Distributed Systems: Consensus

Lecture 12b Srinivas Narayana http://www.cs.rutgers.edu/~sn624/553-S25



Distributed Systems in Internet Services

Coordination services

[>>))

Distributed consensus

Distributed storage

Load management

Distribution & its consequences



- Need replication to tackle machine failures and unavailability
- Q: How to make data consistent across replicas?
- One option: weak consistency models
 - Example: eventually consistent
 - Complex mental models: e.g., read, write, read
 - Some data is too critical to be out of sync, e.g., Kubernetes cluster state
 - Don't sacrifice correctness at any point, e.g., financial systems
- Strong consistency: all reads (e.g., clients) return the same value
 - Operations ordered globally; each read returns the result of the latest write

CAP Theorem



- (Strong) consistency: each read receives the most recent write
- Availability: read to any non-failed node returns a response
- Partition tolerance: continue to operate despite arbitrary message failure between nodes
- CAP theorem: Can't get all three: Pick 2
- You really can't choose CA in a practical system
 Network partitions will happen; design systems to tolerate them
- Result: give up either strong consistency or availability

Agreement on (meta)data that is distributed: **CONSENSUS**

Primitives to build Distributed Systems

- Clients may contact any replica or a distinguished replica
- Replicated state machines
 - Replicated configuration and data stores (e.g., etcd)
- Leader election
- Distributed coordination & locking
 - e.g., barriers between different computation phases (workflow)
- Reliable distributed queueing and messaging





Simple solutions distributed consensus?

- Simple timeouts to determine a primary
 - "Split brain" problem: data corruption
 - Data unavailable if conservative
- Problem amplified in clusters
 - Disagreement in group membership
 - Data corruption with separate leaders per group
- Network unavailability is common
 - Slow network
 - Some messages being dropped
 - Messages throttled in one direction

Setup: Agree on one value forever

- Suppose we have N = 2f + 1 nodes
- Want to agree on one value across the distributed system
- Assume no clock synchronization (clocks can drift)
- Asynchronous: the network can delay or drop messages
- Crash recovery
- Nodes execute correctly (not Byzantine)
- Assume a majority of the nodes can reach each other (quorum)
 - Agree on the same value even as nodes crash and recover
- N = 2f + 1
 - With f failures (unavailable), the rest is consistent and available

Simple Agreement: Prepare and Accept

- Single designated acceptor?
 - That acceptor node can fail
- Multiple acceptors?
 - If a majority accept, agreement!
- Proposer Proposer Prepare(V)

• Problems?

Some problems

- Multiple proposers
 - Each proposer may propose different values
 - Each acceptor may hear multiple proposals
- Acceptors may disagree
 - Acceptors don't know if other acceptors agreeing to the same proposal
- Acceptor may hear a different proposal after it accepted one
 - Should it keep its acceptance or change its mind?



Properties we want: Safety

- If a majority of nodes have agreed on one value, that value cannot change in the future
- e.g., using a different majority or the same majority
- We call this value the chosen value

Properties we want: Liveness

- Liveness: majority of nodes must eventually choose a value
- FLP's impossibility result: Deterministic algorithms cannot guarantee liveness under arbitrary network asynchrony (delay and Proce loss)
- Good news: Realistic networks are not that adversarial (practically live protocols can be designed!)

Process 1 sends Prepare message with a new View number and a transaction number. Process 2 responds with a Promise message.

Process 1 sends Accept for its proposal but Process 2 and 3 cannot accept its proposal because Process 3 has Proposed in the interim and Process 2 has promised. Process 1 makes another attempt, with a higher transaction number. Process 2 promises, which means that Process 3's proposal cannot be accepted. The cycle can repeat indefinitely.



Process 3 sends a conflicting Prepare messge, to which Process 2 responds with a Promise message. Process 1 does not receive the message (or it is delayed).

