There Is More Consensus in Egalitarian Parliaments

Iulian Moraru, David G. Andersen, Michael Kaminsky
Carnegie Mellon University and Intel Labs

Presented by : Ojasvi, Srushti, Priyadarshini, Brian
Supervised By : Prof. Abhilash Jindal

Agenda

- ❑ Introduction
- ❑ Multi-Paxos
- ❑ Paxos Background and working
- ❑ Existing solutions
- ❑ EPaxos : Goals & Approach
- ❑ Working of EPaxos, Execution Flow
- ❑ Design : Preliminaries
- ❑ Protocol Guarantees
- ❑ The Basic Protocol

Introduction

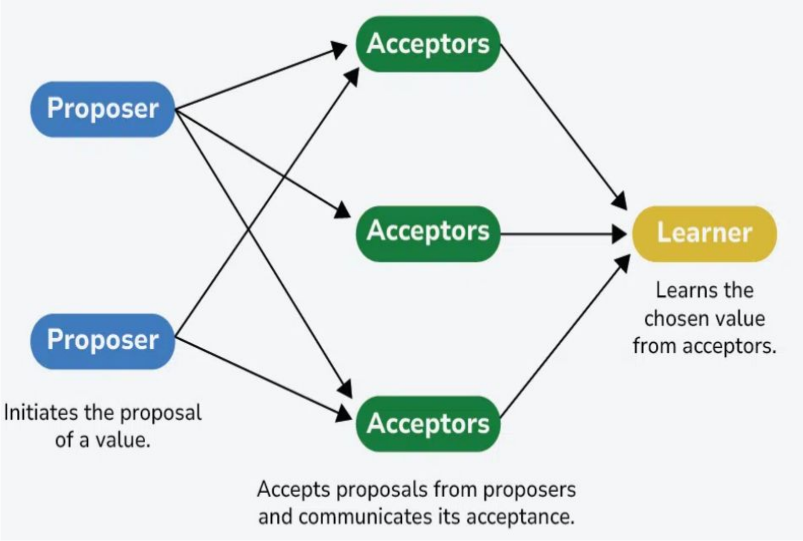
- Demands on Replication Protocols -
  - High throughput for replication inside computing cluster
  - Low latency for geo-replication

- Recent Clusters like Zookeeper, Boxwood, Chubby use fault tolerant coordination engines for activities like operation sequencing, coordination, leader election, resource discovery.
- Limitation: All clients communicate with single master/leader in efficient, fault-free operation

Paxos Background

Paxos is a classic consensus algorithm which ensures that a distributed system can agree on a single value or sequence of values, even if some nodes may fail or messages may be delayed.

- **Proposers:** Proposers initiate the consensus process by proposing a value to be agreed upon.
- **Acceptors:** Acceptors receive proposals from proposers and can either accept or reject them based on certain criteria.
- **Learners:** Learners are entities that receive the agreed-upon value or decision once consensus is reached among the acceptors.


Proposer
Proposer
Acceptors
Acceptors
Acceptors
Learner
Initiates the proposal of a value.
Accepts proposals from proposers and communicates its acceptance.
Learns the chosen value from acceptors.

Paxos Working

Choosing a Leader (Voting Process):

- One replica asks most of the group (> half + 1) for permission to lead by sending "Prepare" message
- It learns about any old decisions or leaders from the group and promises to skip outdated instructions
- If most of the group agrees (majority says yes), it becomes the leader for that instance

Making a Decision (Command Process):

- The new leader sends "Accept" message with a plan (command) to most of the group
- The plan is final when most of the group give ACKs (majority agrees)
- This takes at least 2 back-and-forth steps; more if multiple leaders argue

Multi-Paxos

Multi-Paxos is used to achieve high throughput Paxos-based systems by reusing the leader for multiple rounds of consensus. This algorithmic limitation of using one leader many limitations and consequences-

- Scalability bottleneck
- Increased latency in geo-replication
- Sensitivity to network and load spikes
- Reduced availabilty

Existing Solutions

Protocol	Leader Requirement	Message Complexity (per command)	Load Balancing	Conflict Handling	Key Limitations
Multi-Paxos	One stable leader	Leader Θ(N), others O(1)	Poor - leader handles most messages	Centralized at leader	Leader bottleneck; high latency; pause on re-election
Mencius	Rotating leader (per instance)	O(N) total	Moderate - shared leadership	Must hear from all replicas	Dependent on slowest replica; stalls on failure
Fast Paxos	Coordinator still required	Coordinator Θ (N)	Poor - coordinator overloaded	Two-round fast path; conflicts add more	Leader-dependent; heavy messaging
Generalized Paxos	Stable leader	Learners Θ(N)	Poor - leader overloaded	Out-of-order for non-conflicts; ≥ 2 extra rounds for conflicts	Large messages; leader bottleneck; slow conflict resolution

Existing Solutions (Contd)

Protocol	Leader Requirement	Message Complexity	Availability under Failures	Wide-Area Latency	Key Limitations / Strengths
Multicoordinated Paxos	Multiple coordinators + leader	> $\Theta(N)$ (Clients contact quorum)	Moderate	Moderate-High	Higher message cost; still leader-based for conflicts
S-Paxos	Stable leader + helper replicas	$\Theta(N)$	Moderate	High	Shares client I/O, but leader orders ops → same bottleneck
Generic Broadcast (GB, GB+, Optimistic GB)	No leader	$\Theta(N^2)$	Limited (< 1/3 failures)	High	High message cost; slow conflict resolution
EPaxos (Proposed)	No leader	$O(N)$ - smaller quorum	High - majority suffices	Low	Balanced load, fewer rounds, low latency, robust under failures

Egalitarian Paxos - Goals & Approach

Design Goals	How EPaxos achieves these Goals
1. **Optimal commit latency** in wide-area networks	All replicas can act as **proposers (leaders)** simultaneously, avoiding extra round trips to distant sites.
2. Optimal **load balancing** for high throughput	**No single leader** bottleneck.
3. **Minimal communication** overhead	Commands are committed after contacting the **smallest possible quorum** of replicas.
4. Graceful performance degradation under slow/crashed replicas	Flexible quorum selection allows proposers to **avoid slow/unresponsive replicas.**

How EPaxos Works: Dynamic & Decentralized Ordering

- **Dynamic Instance Ordering**
  Commands are ordered **on the fly** as replicas vote, without predefined slots.

- **Ordering Constraints**
  Each replica attaches **dependencies** to commands during voting.

- **Consistent Global Order**
  All non-faulty replicas commit the **same command with the same constraints**,
  enabling each to **derive the same final order independently**.

- **Key Observation** (from Generalized Paxos)
  - Unlike Raft (strict order for all entries), EPaxos allows flexible ordering.
  - Only commands that **interfere** (e.g., modify the same object) need consistent ordering.
  - Non-interfering commands can be ordered **independently**.

EPaxos Execution Flow – Fast Path and Slow Path

- Clients can send commands to **any replica** — this replica becomes the **command leader** for that command.

- **Common Case (Fast Path)**
  - When commands **don’t interfere**, they can be committed after **one round of communication**.
  - Requires a **fast-path quorum** of $F + \lfloor (F+1) / 2 \rfloor$ replicas (including the leader).

