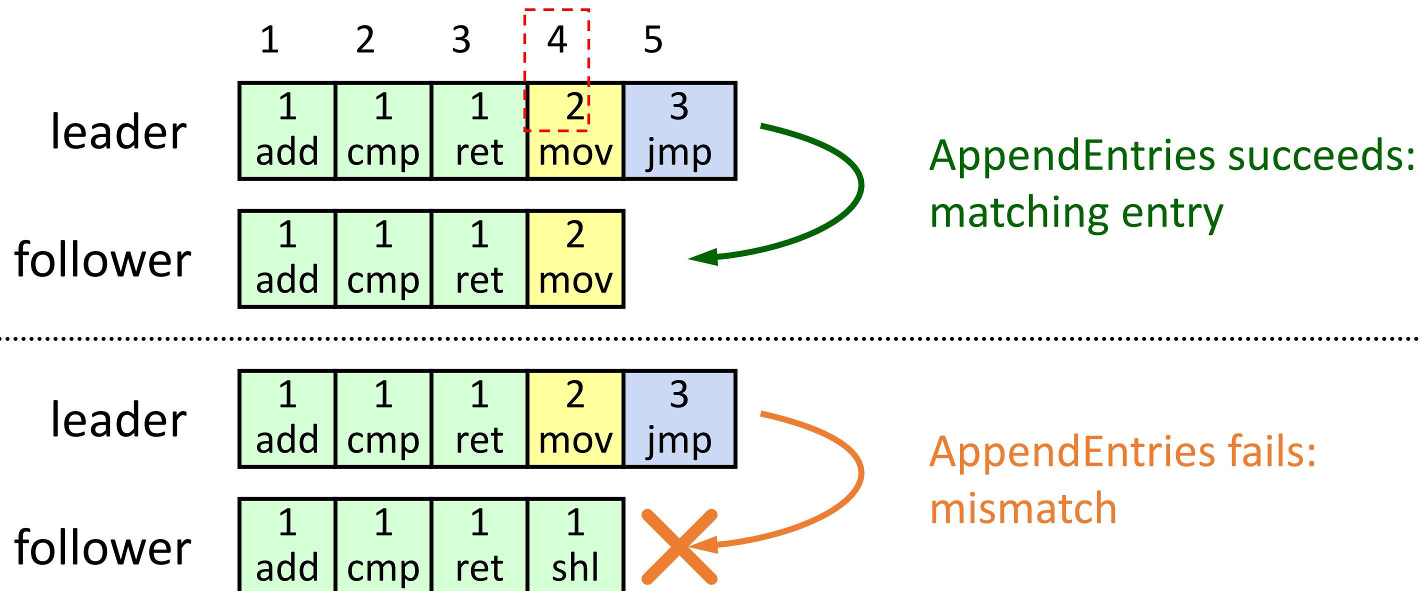
Leader Changes

- At beginning of new leader's term:
  - Old leader may have left entries partially replicated.
  - Multiple crashes can leave many extraneous log entries.
  - 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**
    - *Unless a new leader gets elected during the process.*



