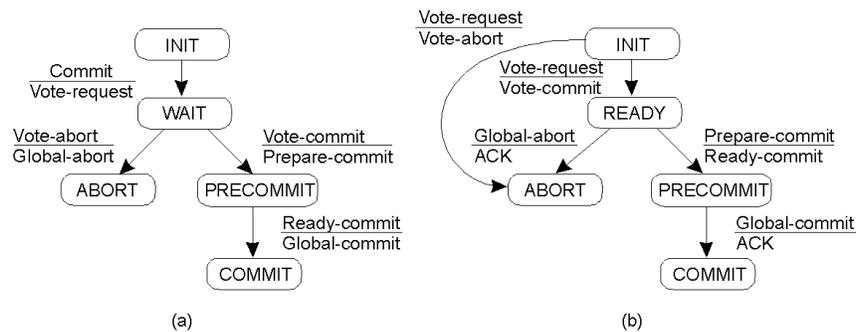


Distributed Consensus

- Part 1: Consensus
- Part 2: Paxos
- Part 3: RAFT

Three-Phase Commit



Two phase commit: problem if coordinator crashes (processes block)

Three phase commit: variant of 2PC that avoids blocking

Replication for Fault Tolerance

- Basic idea: use replicas for the server and data
- Technique 1: split incoming requests among replicas
 - If one replica fails, other replicas take over its load
 - Suitable for crash fault tolerance (each replica produces correct results when it is us).
- Technique 2: send each request to all replicas
 - Replicas vote on their results and take majority result
 - Suitable for BFT (a replica can produce wrong results)
 - 2PC, 3PC, Paxos are techniques

Consensus

- Consensus: get a group of processes to agree on something
- Consensus vs Byzantine Agreement
- Achieve reliability in presence of faulty processes
 - requires processes to agree on data value needed for computation
 - Examples: whether to commit a transaction, agree on identity of a leader, atomic broadcasts, distributed locks
- **4 Properties** of a consensus protocol with **fail-stop** failures
 - Agreement: every correct process agrees on same value
 - Termination: every correct process decides some value
 - Validity: If all propose v , all correct processes decides v
 - Integrity: Every correct process decided at most one value and if it decides v , someone must have proposed v .

2PC, 3PC Problems

- Both have problems in presence of failures
 - **Safety** is ensured but **liveness** is not
- 2PC
 - must wait for all nodes and coordinator to be up
 - all nodes must vote
 - coordinator must be up
- 3PC
 - handles coordinator failure
 - but network partitions are still an issue
- Paxos : how to reach consensus in distributed systems that can tolerate **non-malicious** failures?
 - majority rather than all nodes participate

Paxos: fault-tolerant agreement

- Paxos lets nodes agree on same value despite:
 - node failures, network failures and delays
- Use cases:
 - Nodes agree X is primary (or leader)
 - Nodes agree Y is last operation (order operations)
- General approach
 - One (or more) nodes decides to be leader (aka proposer)
 - Leader proposes a value and solicits acceptance from others
 - Leader announces result or tries again
- Proposed independently by Lamport and Liskov
 - Widely used in real systems (ZooKeeper, Chubby, Spanner)