- **Interfering Commands (Slow Path)**
  - When commands **conflict**, they gain **dependencies** to preserve correct execution order.
  - A second communication round with a **classic quorum (F + 1 replicas)** ensures all replicas agree on these dependencies.

* F is the number of tolerated failures.

Example Message flow

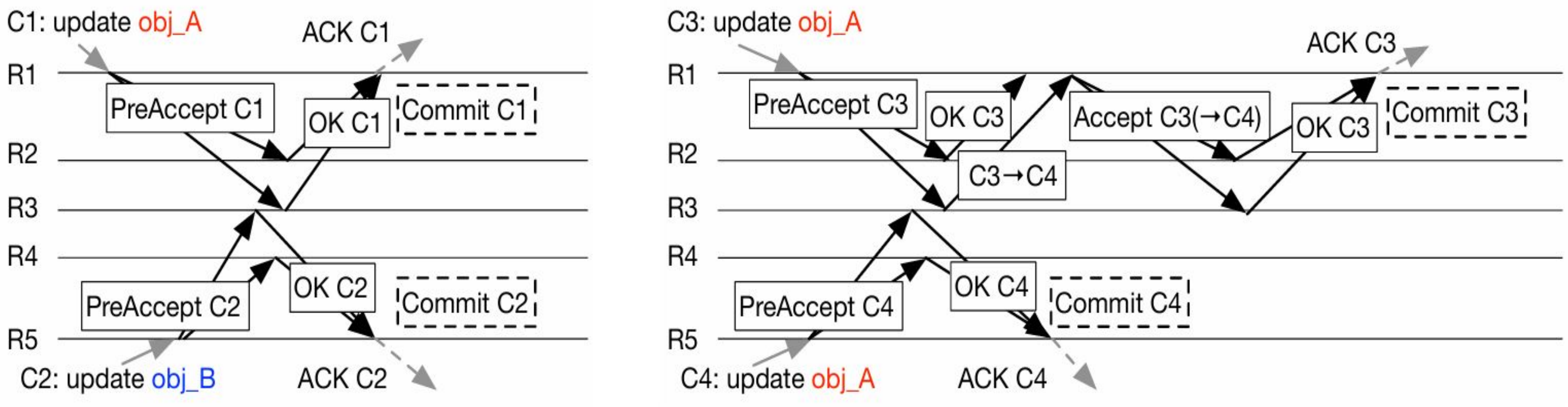


Figure 1: EPaxos message flow. R1, R2, ... R5 are the five replicas. Commands C1 and C2 (left) do not interfere, so both can commit on the fast path. C3 and C4 (right) interfere, so one (C3) will be committed on the slow path. C3 → C4 signifies that C3 acquired a dependency on C4. For clarity, we omit the async commit messages.

Design : EPaxos Preliminaries

System Model

- Asynchronous message passing between clients and replicas.
- Non-Byzantine failures: replicas may stop responding but don’t act maliciously.
- Number of replicas = 2F + 1 → to tolerate up to F failures.

Replica State

- For every replica R → an unbounded sequence of numbered instances R.1, R.2, R.3…
- Each instance holds at most one chosen command.
- System state = all instances from all replicas → think of it as an **N × ∞ grid**.

Dynamic Instance Ordering

- Order of instances is **not predefined**, determined dynamically as commands are committed.

EPaxos Preliminaries – Client Interaction & Interference

Client Operations

- `Request(command)` → sends to any chosen replica .
- `RequestReply` → confirms commit, not execution.
- Execution occurs when the client reads the updated state from committed commands
- `Read(objectIDs)` → reads object state (acts as a no-op that conflicts with updates).
- `RequestAndRead(command, objectIDs)` → allows atomic execution and immediate read.

Command Interference

- Two commands **γ** and **δ** interfere if there exists a sequence of commands **Σ** such that the serial execution **Σ, γ, δ** is not equivalent to **Σ, δ, γ**.

Protocol Guarantees

1. **Nontriviality:** Only client-proposed commands can be committed by replicas.

2. **Stability:** Once a command is committed in an instance, it remains committed at all later times.

3. **Consistency:** No two replicas commit **different commands** for the same instance.

4. **Execution Consistency**: Interfering commands are executed in the same order by all replicas.

5. **Execution Linearizability**: If a command is proposed after another is committed, all replicas execute them in that client-observed order.

6. **Liveness** (w/ high probability): Commands are eventually committed by all non-faulty replicas if:
    a. Fewer than half the replicas fail, and
    b. Messages eventually arrive before timeouts.

Basic EPaxos Commit Protocol

- **Context:** EPaxos is a **leader-less, generalized** consensus protocol.
- **Key Distinction:** It works by building a **replicated conflict graph** where commands have **dependencies** on each other.
- **Replica State:** Each replica maintains a private **cmds log** that records all commands seen, even if they are not yet committed.
- **Separation: Commit** (consensus reached on command and attributes) is separated from **Execute** (applying command to state machine).
- **Basic Protocol Quorum:** Fast-Path Quorum is simplified to 2F (out of N=2F+1 replicas). The Slow-Path Quorum is F+1.

Establishing Ordering Constraints (PreAccept)

Command Leader (L): Any replica receiving a client request becomes the command leader. It chooses the next available instance in its subspace (L.i).

Ordering Attributes: The leader attaches two critical safety attributes:

- **Dependencies (deps):** A list of all uncommitted instances that **interfere** (conflict) with γ. This upholds **Dependency Safety**.
- **Sequence Number (seq):** A logical value set higher than the seq of all conflicting commands. It's used to **break dependency cycles** during execution.

Message: The leader sends (γ,seq,deps,L.i) as a **PreAccept** message to a **Fast-Path Quorum (F) of replicas**.

Fast Path: The Optimal 1-RTT Commit

- **Condition:** Used when **no interference** is detected between γ and any concurrent command. This is the common case.
- **Replica Action:** Replicas check for conflicts, update attributes if necessary, and reply with a **PreAcceptOK**.
- **Leader Commits IF:**
  - The leader receives ≥⌊N/2⌋ replies.
  - **CRITICAL: ALL** replies from the Fast Quorum (F) have the **identical** seq and deps attributes.
- **Result:** The command is committed after **one communication round trip** (1 RTT).

Slow Path: Conflict Resolution in 2 RTTs

- **Condition:** Used when the Fast Path fails (e.g., if attributes are inconsistent, indicating contention/interference).
- **Leader Action (Consolidation):** The leader resolves conflict by combining attributes:
  - deps becomes the **union** of all received lists.
  - seq becomes the **maximum** seq observed.
- **Phase 2 (Paxos-Accept):** The leader initiates the classical Paxos-Accept phase, sending the consolidated triplet $(\gamma,deps,seq)$ to a **majority** ($\lfloor N/2 \rfloor+1$) of replicas.
- **Safety:** This extra round guarantees that **at most one value $(\gamma,deps)$ is chosen for every instance L.i** on the final attributes.
- **Result:** The command is committed after **two communication rounds** (2 RTTs).

