Mencius: Building Efficient Replicated State Machines for WANs

About Mencius

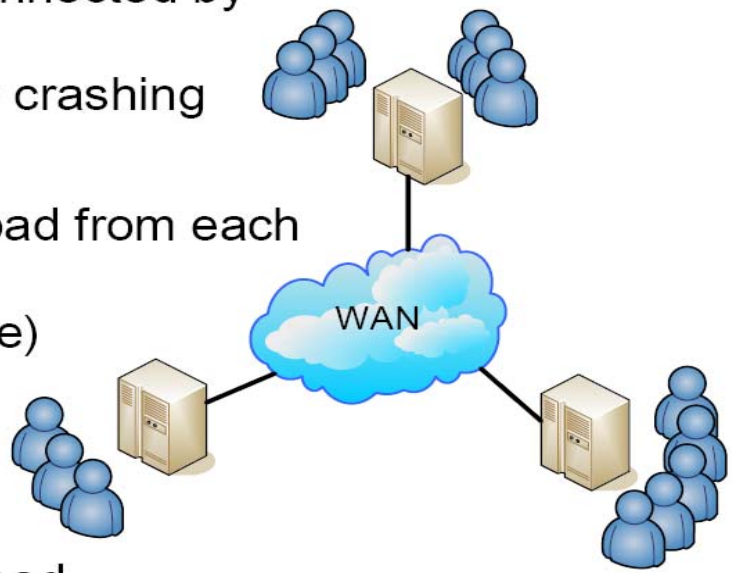
Mencius, or Meng Zi, was one of the principal philosophers during the Warring States Period. During the fourth century BC, Mencius worked on reform among the rulers of the area that is now China.

Building replicated state machines with consensus

- General approach to replicate stateful deterministic services
  - Provide strong consistency
- Servers agree on the same sequence of commands to execute
- Consensus to reach agreement
  - E.g. Paxos, Fast Paxos, Mencius, PBFT, Zyzzyva, ...

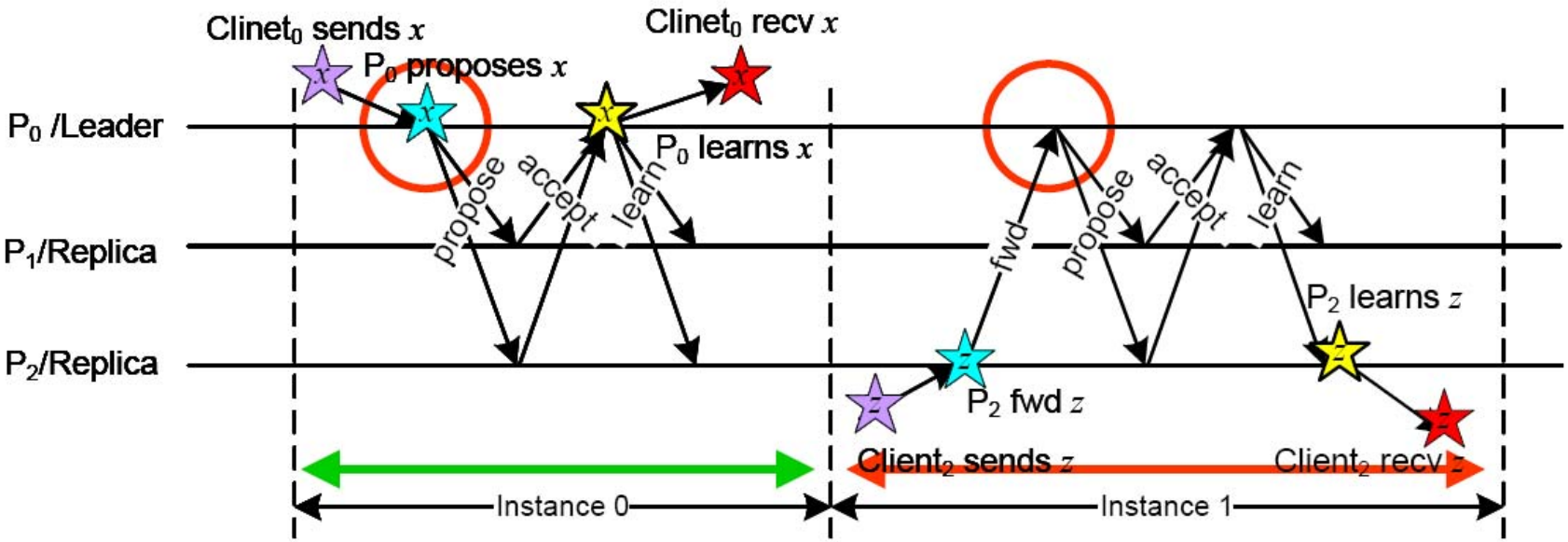
High performance replicated state machines for WANs