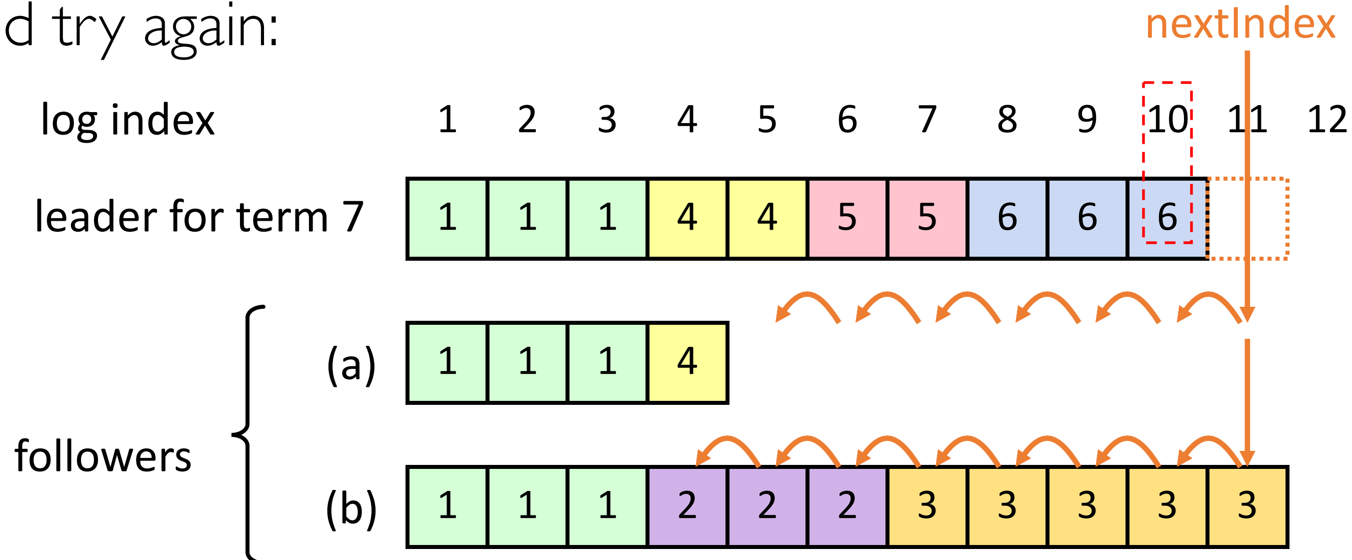
AppendEntries Consistency Check

- Each AppendEntries RPC contains index and term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request
- Implements an induction step, ensures coherency



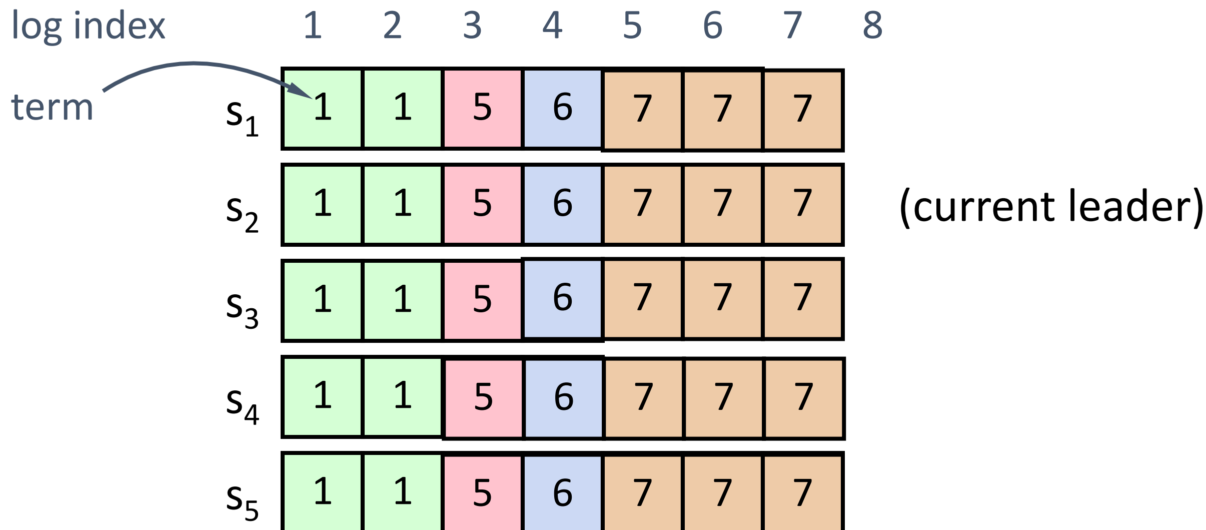
Repairing Follower Logs

- 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:



Log Inconsistencies and Repair

- Leader's log is "the truth"
- Will eventually make follower's logs identical to leader's



Log Consistency

High level of coherency between logs:

Raft guarantees that:

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

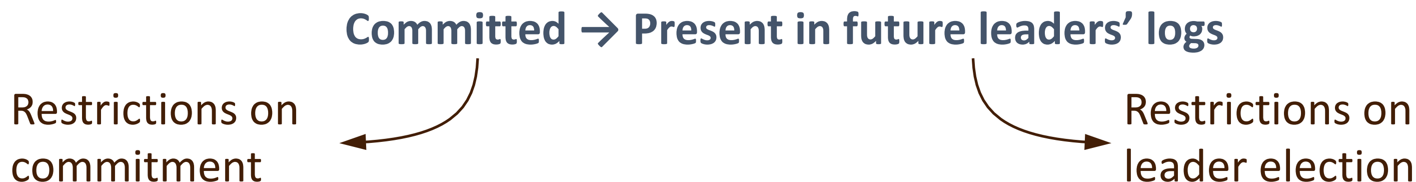
1	2	3	4	5	6	
1 add	1 cmp	1 ret	2 mov	3 jmp	3 div	
1 add	1 cmp	1 ret	2 mov	3 jmp	4 sub	4 add

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

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



Picking the Best Leader

- 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 *up-to-date*":

    $(lastTerm_V > lastTerm_C)$ ||
    $(lastTerm_V == lastTerm_C)$ && $(lastIndex_V > lastIndex_C)$

    - $lastTerm_V$ is the term in the last log entry of voting server V
    - $lastTerm_C$ is the term in the last log entry of candidate C.
    - $lastIndex_V$ is the index of the last log entry of voting server V
    - $lastIndex_C$ is the index of the last log entry of candidate C.

Election Basics: handling RequestVote RPCs

- Suppose a server S in term currentTerm has voted for process with id votedFor in that term.
- When it receives RequestVote RPC from process candidateId with term voteRequestTerm:

  If voteRequestTerm < currentTerm

  Reply false, return.

  If voteRequestTerm > currentTerm

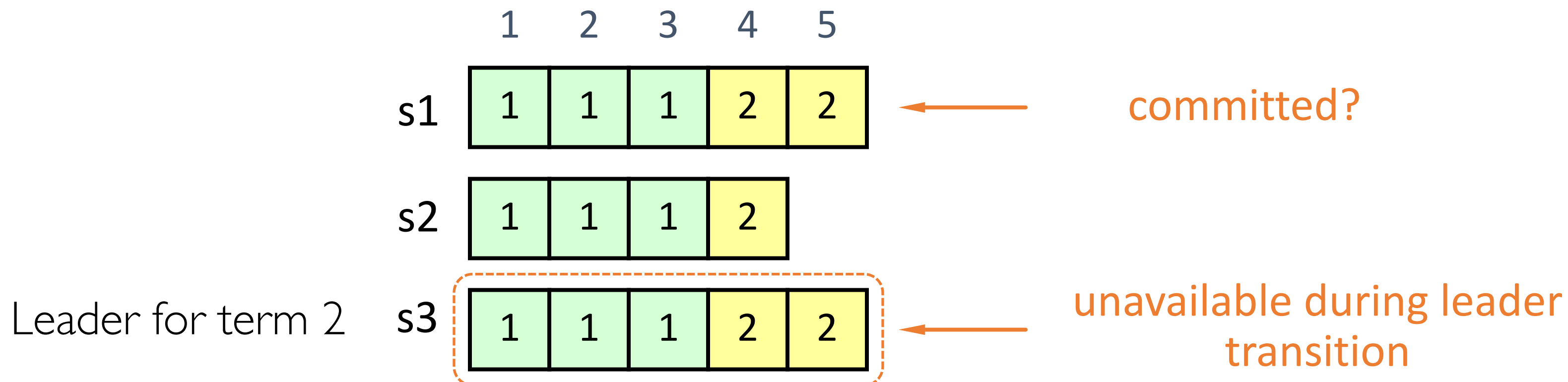
  currentTerm = voteRequestTerm, votedFor = null

  If (candidate's log is at least as *up-to-date* S's log) and (votedFor is null or candidateId)