EPaxos - Recap

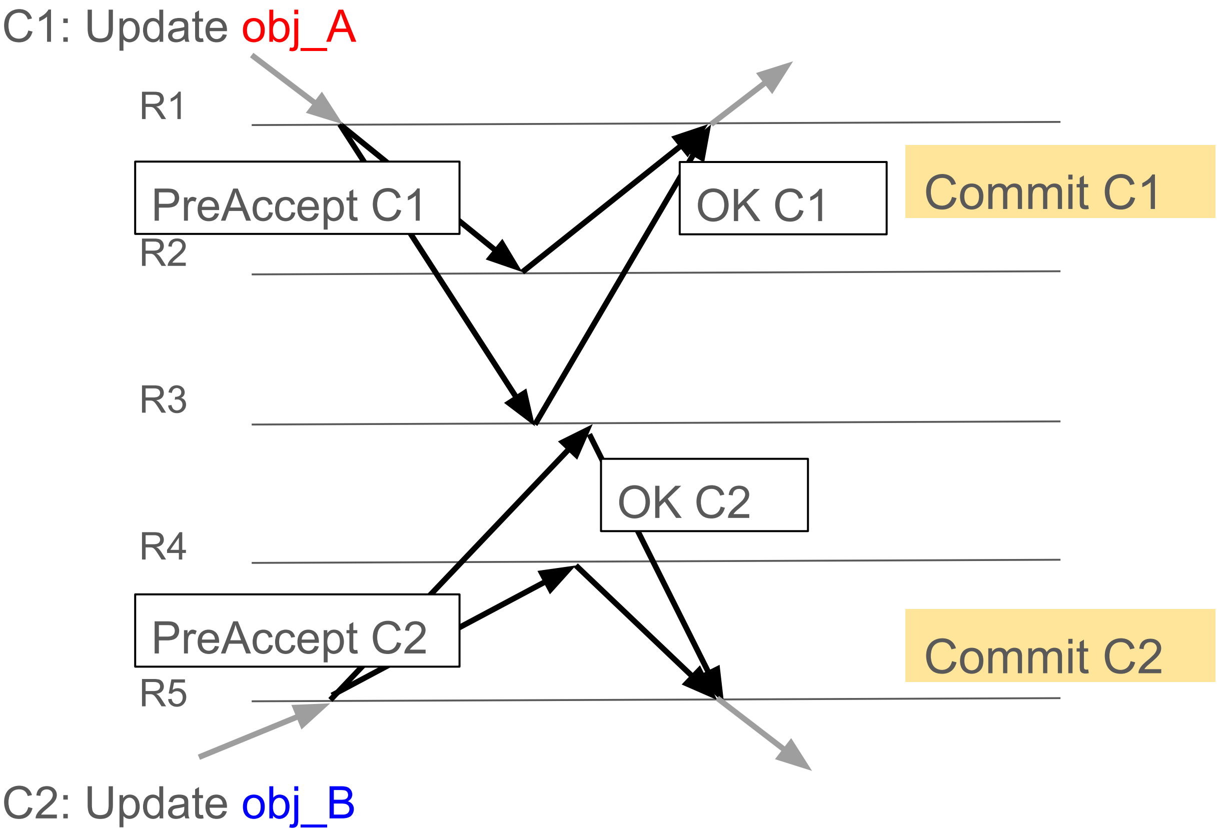
The protocol rely on the following two key invariants for correctness.

- Consensus Safety:
  - Ensures a **single agreed-upon value** for every instance (slot).
  - At most **one command and dependency set** (γ, deps(v)) can be chosen per instance.
  - **Guarantee:** No two replicas ever commit **different commands or attributes** for the same slot..

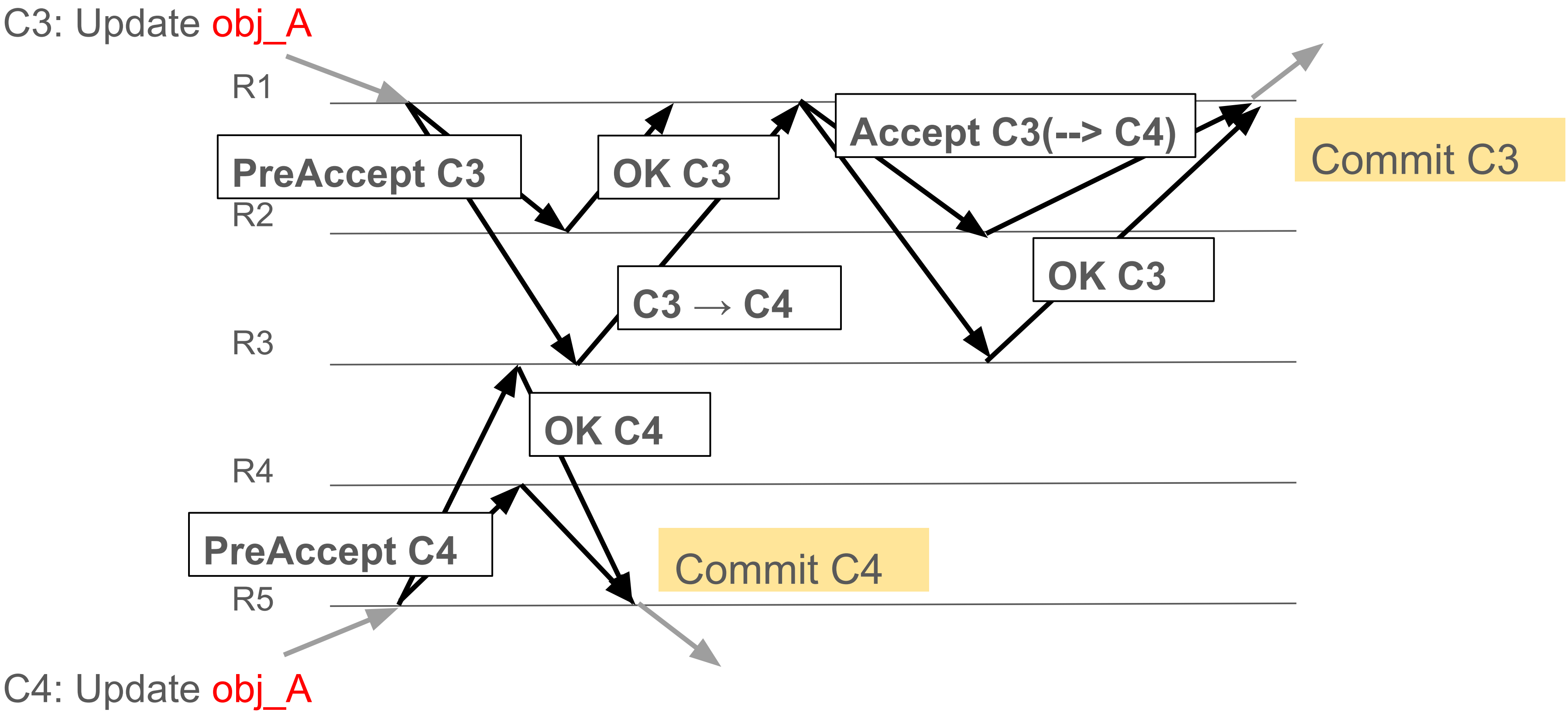
EPaxos - Recap

- Dependency Safety:
    - Ensures **correct ordering** of only those commands that **conflict**.
    - If two commands x and y conflict → at least one must **depend on the other**:
        - x ∈ deps(y) **or** y ∈ deps(x) (or both).
    - Guarantee:
        - Conflicting commands → executed **in the same order** everywhere.
        - Non-conflicting commands → can execute **independently**

Recap: Fast Path



Recap: Slow Path


C3: Update obj_A

C4: Update obj_A

Why does this ensure Dependency Safety?