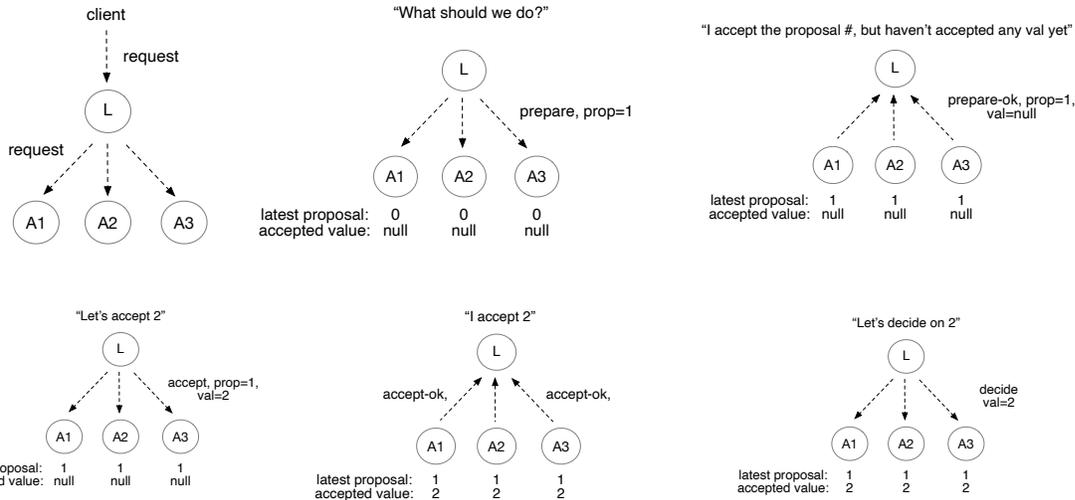
Paxos Requirements

- Safety (Correctness)
 - All nodes agree on the same value
 - Agreed value X was proposed by some node
- Liveness (fault-tolerance)
 - If less than $N/2$ nodes fail, remaining nodes will eventually reach agreement
 - Liveness not guaranteed if steady stream of failures
- Why is agreement hard?
 - Network partitions
 - Leader crashes during solicitation or after deciding but before announcing results,
 - New leader proposes different value from already decided value,
 - More than one node becomes leader simultaneously....

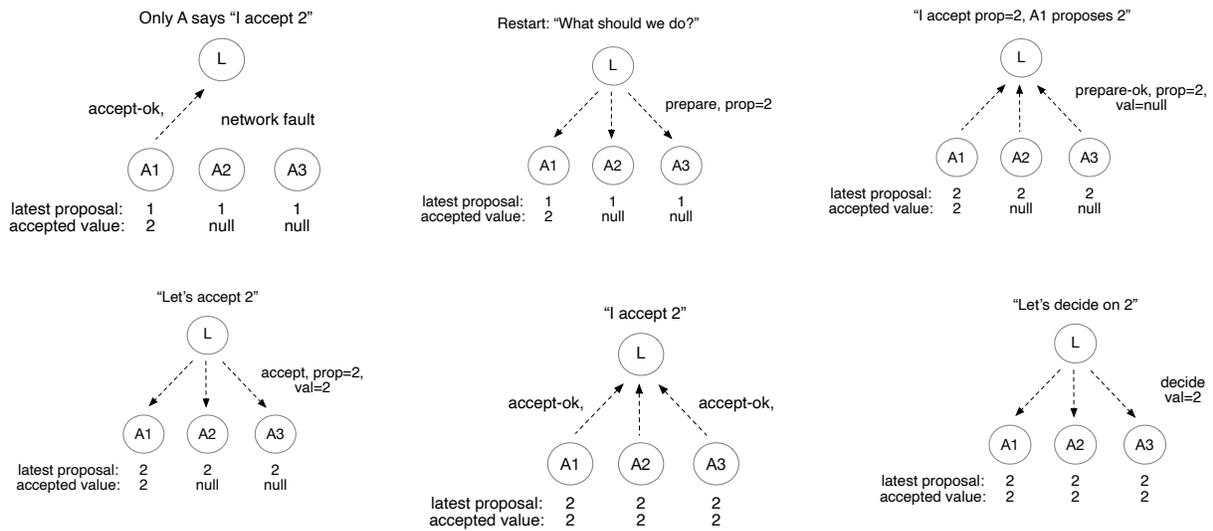
Paxos Setup

- Entities: Proposer (leader), acceptor, learner
 - Leader proposes value, solicits acceptance from acceptors
 - Acceptors are nodes that want to agree; announce chosen value to learners
- Proposals are ordered by proposal #
 - node can choose any high number to try to get proposal accepted
 - An acceptor can accept multiple proposals
 - If prop with value v chosen, all higher proposals have value v
- Each node maintains
 - n_a, v_a : highest proposal # and accepted value
 - n_h : highest proposal # seen so far
 - my_n : my proposal # in current Paxos

Paxos Example



Paxos Example with failure



Paxos operation: 3 phase protocol

- **Phase 1 (Prepare phase)**

- A node decides to be a leader and propose
- Leader chooses $my_n > n_h$
- Leader sends $\langle prepare, my_n \rangle$ to all nodes
- Upon receiving $\langle prepare, n \rangle$ at acceptor
 - If $n < n_h$
 - reply $\langle prepare-reject \rangle$ /* already seen higher # proposal */
 - Else
 - $n_h = n$ /* will not accept prop lower than n */
 - reply $\langle prepare-ok, n_a, v_a \rangle$ /* send back previous prop, value/
 - /* can be null, if first */

Paxos operation

- **Phase 2 (accept phase)**

- If leader gets prepare-ok from **majority**
 - $V =$ non-empty value from highest n_a received
 - If $V =$ null, leader can pick any V
 - Send $\langle accept, my_n, V \rangle$ to all nodes
- If leader fails to get majority prepare-ok
 - delay and restart Paxos
- Upon receiving $\langle accept, n, V \rangle$
 - If $n < n_h$
 - reply with $\langle accept-reject \rangle$
 - else
 - $n_a = n$; $v_a = V$; $n_h = h$; reply $\langle accept-ok \rangle$