  Grant vote, votedFor = candidateId

Picking the Best Leader

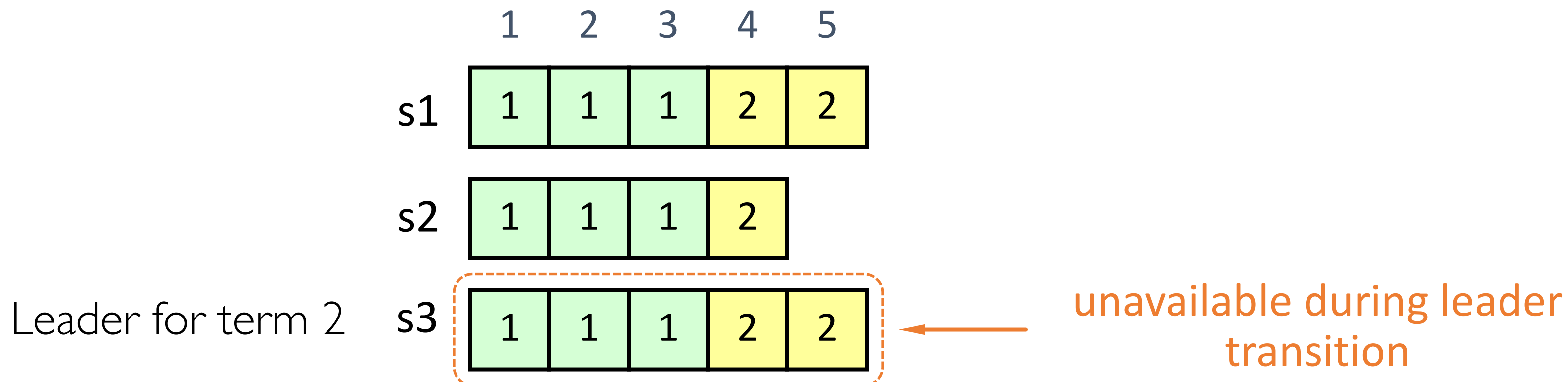
- During elections, choose candidate with log most likely to contain all committed entries
  - Candidates include log info in RequestVote RPCs (index & term of last log entry)
  - Voting server V denies vote if its log is "more *up-to-date*":
    $(lastTerm_V > lastTerm_C) \,||$
    $(lastTerm_V == lastTerm_C) \,\&\&\, (lastIndex_V > lastIndex_C)$



Can s2 be elected leader for term 3? No

Picking the Best Leader

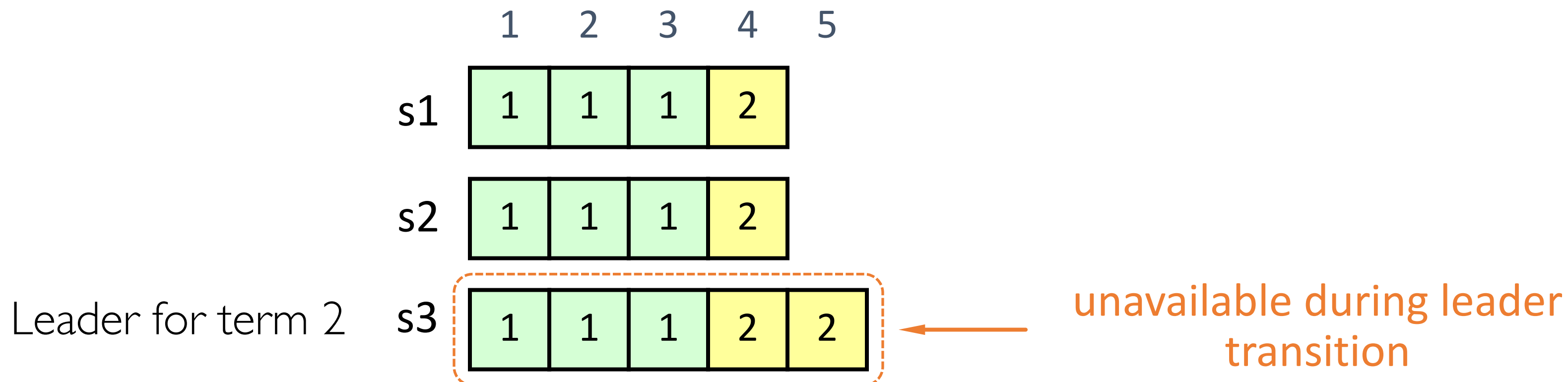
- During elections, choose candidate with log most likely to contain all committed entries
  - Candidates include log info in RequestVote RPCs (index & term of last log entry)
  - Voting server V denies vote if its log is "more *up-to-date*":
    $(lastTerm_V > lastTerm_C)$ ||
    $(lastTerm_V == lastTerm_C)$ && $(lastIndex_V > lastIndex_C)$