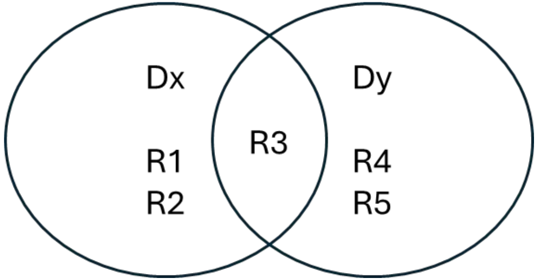
- **Goal:** If two conflicting commands **x** and **y** commit →

  at least one edge exists: x →y or y →x

- **The Mechanism:** Each leader collects **F+1 replies** →

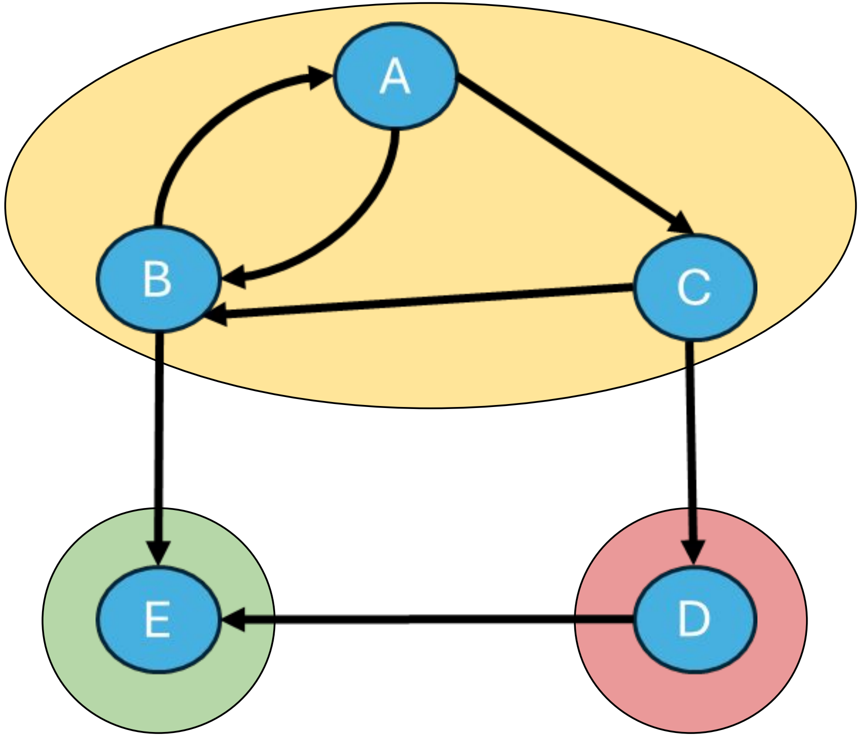
  Takes the **Union of deps** from all replies.

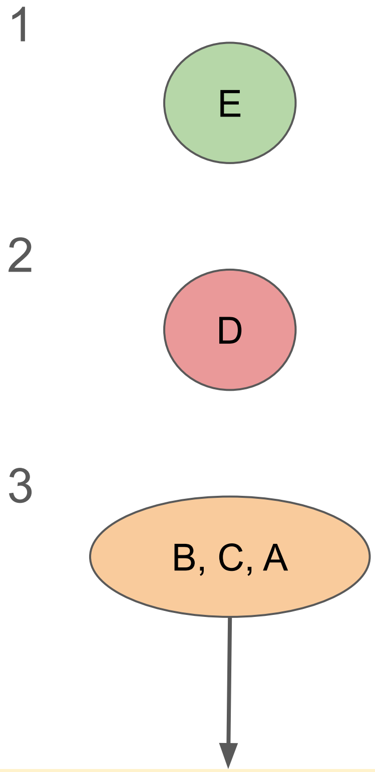
- Suppose **N = 2F+1 = 5**. Then **F+1 is a majority**.
- **R3** sees both x and y:
  - If R3 saw **x first** → **y.dep = {x}**
  - If R3 saw **y first** → **x.dep = {y}**
- Union ensures at least one dependency is recorded.



Execution

Strongly connected component




1

E

2

D

3

B, C, A

Increasing sequence number order (Lamport Clock)

Failures

- Resuming failed/delayed runs
- When executed? If replica Q wants to know status of instance L.i

  Why would Q want to know? To resolve dependency values

Explicit Prepare algorithm

- Increment ballot number and resume EPaxos on behalf of L

  Q will send Prepare to all, wait for atleast F+1 replies (including self)

  Others reply with command data and status (committed, accepted, etc)

Explicit Prepare

- Cases wrt replies R:

  - **Case 1**: R contains a committed msg (Failed during commit)

  Run Commit phase for L.i

  - **Case 2**: R contains atleast 1 accepted msg or atleast F identical pre-accept msgs

  (Failed during Paxos-Accept)