Paxos Operation

- **Phase 3 (decide)**

- If leader gets accept-ok from majority
 - Send <decide, v_a> to all learners
- If leader fails to get accept-ok from a majority
 - Delay and restart Paxos

- **Properties**

- P1: any proposal number is unique
- P2: any two set of acceptors have at least one node in common
- P3: value sent in phase 2 is value of highest numbered proposal received in responses in phase 1

Issues

- Network partitions:
 - With one partition, will have majority on one side and can come to agreement (if nobody fails)
- Timeouts
 - A node has max timeout for each message
 - Upon timeout, declare itself as leader and restart Paxos
- Two leaders
 - Either one leader is not able to decide (does not receive majority accept-oks since nodes see higher proposal from other leader) OR
 - one leader causes the other to use its value
- Leader failures: same as two leaders or timeout occurs

Part 3: Raft Consensus Protocol

- Paxos is hard to understand (single vs multi-paxos)
- Raft - *understandable* consensus protocol
- **State Machine Replication (SMR)**
 - Implemented as a replicated log
 - Each server stores a log of commands, executes in order
 - Incoming requests → replicate into logs of servers
 - Each server executed request log in order: stays consistent
- Raft: first elect a leader
- Leader sends requests (log entries) to followers
- If **majority** receive entry: safe to apply → commit
 - If entry committed, all entries preceding it are committed

Log replication

- Servers maintain log of commands: order to perform ops
- Replicated log: replicated state machine (SMR)
 - all servers (replicas) execute commands in log order

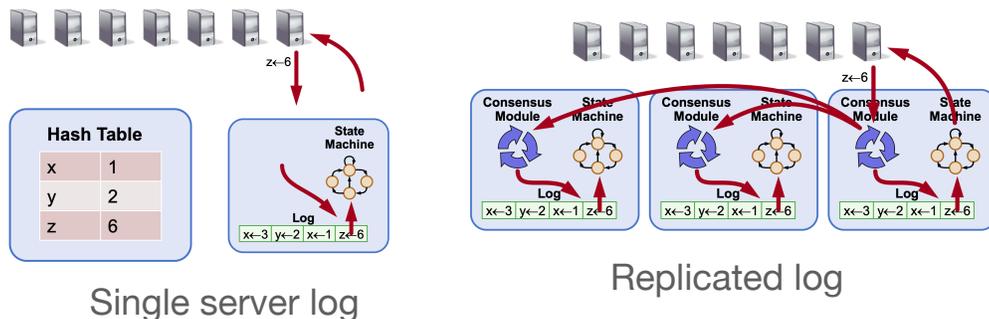


Fig courtesy: D. Ongaro

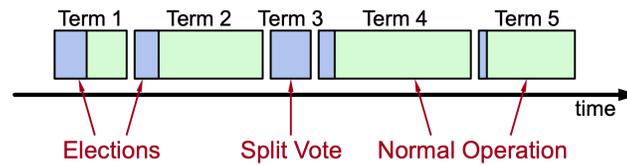
Consensus Approaches

- Leaderless (symmetric)
 - Client can contact any server
- Leader-based (asymmetric)
 - One server is leader and other servers follow the leader
 - Clients contact leader
- RAFT is a leader-based consensus protocol
 - Two aspects: leader changes and normal operation

RAFT Overview

- Leader election
 - Select one server to serve as a RAFT leader
 - detect leader crash, elect new leader
- Normal operation
 - Perform log replication
 - Leader receives client commands, append to log
 - Leader then replicates log to followers
 - Detect and overwrite inconsistencies in log
- Safety
 - Committed log entries are not impacted by leader crash
 - Almost one leader

Terms



- Time is divided into terms
 - Election
 - Normal operation with elected leader
 - New term starts upon leader failure
- At most one leader per term
 - Some terms may have no leader (failed term)
- All servers maintain current term value
- At any time, each server is either:
 - **leader**: receives all client requests and log replication
 - **follower**: passively follows leader
 - **candidate**: participates in leader election

Fig courtesy: D. Ongaro

RAFT Election

- Election timeout: no RPCs received for a while ~100-500ms
- Increment current term and become candidate
- Vote for self (!)
- Send election (RequestVote RPC) message to followers
 - Receive vote from majority: become leader
 - send heartbeat message (AppendEntries RPC)
 - Receive RPC from leader: become follower again
 - Failed election: no majority votes within election timeout
 - Increment term, start new election
- **Safety**: at most one server wins; servers vote once per term
- **Liveness**: someone eventually wins
 - choose random election timeouts; one server times out/wins