- Model
  - A small number of n sites inter-connected by asynchronous wide-area network
  - Up to f out of n servers can fail by crashing
- System load
  - Possible variable and changing load from each site
  - Network-bound (large request size)
  - CPU-bound (small request size)
- Goals
  - Low latency under low client load
  - High throughput with high client load



Paxos in steady state

- Pros
  - Simple and low message complexity (3*n*-3)
  - Low latency at the leader (2 steps)
- Cons
  - High latency at the non-leader replicas (4steps)
  - Not load balanced: bottleneck at the leader or the leader's links



Deriving Mencius from Paxos

- Approach
  - Rotating leader
  - A variant of consensus: simple consensus
  - Optimizations
- Advantages
  - Ensure safety
    - Safety is easy when derived from existing protocol
  - Flexible design
    - Others may derive their own protocol

First step: Rotating the leader

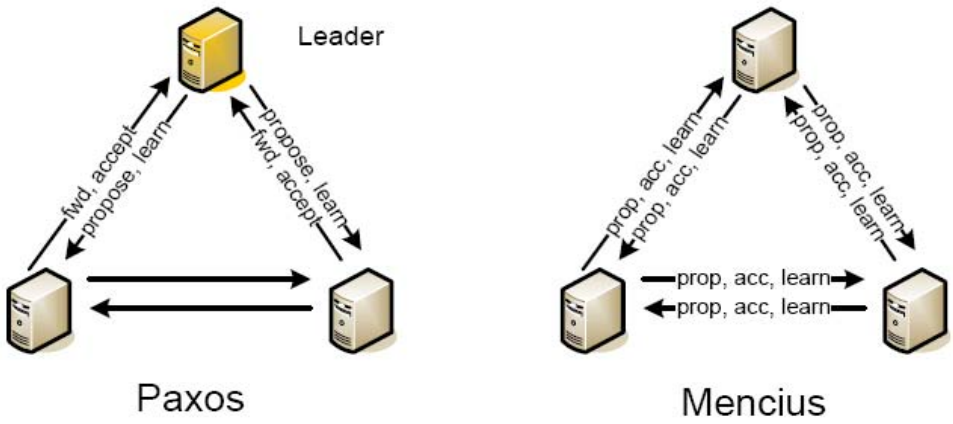
- Each instance of consensus is assigned to a coordinator server
  - The coordinator is the default leader of that instance
  - Simple well-known assignment: e.g. round-robin
- A server proposes client requests immediately to the next available instance it coordinates
  - Don’t have to wait for other servers: reduce latency
- A server only proposes client requests to instances it coordinates
  - There will be no contention

Mencius Rule 1

- A server p maintains its *index* I_p, *i.e.*, the next consensus instance it coordinates.
- **Rule 1**: Upon receiving a client request v, it immediately proposes v to instance I_p and updates its index accordingly.