  Run Paxos-Accept phase

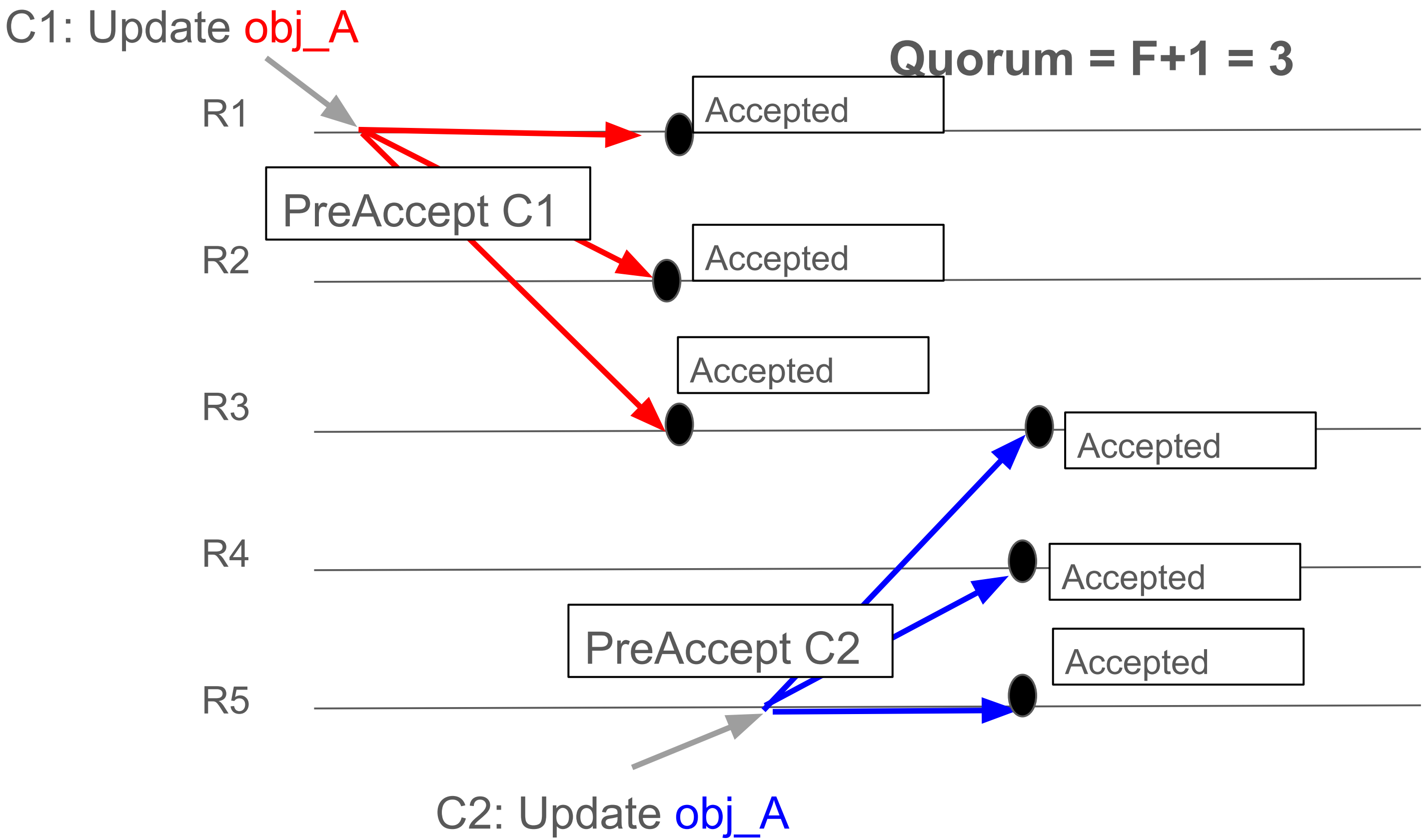
  - **Case 3**: atleast 1 pre-accept

  Run Phase 1

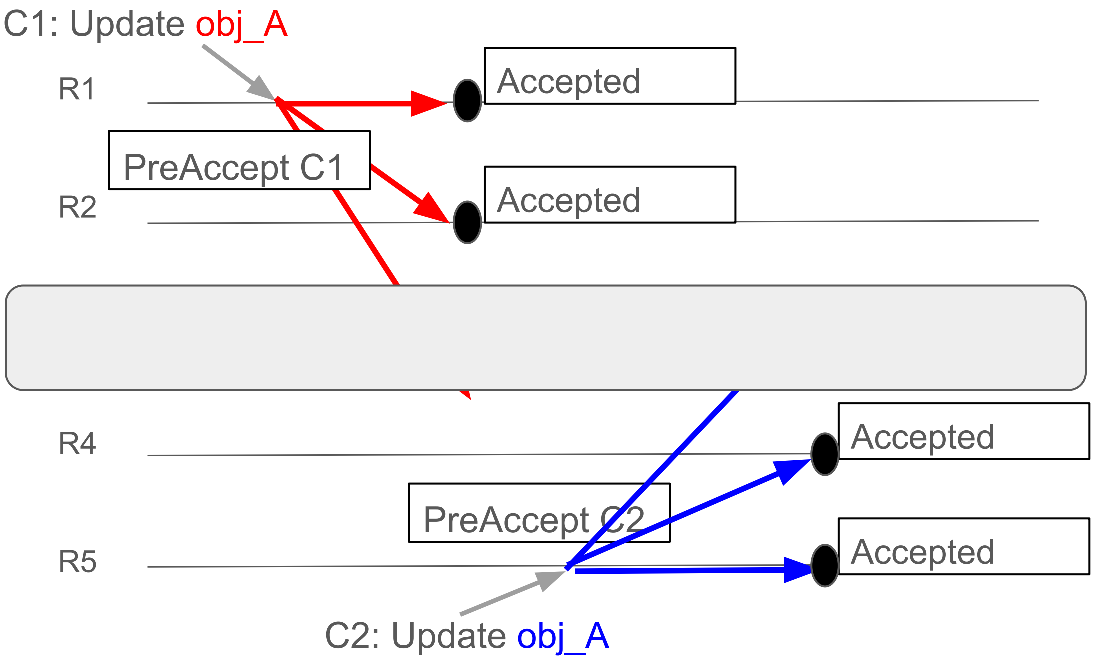
  - **Case 4**: any other cases - not enough info to run EPaxos again

  Run Phase 1 with no-op value

Why Simple Majority Fails in Fast Path?



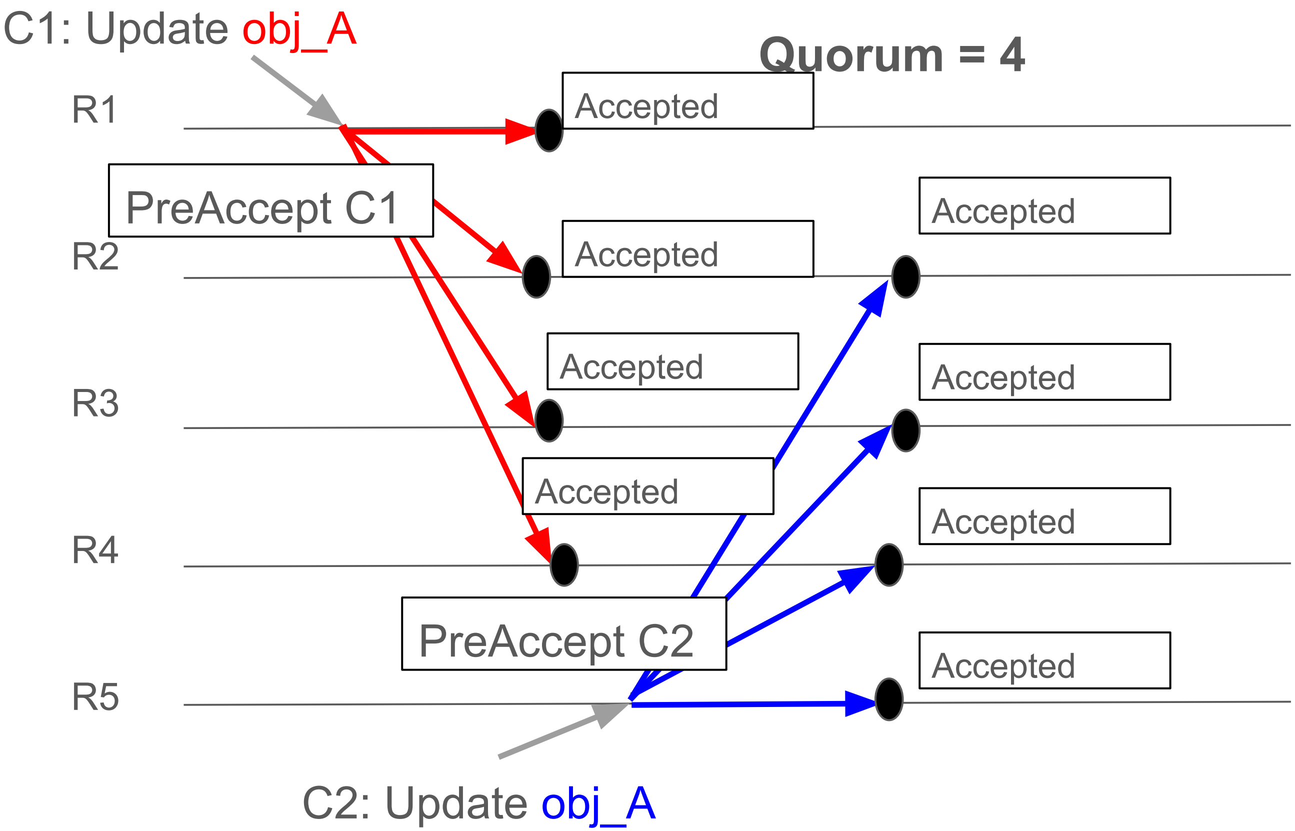
What if R3 is gone forever?