Can s1 be elected leader for term 3? Yes

Picking the Best Leader

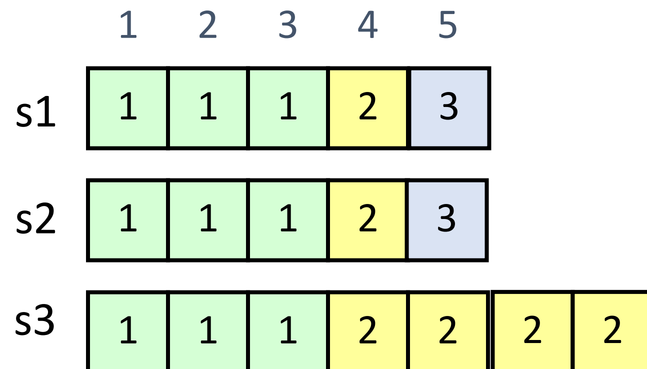
- During elections, choose candidate with log most likely to contain all committed entries
  - Candidates include log info in RequestVote RPCs (index & term of last log entry)
  - Voting server V denies vote if its log is "more *up-to-date*":
    $(lastTerm_V > lastTerm_C)$ ||
    $(lastTerm_V == lastTerm_C)$ && $(lastIndex_V > lastIndex_C)$



Can s2 be elected leader for term 3? Yes

Picking the Best Leader

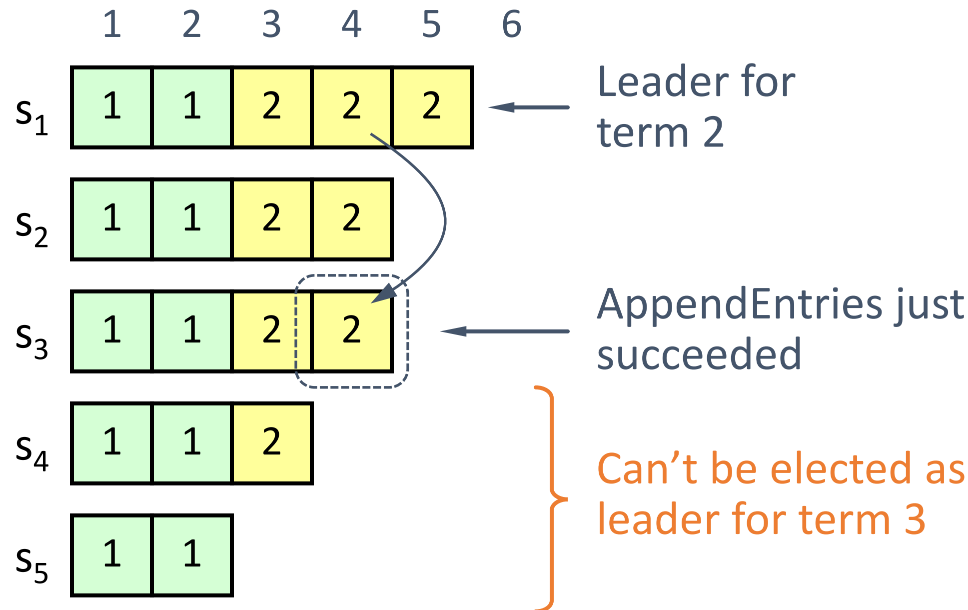
- During elections, choose candidate with log most likely to contain all committed entries
  - Candidates include log info in RequestVote RPCs (index & term of last log entry)
  - Voting server V denies vote if its log is "more *up-to-date*":
    $(lastTerm_V > lastTerm_C) \mid\mid$
    $(lastTerm_V == lastTerm_C) \&\& (lastIndex_V > lastIndex_C)$