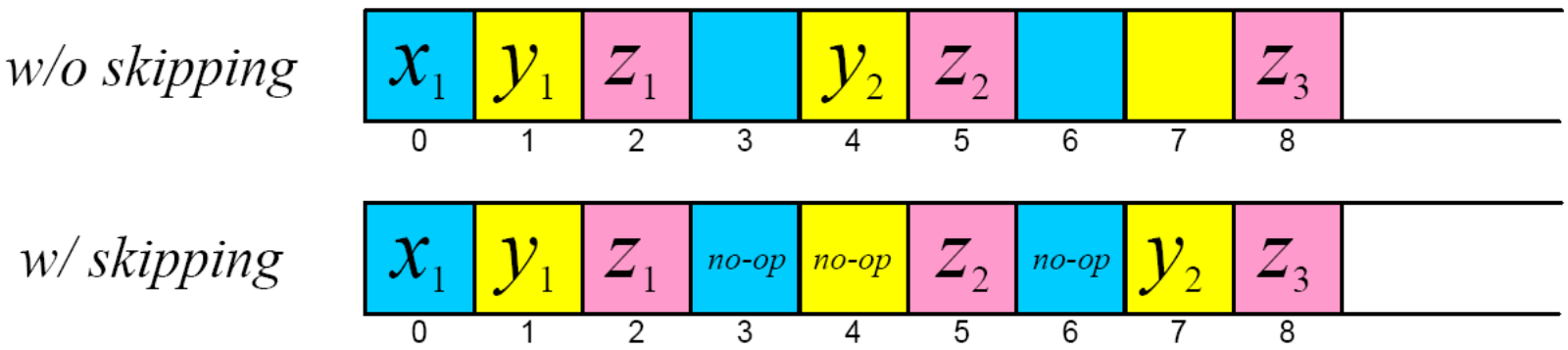
Benefits of rotating the leader

- All servers can now propose requests directly
  - Potentially low latency at all sites
- Load balancing at the servers
  - Higher throughput under CPU-bound client load
- Balanced communication pattern
  - Higher throughput under network-bound load



Ensure liveness when servers have different load

- Rule 1 only works well when all the servers have the same load
- Servers may observe different client loads
  - Servers can *skip* their turns (propose *no-op*s)



Mencius Rule 2

- **Rule 2**: If server p receives a *propose* message with some value v other than *no-op* for instance i and $i > I_p$, before accepting the v and sending back an *accept* message, p updates its index I_p to be greater than i and proposes *no-op*s for each instance in range $[I_p, I_p')$ that p coordinates, where I_p' is p's new index.

Proposing *no-ops* is costly

- Consider the case where only one server proposes requests
  - It takes $3n(n\text{-}1)$ messages to get one value v chosen
    - $3(n\text{-}1)$ for choosing v
    - $3(n\text{-}1)^2$ for $n\text{-}1$ *no-ops*
- Solution
  - Use a simpler abstraction than consensus

