ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 30 Consensus II

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

December 3, 2025

Outline

Reading Assignment

- This lecture: Practical Consensus and Raft
  - In Search of an Understandable Consensus Algorithm
    `https://raft.github.io/raft.pdf`
- No final exam

Outline

Why Raft?

- Paxos is the classic protocol for consensus, but
  - Hard to understand
  - Hard to implement correctly
- Raft: a new consensus algorithm
  - A better foundation for system building and education
  - Understandability via decomposition and state space reduction
- Raft features
  - Strong leader, e.g. to distribute log entries
  - Leader election with random timers
  - Membership change via joint consensus

Consensus as Replicated State Machines

- Individual servers compute with the same state machine
  - Same initial state
  - Same inputs and transitions
  - A server could run slower than others and may fail any time.
- Typically implemented as a replicated log
  - Each server stores a log of inputs and state transitions.
  - Consistent replicated logs on servers lead to consensus on computing states.

Desired Properties of Replicated Log

- Safety: every log eventually contains the same inputs and transitions in the same order
  - Under all non-Byzantine conditions like network delays, partitions, and packet loss, duplication, and reordering.
  - Don't depend on performance of individual servers or the network.
- Availability: as long as a majority of servers are up, the system is fully functional.
  - And complete requests as soon as so.
  - When there is a network partitioning, availability only applies to clients connected to the partition with the majority of servers – therefore the system does not provide availability per definition of the CAP Theorem.

Managing Replicated Log via a Leader

- Elect a distinguished leader to manage the replicated log.
  - Take requests from clients to generate a log entries.
  - Replicates log entries to other servers.
  - Tell other servers to apply log entries to update their state machines.
- Problems to solve
  - Leader election: choose a new leader when an existing leader fails – what if some servers consider the leader failed while the others don't?
  - Log replication: leader generate log entries and replicate them to other servers – what if the leader fails before replicating an entry to all servers?
  - State machine safety: once a server applies a log entry, no other server can use a different log entry at the same log index.

Outline

Raft

The Raft Consensus Algorithm

Raft Server States and State Transitions I

- Each server is in one of the three states.
  - *leader*, *follower*, and *candidate*.
  - All servers start as *follower*.
- Raft divides time into terms of arbitrary length.
  - Terms are numbered with consecutive integers.
- Each term begins with an election.
  - When at least one *follower* decides to become a *candidate*.
  - Then these *candidate*s attempt to become *leader*.
- If a *candidate* wins the election,
  - It serves as *leader* for the rest of the term.
  - The term lasts as long as all other servers consider this *leader* as functional.

Raft Server States and State Transitions II

- An election may fail
  - A new term will begin shortly.
- Servers may miss elections or terms.
  - Servers may have different views on what the current term is.
  - Each server S maintains its own current term as **term**(S).
  - Each server S sets **term**(msg)=**term**(S) when sending or broadcasting a message msg.
  - A server drops a message with lower term, i.e. when **term**(S)>**term**(msg).
  - Messages with a higher term, i.e. **term**(S)<**term**(msg), will cause the server to update its **term**(S) and become *follower*.

Leader Election I

- Each server maintains its own election timeout.
- Servers as *leader*s broadcast periodic Heartbeat messages.
- If a server S receives Heartbeat H,
  - Heartbeat H is stale if **term**(S)>**term**(H) – drop H.
  - Otherwise, S is or will become *follower*. Then S will reset its election timeout.
- If a *follower* S reaches its own election timeout,
  - No Heartbeat was received recently – the *leader* fails!
  - S becomes *candidate* and starts a new term by adding 1 to **term**(S) before broadcasting a RequestVote message.
  - Then S resets its election timeout.
- If a server S receives RequestVote R from a *candidate*,
  - RequestVote is stale if **term**(S)≥**term**(R) – drop R.
  - Otherwise, S updates **term**(S) and votes for the *candidate*.
  - S won't vote two different *candidate*s for the same term!

Leader Election II

- If a *candidate* S receives votes from a majority of the servers,
  - S becomes the only *leader* for **term**(S).
  - Until S hears a larger **term**(msg) from another message. Then S should update **term**(S) and become *follower*.
- If a *candidate* S receives a message from a *leader*,
  - The *leader* is stale if **term**(S)>**term**(msg) – drop the message.
  - Otherwise a *leader* is elected. Then S should update **term**(S), return to *follower*, and reset its election timeout.
- If a *candidate* reaches its own election timeout,
  - Start a new term the same way as a *follower*.
- Is it possible for *candidates* to always have election timeouts?
  - Yes it is possible and there will be terms without a leader indefinitely – this makes the whole system unavailable.
  - Raft uses randomized election timeouts so more likely some *candidate* will win the election.

