ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 29 Consensus I

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

December 1, 2025

Outline

Reading Assignment

- This lecture: Consensus and Paxos
  - Paxos Made Simple
    `https://lamport.azurewebsites.net/pubs/paxos-simple.pdf`
- Next lecture: Practical Consensus and Raft
  - In Search of an Understandable Consensus Algorithm
    `https://raft.github.io/raft.pdf`

Outline

Consensus

Paxos

Consensus

- Consensus: how can multiple parties reach agreement?
  - E.g. to ensure there is a single branch for data management.
  - Assume some parties and communications could be faulty.
  - A fundamental problem of distributed computing and security.
    - If arbitrary faulty behavior is allowed, then one must consider possible attacks by participating parties.
- An example: each party presents a value of 0 or 1, and together they want to agree on the majority.
  - What faulty behavior can you think of?

The Byzantine Generals Problem

- A recast of the previous example by Lamport et al. 1982.
  - Assume arbitrary faulty behavior.
  - Not related to any historical events. But in a more realistic setting for people to reason with possible attacks.
  - A.k.a. Byzantine Fault Tolerance (BFT)
- There is a group of Byzantine generals.
  - Each commands a division of army encircling an enemy city.
  - The generals individually decide if they should attack or not.
  - Together they vote and follow the majority.
- We only care whether the consensus is reached or not – we don't care if they actually attack or not.

Traitors

- However, some of the generals are traitors.
  - Traitors do whatever they want.
  - Traitors may collude.
- The objective of the traitors is to break consensus.
  - E.g. if Alice and Bob are loyal generals and Alice votes yes while Bob votes no, then the traitor Oscar can trick them by sending a vote of yes to Alice and a vote of no to Bob.
- Protocol design: a protocol all loyal generals follow.
  - So that they will reach a common decision after sending each other many messages, usually in rounds.
  - Assume there are at least 2 loyal generals, how many traitors could there be at most?

Some Results with BFT

- If multiple rounds are allowed, for $3m + 1$ generals, there is a protocol to cope with at most m traitors.
  - No protocol can cope with more traitors, e.g. 1 in 3 as our Alice/Bob/Oscar example.
- With digital signatures, a protocol runs $m + 1$ rounds to cope with at most m traitors among any number of generals.
- Limitations
  - Not efficient enough for distributed computing because the need of multiple rounds of communications.
  - If there are unlimited number of traitors, none of the above protocols is secure.
- Cryptocurrencies use more complex algorithms like blockchain proof-of-work consensus but they are still quite costly.
- Can we do better if only certain faulty behaviors need to be addressed, e.g. for servers that simply fail and restart?

Outline

Consensus

Paxos

Paxos

- A consensus protocol first described by Lamport in 1990.
  - A known number of parties follow protocols faithfully, though messages could be lost, delayed, repeated, or reordered.
  - Not related to any historical locations or events.
- A basic (one-shot) Paxos solves a single consensus problem.
- A multi-Paxos repeatedly executes basic Paxos to implement a replicated state machine.
  - So all replicas use the same sequence of state transitions.
  - Used by many cloud services that need to maintain consistency when servers and network fail.

Basic Paxos

- Participating parties are processes.
  - Processes will trust each other's decisions and faulty processes can be treated as faults in message communications.
- Each process will take any among the three roles
  - Proposers: propose candidates of the consensus value, e.g. a state transition, and make a decision on the value after communicating with accepters.
  - Accepters: vote on which among proposed candidates should be accepted as the consensus value, and record decisions from proposers.
  - Learners: observe the decision making process to learn the consensus value.