Normal RAFT Operation

- Leader receives client commands and appends to log
- Send AppendEntry RPC to all followers
- Once entry safely committed to log
 - execute command and return result to client
- Followers catch up in background
 - Notify followers of committees entries in subsequent RPCs
 - Followers apply committed commands to their state m/c
- Log entry: index, term, command (stored on disk)

index ->	1	2	3	4	5	6	7	8
term	1	1	1	2	3	3	3	3
command	x←3	y←2	x←1	z←6	z←0	y←9	y←1	x←4

Log consistency

- Consistency check: include index, term of prev entry
 - follower must contain matching entry: reject otherwise

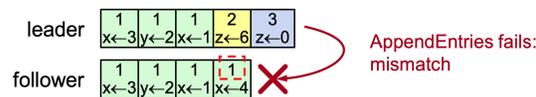


Fig courtesy: D. Ongaro

- Log entries can become inconsistent due to leader failure

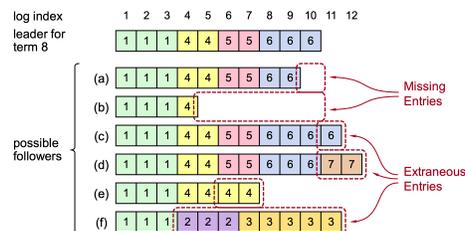


Fig courtesy: D. Ongaro

Log Repair

- Leader tracks nextIndex for each follower
- If AppendEntry check fails, decrement and try again
 - rewind to find match; follower deletes all subsequent entries

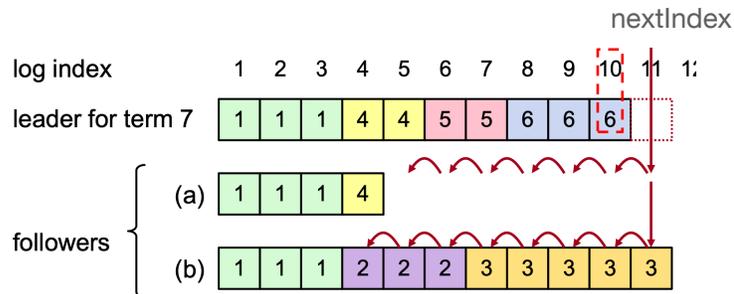
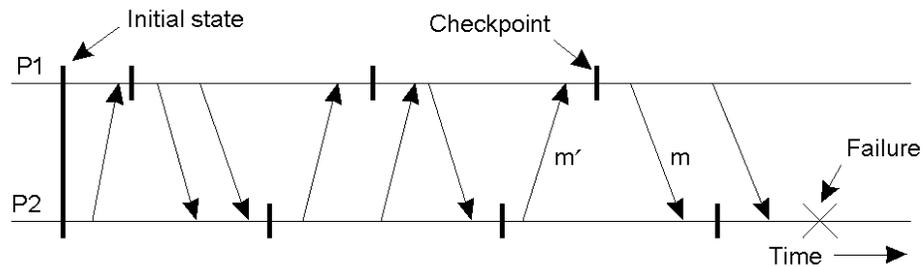


Fig courtesy: D. Ongaro

Recovery

- Techniques thus far allow failure handling
- Recovery: operations that must be performed after a failure to recover to a correct state
- Techniques:
 - Checkpointing:
 - Periodically checkpoint state
 - Upon a crash roll back to a previous checkpoint with a *consistent state*

Independent Checkpointing



- Each processes periodically checkpoints independently of other processes
- Upon a failure, work backwards to locate a consistent cut
- Problem: if most recent checkpoints form inconsistent cut, will need to keep rolling back until a consistent cut is found
- Cascading rollbacks can lead to a domino effect.

Coordinated Checkpointing

- Take a distributed snapshot [discussed in Lec 13]
- Upon a failure, roll back to the latest snapshot
 - All process restart from the latest snapshot

Logging

- Logging : a common approach to handle failures
 - Log requests / responses received by system on separate storage device / file (stable storage)
 - Used in databases, filesystems, ...
- Failure of a node
 - Some requests may be lost
 - Replay log to “roll forward” system state

Message Logging

- Checkpointing is expensive
 - All processes restart from previous consistent cut
 - Taking a snapshot is expensive
 - Infrequent snapshots => all computations after previous snapshot will need to be redone [wasteful]
- Combine checkpointing (expensive) with message logging (cheap)
 - Take infrequent checkpoints
 - Log all messages between checkpoints to local stable storage
 - To recover: simply replay messages from previous checkpoint
 - Avoids recomputations from previous checkpoint