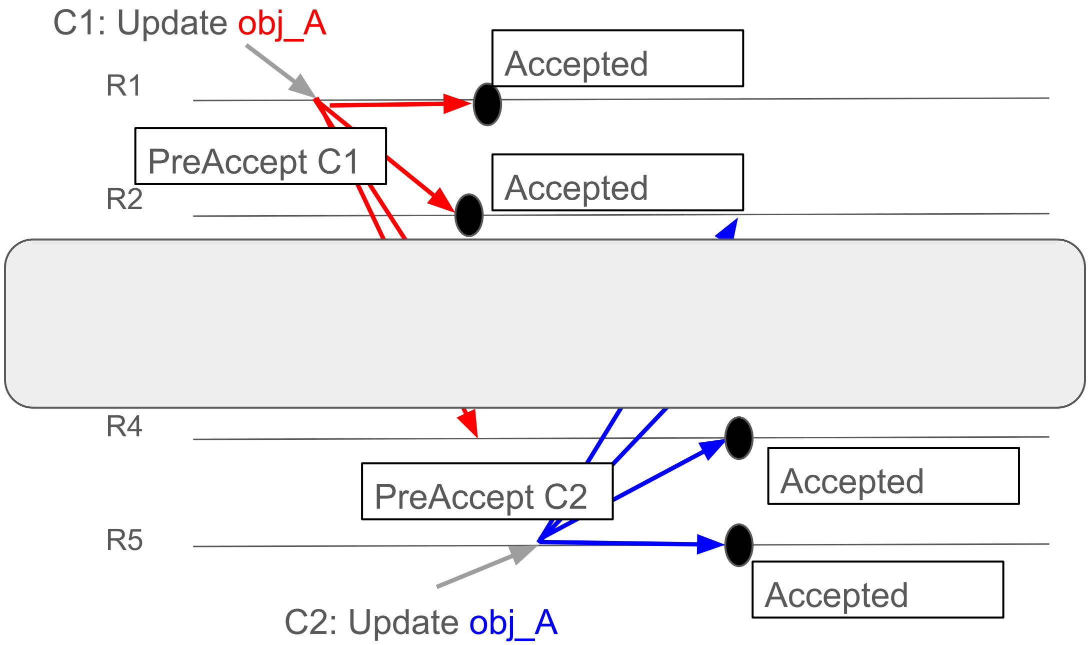


If a simple majority is used, a recovering leader **cannot tell which deps set (if any) won**.

Leads to ambiguity - Violates Consensus Safety

Avoiding Ambiguity with Larger Quorum



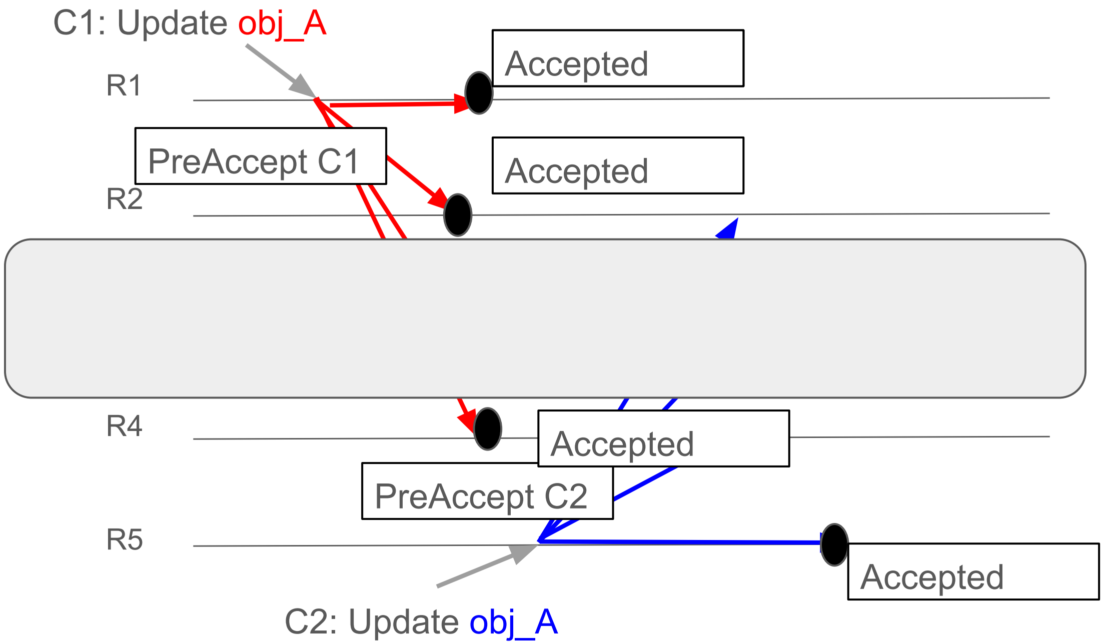


Red proposal received and accepted by **R1 and R2 (2 votes)**

Blue proposal received and accepted by **R4 and R5 (2 votes)**

Neither red nor blue made it

So the system can safely **"Forget both red and blue"** and **"treat as clean slate"**.



Red proposal received and accepted by **R1, R2, R3 (3 votes)**

Blue proposal received and accepted by **R5 (1 vote)**

The leader must be **conservative** and assume **Red** *may have been chosen*.

If a value might have been committed, a new leader **must** propose that value to maintain safety.

Therefore, the leader is forced to **"run from Phase -1 with red"**.

Root Cause

Situation	**Who Proposes?**	**Needed Overlap**	**Why?**
Slow Path (Classic Paxos)	One leader per instance	**>= 1 replica**	Only one proposal active - one overlap replica can remember it
Fast Path (Fast Paxos / EPaxos)	Many replicas may propose concurrently	**>= F + 1 replicas**	Several conflicting proposals - we need enough overlap that at least one **correct** replica survives even if F fail

Deriving the Fast-Path Quorum Size

Fast Quorum - Qf

Classic Quorum - Qc

1. Safety Condition: $|Qf \cap Qc| >= F+1$ (at least one correct overlap after F failures)
2. Using set identity: $|Qf \cap Qc| \geq |Qf| + |Qc| - N$
3. Therefore, $|Qf| + |Qc| > N + F$
4. Substitute $|Qc| = F + 1$ and $N = 2F + 1$ —> $|Qf| > F + (F + 1)/2$
5. Round up to integer —> $|Qf| = F + \lceil (F + 1)/2 \rceil$

Example: N = 5 (F=2)

Fast-Path Quorum = 4 (F + ⌈(F + 1)/2⌉)

C1 quorum: {R1, R2, R3, R4}

C2 quorum: {R2, R3, R4, R5}

Fast path ⇒ many concurrent proposers ⇒ need larger quorum.
Slow path ⇒ one leader at a time ⇒ simple majority is enough.