Proposer and Accepter Actions

1. To start, proposer p sends $prepare(r)$ to all accepters.
   - r needs to be unique.
   - Accepter maintains largest r received as r_{ack}, as well as r_a and v_a as accepted decision from proposers.
     - Initialize r_{ack} and r_a to $-\infty$ and v_a to nil.
2. Accepter receiving $prepare(r)$ from p:
   - If $r > max(r_{ack}, r_a)$, reply $ack(r, v_a, r_a)$ and update r_{ack} to r.
   - Reject/do nothing otherwise.
3. Proposer receiving $ack(r, v_a, r_a)$ from a majority of accepters:
   - If one of the v_a is not nil, find the v_a with the largest r_a and send $accept!(r, v_a)$ to all accepters.
   - Otherwise, proposer send $accept!(r, v)$ to all accepters where v is the proposed candidate.
4. Accepter receiving $accept!(r, v)$:
   - If $r \geq max(r_{ack}, r_a)$, send $accepted(r, v)$ to all learners, and update (r_a, v_a) to (r, v) if $r > r_a$.
   - Reject/do nothing otherwise.

Learner Action

5.a Learners learn the consensus value v when receiving $accepted(r, v)$ from majority of accepters.

5.b Learners may query accepters for their (r_a, v_a) if $accepted(r, v)$ messages are lost.

5.c Learners may query other learners for the consensus value v.

- Is it possible for those $accepted(r, v)$ and (r_a, v_a) to have different v's?

Example: A Single Proposer

1. Proposer sends *prepare*(100)
2. All accepters reply $ack(100, nil, -\infty)$
   - Update r_{ack} to 100. (r_a, v_a) remain $(-\infty, nil)$.
3. If majority of replies arrive, proposer sends $accept!(100, yes)$.
4. Accepters send $accepted(100, yes)$ to learners.
   - Update (r_a, v_a) to $(100, yes)$.
5. Learners then learn "*yes*" from majority of accepters.

- Lost and delayed messages.
  - Before Step 3, if proposer receives less than majority of replies, system will not make any progress.
  - If less than majority of accepters receive $accept!(100, yes)$, system will not make any progress.
  - Using a timer, either proposer decides to restart the process from Step 1, or learners notify (or act as) proposer to do so.

Example: Proposer Restart

1. Proposer sends *prepare*(200)
   - Use an increasing r to make progress.
2. Accepters reply $ack(200, nil, -\infty)$ or $ack(200, yes, 100)$
   - Depend on whether proposer sends or they receive $accept!(100, yes)$ from the first time.
   - Update r_{ack} to 200. (r_a, v_a) unchanged.
3. If majority of replies arrive,
   - With an $ack(200, yes, 100)$, proposer sends $accept!(200, yes)$
   - With all $ack(200, nil, -\infty)$, proposer may change mind and sends $accept!(200, no)$.
4. All accepters receive the same *accept*! message.
   - Notify learners and update (r_a, v_a) accordingly.

- Lost and delayed message *accept*!
  - Only matter for $accept!(200, no)$ as some accepters may have $(r_a, v_a) = (100, yes)$ while others have $(200, no)$ or $(-\infty, nil)$.
  - Will learners learn different values?

Example: Consensus

- Possible accepter state (r_a, v_a)
  - $(200, no)$: those received the second *accept*!
  - $(100, yes)$: those missed the second *accept*!
  - $(-\infty, nil)$: those missed the first *accept*!
- The choice of "*no*" indicates there is majority of accepters replying $ack(200, nil, -\infty)$ in Step 2.
- Less than majority of accepters have $(100, yes)$ from the first time and learners will not learn "yes".

5. Learners can only learn "no" or proposer restarts the process again if many messages are lost.

Repeated or Reordered Messages

- Reordered *prepare*(r), *accept*!(r, v), and *accepted*(r, v) messages are rejected based on r.
  - r need to be unique.
  - Proposer need to use increaing r's to make progress.
- Repeated *prepare*(r) messages are rejected.
- Repeated *accept*!(r, v) and *accepted*(r, v) messages are idempotent.
- Proposer keeps records to reject repeated or reordered *ack* messages.

Multiple Proposers

- Same as if there is only one proposer that,
  - Restart and change mind frequently.
  - Forget to use an increasing r when restarting.
  - With lost, delayed, and reordered messages.
- It is possible for multiple proposers to prevent each one from making progress.
  - An exponential backoff strategy may be used by proposers to ensure progress.

Summary

- Consensus protocols ensure parties to reach agreements despite failures in the system.
- Different assumptions on failures result in very different consensus protocols designs.
- Still, Paxos is difficult to understand and implement.