Can s3 be elected leader for term 4? No

Committing Entry from Current Term

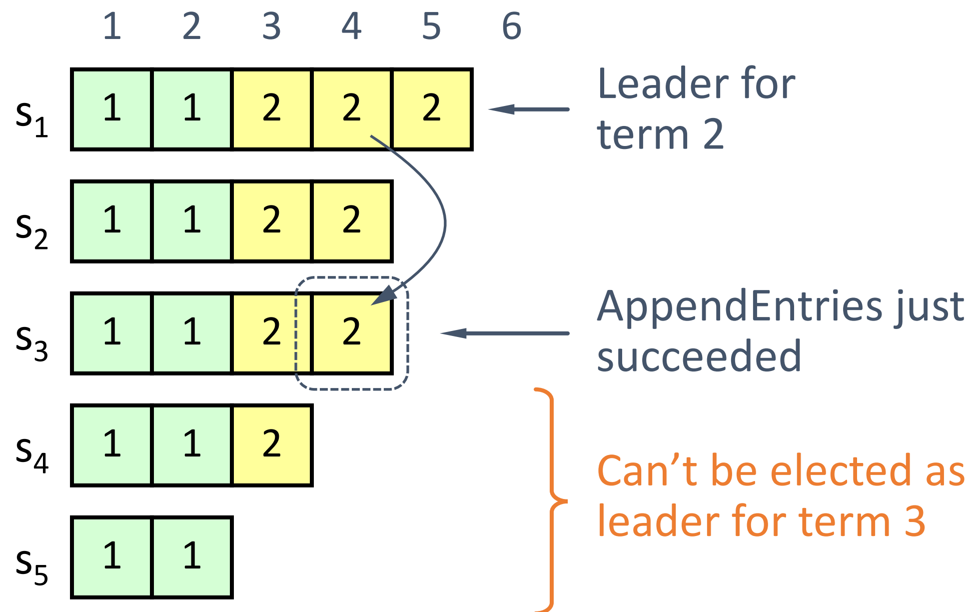
- *When can a leader commit entries?*



- Can the entry in index 4 be committed?
  - Term 2 leader can decide to commit entry in index 4
  - Safe: leader for term 3 must contain entry 4

Committing Entry from Current Term

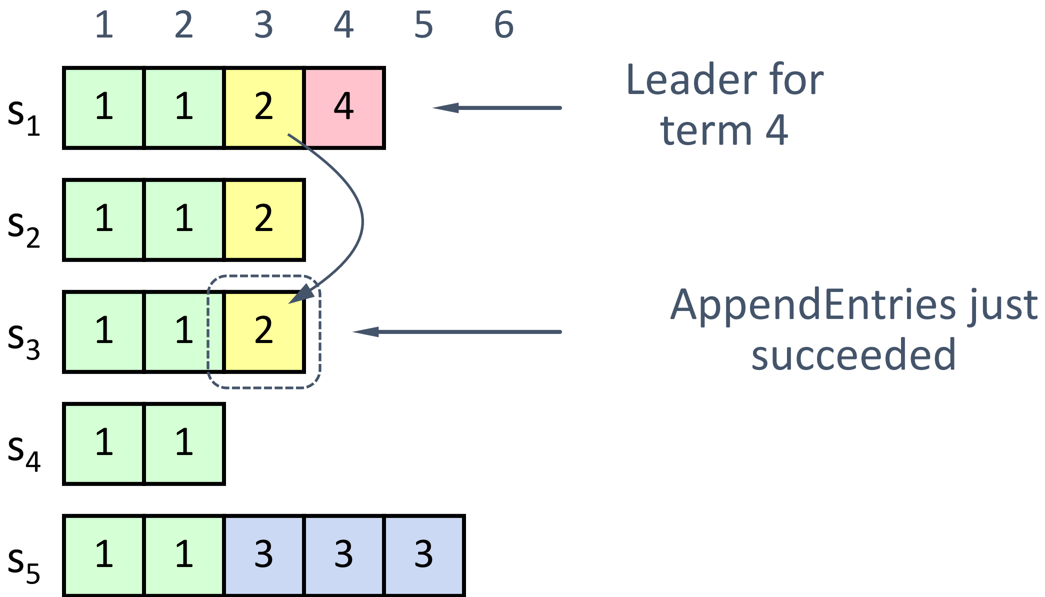
- *When can a leader commit entries?*



- What about committing entry in index 5?
  - Cannot be committed (could be overwritten if s2 or s3 become leader for term 3).
- *Perhaps leader can commit an entry once replicated on majority of servers?*