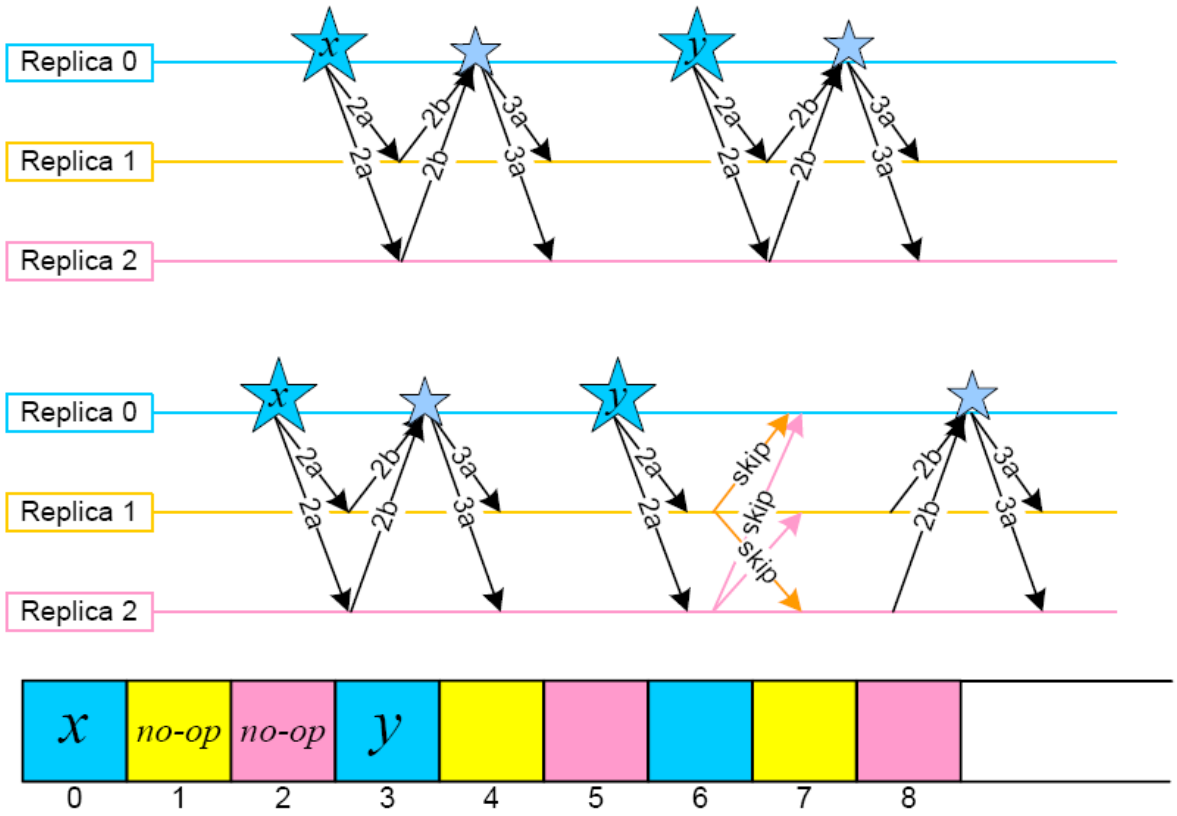
Simple consensus for efficiency

- *Simple consensus*
  - Coordinator: can propose either a client request or a *no-op*
  - Non-coordinator: only *no-op*
- Benefits
  - *no-op* can be learned in one message delay if the coordinator skips (proposes a *no-op*)
  - Easy to piggyback *no-op*s to improve efficiency: essentially no extra cost

Coordinated Paxos

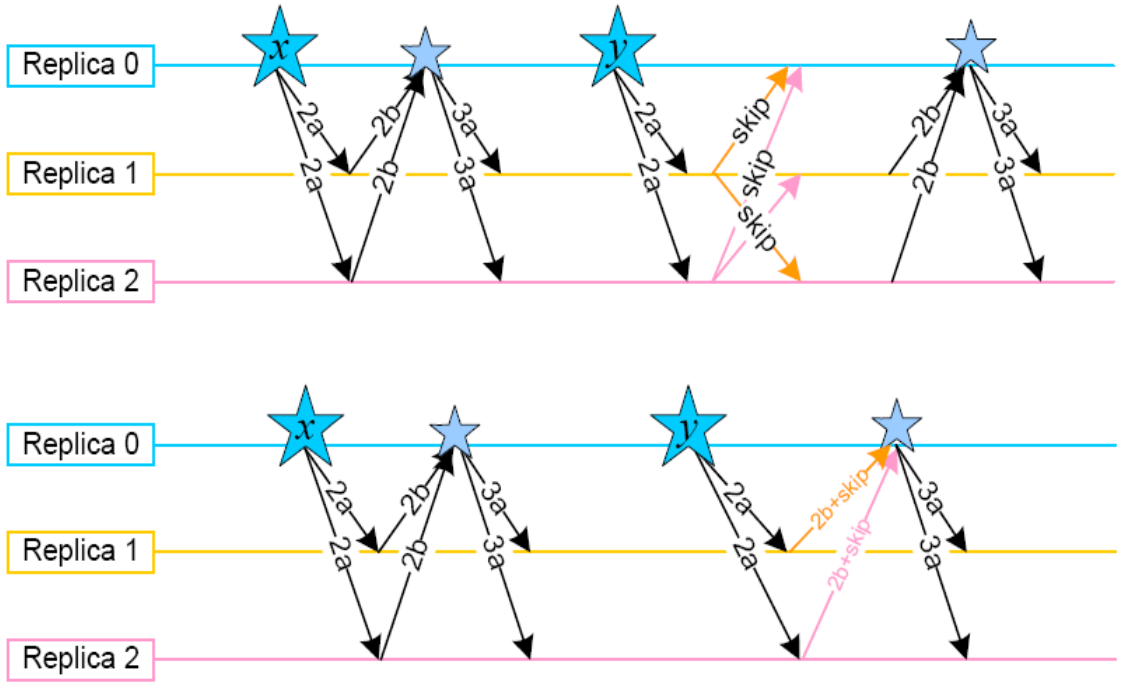
- Starting state
  - The coordinator is the default leader
  - Start from state as if phase one is done by the coordinator
- Suggest
  - The coordinator propose a request by running phase 2
- Skip
  - The coordinator proposes a *no-op*
  - Fast learning for skipping
- Revoke
  - A replica start the full phase 1 and 2
  - Only needed when the coordinator is detected as having failed