Log Replication I

- A elected *leader* will start to serve clients immediately.
  - Convert client requests into log entries and replicate them via AppendEntries messages.
  - Each log entry contains the term of the *leader* and an index for its position with the log.
- What if multiple *leader*s serve clients at the same time?
  - It is possible but these *leader*s will have different terms.
  - Log replication will ensure only the *leader* with the highest term will be able to serve clients successfully.
- Following the order of each log index, the *leader* wait for a majority of *follower*s to confirm replication of each log entry.
  - Now it is safe to apply the entry to compute the next state.
  - The *leader* will declare this entry as committed and broadcast.

Log Replication II

- How *followers* confirm replication of log entries?
  - What if log at a *follower* diverge from that of the *leader*?
- Assume a server S receives AppendEntries A from a *leader*.
  - The *leader* is stale if `term`(S)>`term`(A) – drop A.
  - Otherwise, S is or will become *follower*, updating `term`(S) to be the same as `term`(A) if necessary.
- The AppendEntries message will contain `prevLogIndex` and `prevLogTerm` for S to perform consistency check
  - When there is no failure, the last log entry of S should match `prevLogIndex` and `prevLogTerm`. S will append the new entries and confirm with the *leader*.
  - Otherwise the log diverge – S should resolve the conflict, possibly with additional help from the *leader*.

Log Replication III

- When the last log entry of S doesn't match `prevLogIndex` and `prevLogTerm` from AppendEntries
  - Case 1: log entries of S doesn't reach `prevLogIndex` yet.
  - Case 2: S has a log entry at `prevLogIndex` but with different `prevLogTerm`.
  - Case 3: S has a log entry at `prevLogIndex` with the same `prevLogTerm` but it is not the last entry.
- For Case 3, S overwrites any entries beyond `prevLogIndex` with those from AppendEntries and confirm with the *leader*.
- For Case 1 and 2, S needs more information from the *leader*.
  - It will explicit reject AppendEntries with the *leader*.
  - The *leader* will then retry an updated AppendEntries with a smaller `prevLogIndex` that includes more entries in the past.
  - This retrying process iterates until S could confirm with this or a future *leader*, possibly elected at a future time.
  - This also handles cases when servers fail – they restart into *followers* and obtain missing log entries from current *leader*.

Revised Leader Election for Safety

- What guarantees *leader* to have up-to-date log entries so it can update *followers*?
- Is it possible for a *follower* with diverged log from the majority of the servers to become *candidate* and then *leader*?
  - We should prevent so since Raft doesn't allow to update a *leader*'s log from those of *followers*.
- Revised voting process
  - A *candidate* should include information about its log in RequestVote.
  - For a server to vote the *candidate*, in addition to previous condition to vote, the server needs to make sure the *candidate*'s log is as up-to-date as its own log.
  - Logs whose last entry has highest term will be the most up-to-date one. If the last entries have the same term, the longest log is the most up-to-date.

Membership Changes

- Adding and removing servers change majority.
  - Need to prevent different servers to have different views on what servers are needed for majority.
  - Changing other cluster configurations will cause similar issues.
  - This is another consensus problem Raft needs to solve.
- Two-phase joint consensus
  - Assign each server a unique ID, e.g. a long random string.
  - Denote old set of servers as C_{old} and new set C_{new}.
  - First phase: *leader* appends a special log entry $C_{old,new} = (C_{old}, C_{new})$ to start a configuration update.
  - Second phase: once $C_{old,new}$ has been committed, *leader* appends a log entry C_{new} to complete the update.
  - Servers receiving $C_{old,new}$ will be in the first phase, making all decisions requiring both majority from C_{old} and C_{new}.
  - Then they will move into the second phase after receiving C_{new}, making all decisions requiring majority from C_{new} only.

Summary

- Raft provides a practical solution to consensus
  - Leader election + log replication + safety
  - Widely used in real systems to replace Paxos.
- If you are familiar with blockchain proof-of-work consensus, does Raft look more similar to it than to Paxos?
  - While Raft and Paxos are fundamentally different than blockchain proof-of-work consensus?