Committing Entry from Earlier Term

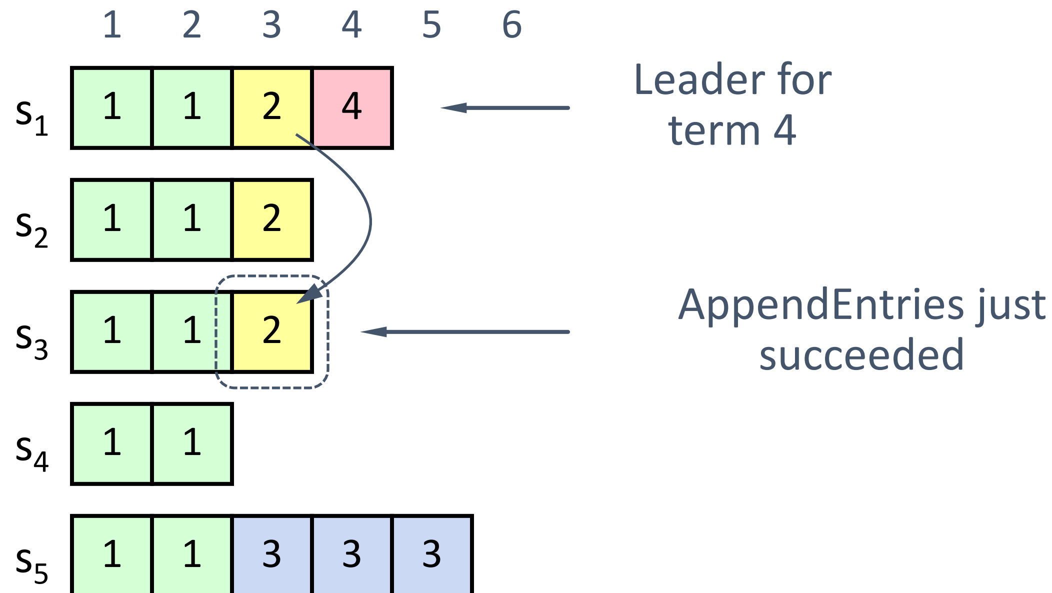
- Leader is trying to finish committing entry from an earlier term



- Is it safe to commit entry 3 with term 2?

Committing Entry from Earlier Term

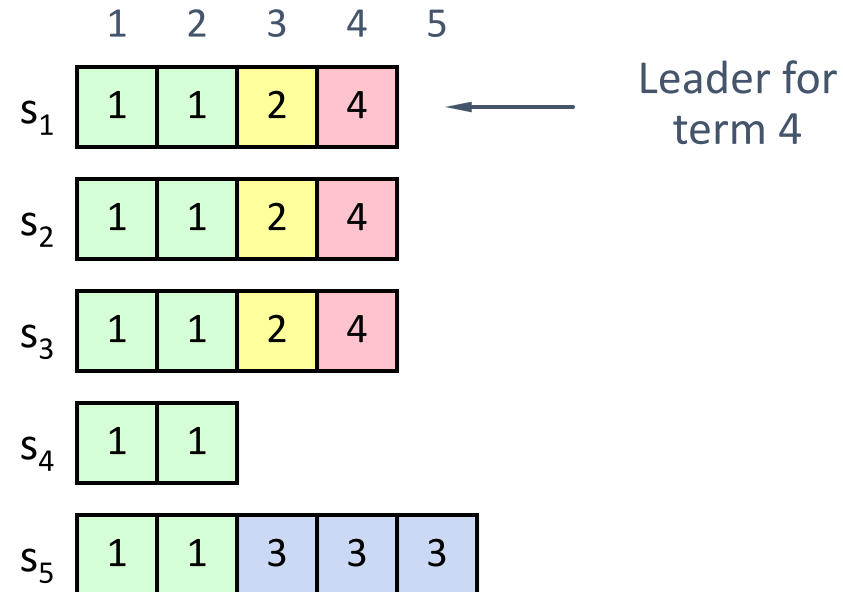
- Leader is trying to finish committing entry from an earlier term