Overlap = {R2, R3, R4} (3 replicas)

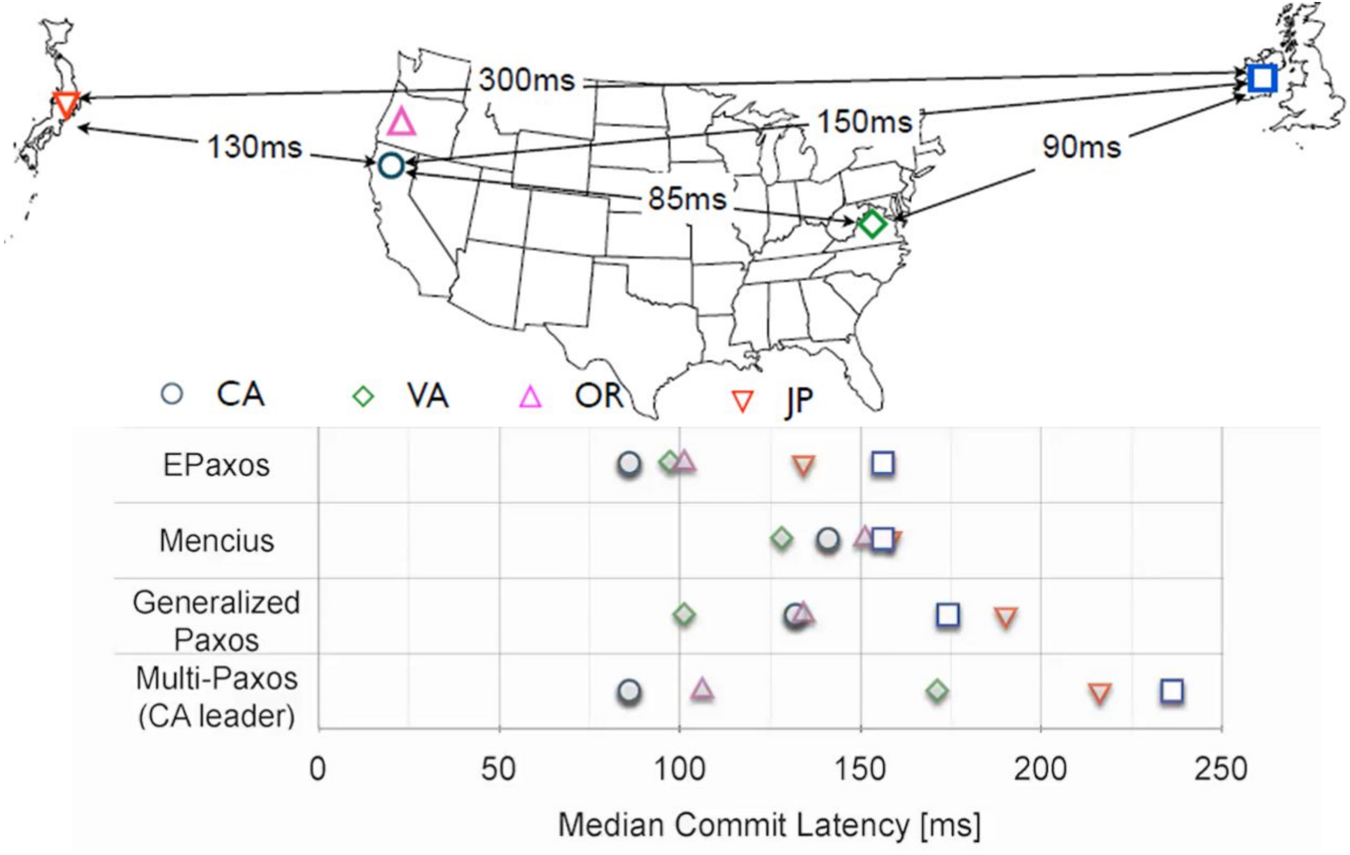
- Even if F = 2 fail, one overlap replica remains alive.
- That replica "remembers" both proposals → can force a consistent dependency set.
- Thus only **one `(cmd, deps, seq)`** can be chosen → **Consensus + Dependency Safety** hold.

Reads

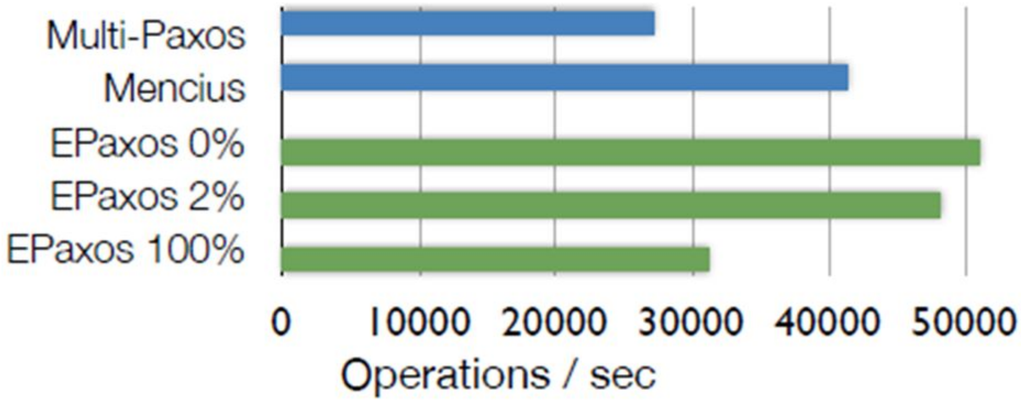
- Normal paxos: Run paxos algo for reads as well
    - Expensive
- For Epaxos:
    - Each replica holds “leases” for reads of different objects
    - With assurance that during writes (infrequent) the lease is always included for that object