Basic Paxos (1/3)

- Simple operation: a single proposer and (pure) acceptors
- P1: If only one value were proposed, it must be chosen
 - An acceptor must accept the first proposal it receives
- If an acceptor fails, and two proposals were each accepted by roughly half the acceptors, the cluster cannot determine the chosen value
- Hence, want acceptors to be able to accept multiple proposals
 - Proposals include transaction numbers (n); assume unique & increasing
- P2: If a proposal with value v is chosen, every other chosen proposal must have the same value v
 - "Agree on **some** value, not necessarily **my** value". P2 guarantees safety

Basic Paxos (2/3)

- P2a: One way to satisfy P2 is for acceptors to only accept proposals with the same value v once v is chosen
- P2b: One way to satisfy P2a for proposers to only propose proposals with the same value v that was chosen earlier
- P2c: One way to satisfy P2b is for any proposer to ask a set of acceptors (at least a majority responding) which proposal # and values accepted so far
 - These are called promises

Basic Paxos (3/3)

- Acceptor agrees not to accept any proposal < n
- Proposer must use the value from the highest-numbered proposal accepted from the responses, or use its own value if no existing proposals were accepted
- Acceptor can only accept if not promised any proposal > n
- If a majority of acceptors promise, at least one acceptor must have responded with the chosen value and chosen proposal # (inductive argument)
- Subtle: Write information that must not be reneged into persistent storage (disk) for use after crash recovery



Phase 1. (a) A proposer selects a proposal number n and sends a *prepare* request with number n to a majority of acceptors.

(b) If an acceptor receives a *prepare* request with number n greater than that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.

Phase 2. (a) If the proposer receives a response to its *prepare* requests (numbered n) from a majority of acceptors, then it sends an *accept* request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

(b) If an acceptor receives an *accept* request for a proposal numbered n, it accepts the proposal unless it has already responded to a *prepare* request having a number greater than n.

Learning the chosen value

- Read-only consensus: A majority of nodes must respond with a specific accepted value
 - e.g., a client sends a message to all nodes
- Read from a replica guaranteed to be up to date, e.g., leader
- Quorum leases: a set of nodes hold a time-bound lease on a value. Updates must be acknowledged by all of these machines (reduced write performance)

Implementing primitives atop Paxos

• Leader election: The chosen value is the leader

• Replicated State Machines:

- Agree on each command through Paxos: one Paxos instance for each "slot" in the ordered log of commands
- Once all slots up to slot j chosen, can compute state up to log slot j
- Locking, configuration store, etc. can be similarly implemented

Paxos does not guarantee liveness

One solution: pre-determine or elect a distinguished leader

It's easy to construct a scenario in which two proposers each keep issuing a sequence of proposals with increasing numbers, none of which are ever chosen. Proposer p completes phase 1 for a proposal number n_1 . Another proposer q then completes phase 1 for a proposal number $n_2 > n_1$. Proposer p's phase 2 *accept* requests for a proposal numbered n_1 are ignored because the acceptors have all promised not to accept any new proposal numbered less than n_2 . So, proposer p then begins and completes phase 1 for a new proposal number $n_3 > n_2$, causing the second phase 2 *accept* requests of proposer q to be ignored. And so on.

Process 1 sends Prepare Process 1 sends Accept Process 1 makes another message with a new View for its proposal but Process attempt, with a higher number and a transaction 2 and 3 cannot accept its transaction number. Process 2 number. Process 2 responds proposal because Process 3 promises, which means that with a Promise message. has Proposed in the interim Process 3's proposal cannot and Process 2 has promised. be accepted. The cycle can repeat indefinitely. Processes in the consensus group -----Process 3 sends a conflicting Prepare messge, to which Process 2 responds with a Promise

> message. Process 1 does not receive the message (or it is delayed).

Improvements and Considerations

- Agreeing on multiple values and ordering
- e.g., multi-paxos: Elect a distinguished node (leader)
- only execute phase 2 afterwards until failure



Improvements and Considerations

- Disk latency vs. Network latency
- Uneven network latencies and choosing leaders