- 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

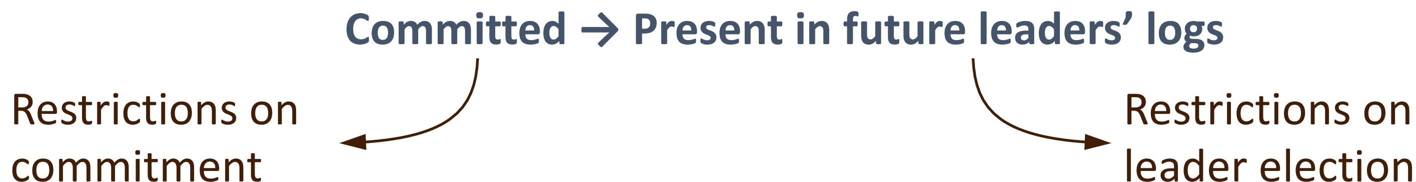
- For a leader to decide an entry is committed:
  - Must be stored on a majority of servers
  - *At least one new entry from leader's current term must also be stored on majority of servers*
- Once entry 4 committed:
  - s_5 cannot be elected leader for term 5
  - Entry 3 (with term 2) and entry 4 (with term 4) both safe to commit.



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

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
  - 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 = one which chose a random number that meets a specific criteria (next slide).

Choosing the random number

- Cryptographic hash function:
  - $H(x) \rightarrow \{0, 1, ..., 2^{256}-1\}$
- 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) less than a threshold 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)
- What to use as a seed?
- Hash of:
  - Previous log
  - Node identifier
  - New messages to add to log
- The appropriate value of x *changes* for each block.
  - Cannot be reused across blocks.
  - Must be recomputed.

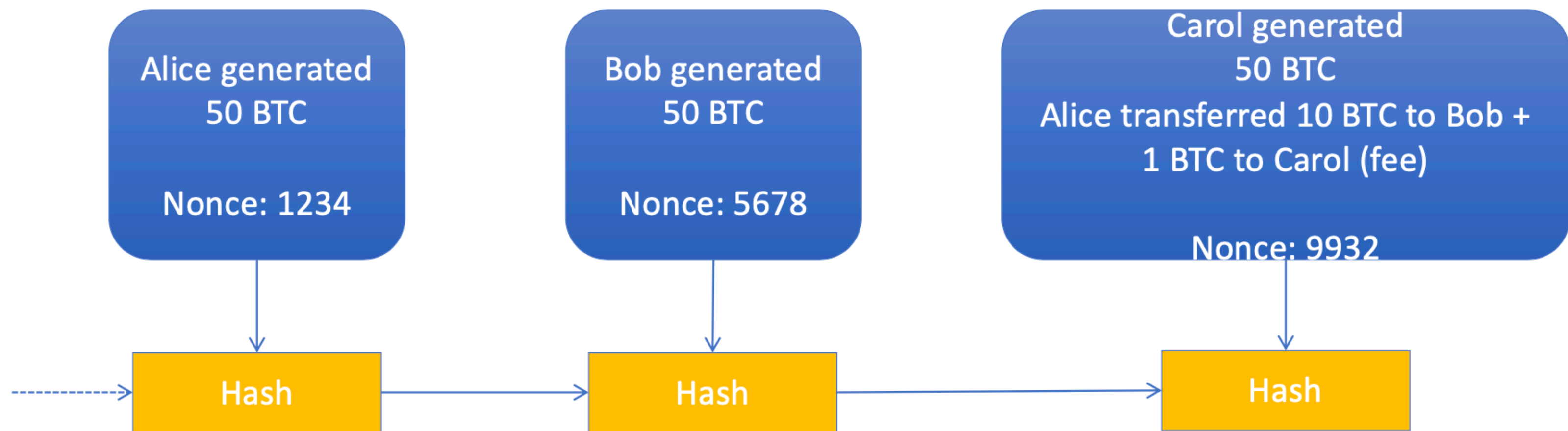
Iterated Hashing / Proof of work

- 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).

Using a seed

- Every node picks x, computes H(seed, x)
- What to use as a seed?
- Hash of:
  - Previous log
  - Node identifier
  - New messages to add to log
- The appropriate value of x *changes* for each block.
  - Cannot be reused across blocks.
  - Must be recomputed.

Chaining the blocks



Account	Balance
Alice	39 BTC
Bob	60 BTC
Carol	51 BTC

To be continued in next class....