Results

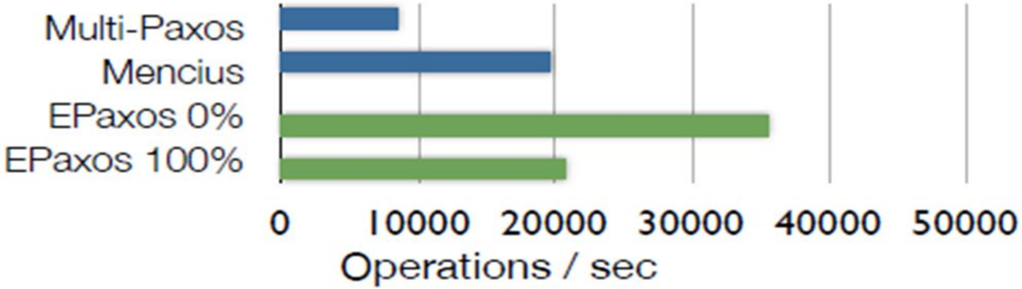
Wide Area Commit Latency



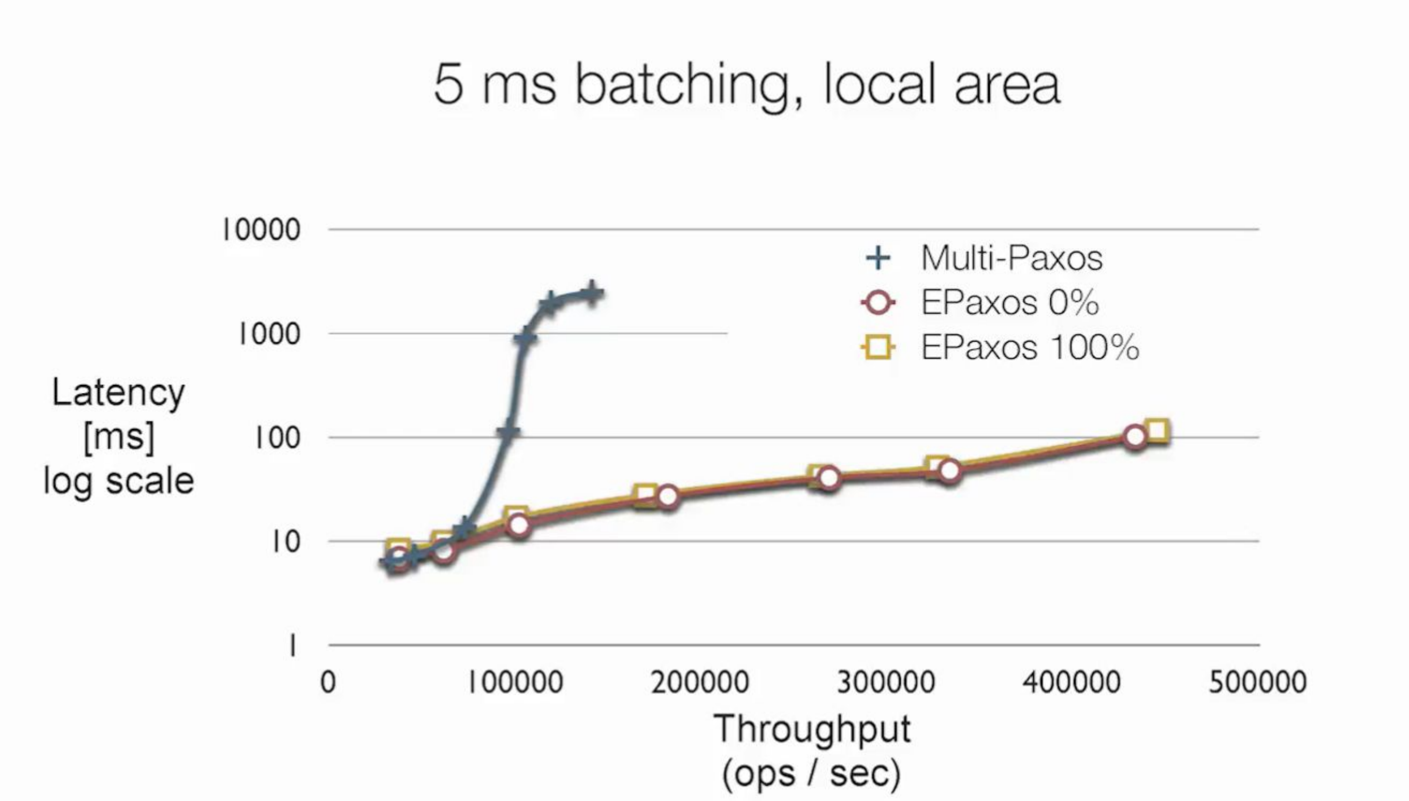
Consistently High Throughput



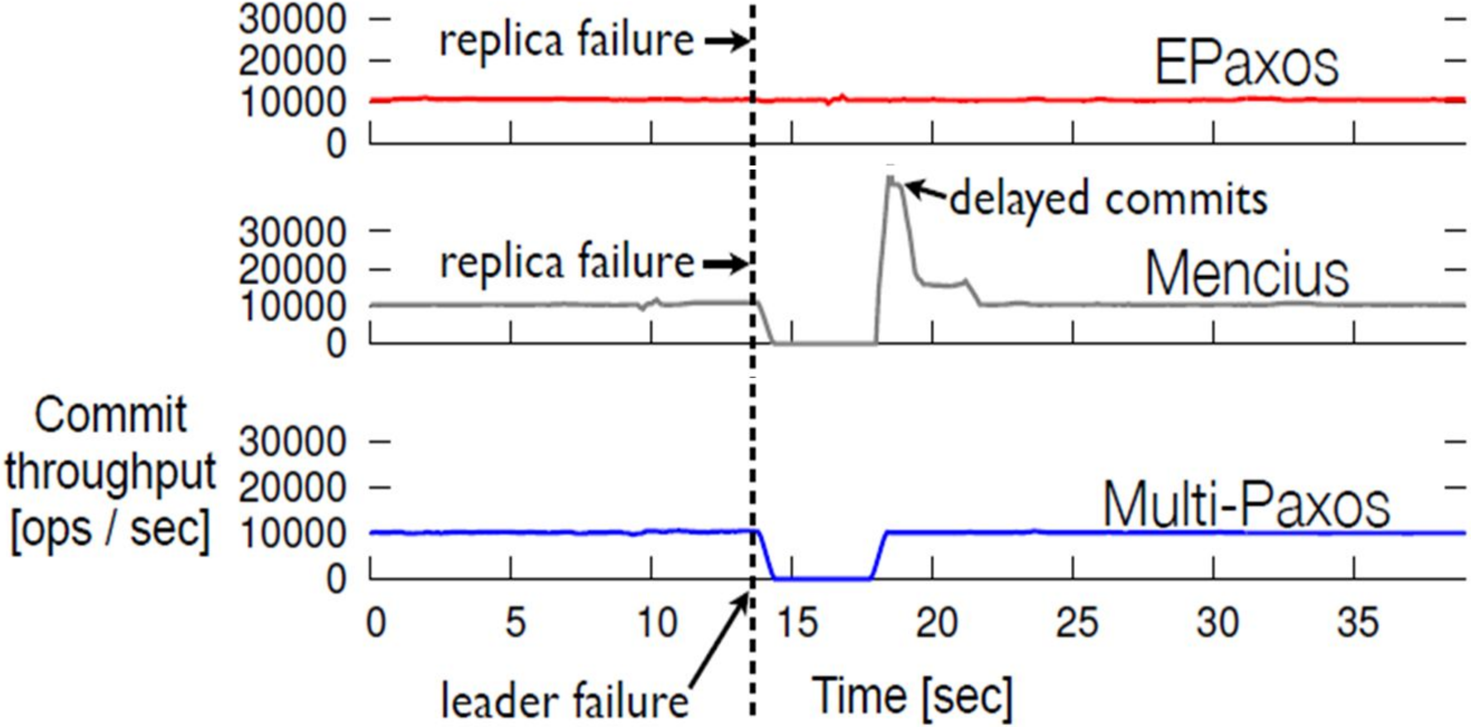
When one replica is slow



Commit Latency



Constant Availability


replica failure
EPaxos
delayed commits
replica failure
Mencius
Commit throughput [ops / sec]
Multi-Paxos
leader failure
Time [sec]