Skips with simple consensus

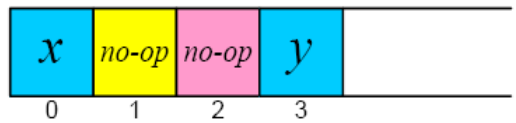


- Pros: Simple and correct protocol
- Cons: High Message complexity: $(n\text{-}1)(n\text{+}2)$

Reduce message complexity



Replica 0's view:



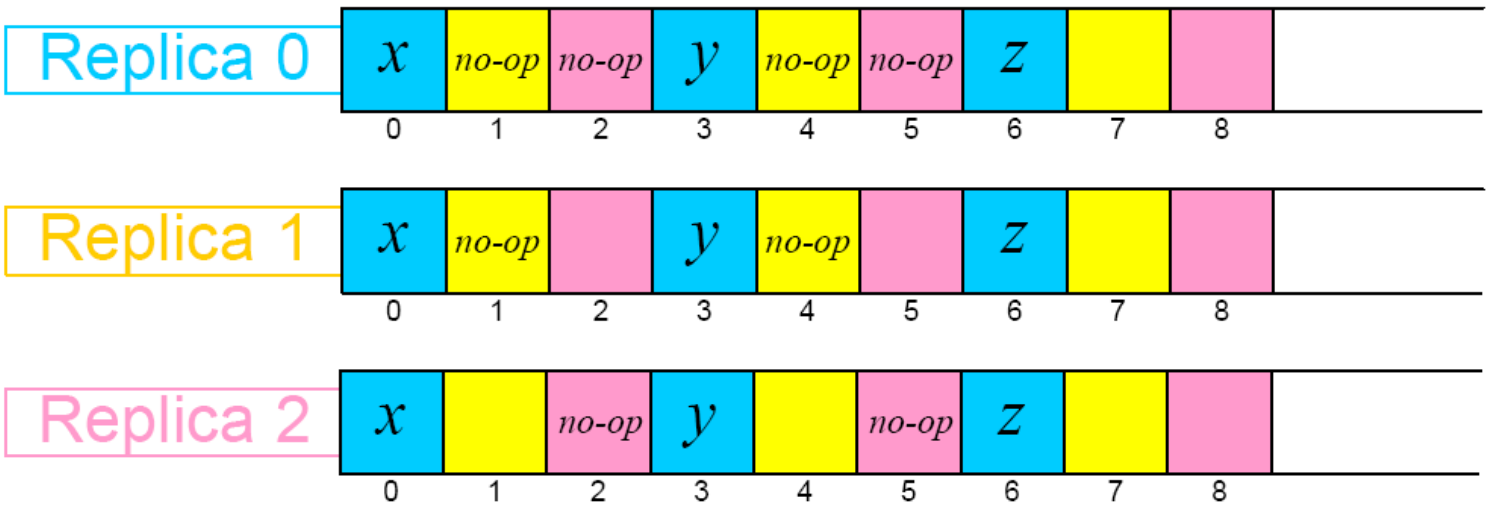
- Idea: piggyback skip to 2b messages
- Can piggyback to future 2a messages too
- No longer need to broadcast

Mencius Optimization 1 & 2

- When a server p receives a suggest message from server q. Let r be a server other than p or q.
- **Optimization 1**: p does not send a separate *skip* message to q. Instead, p uses the *accept* message that replies the *suggest* to promise not to suggest any client request to instance smaller than i in the future.
- **Optimization 2**: p does not send a *skip* message to r immediately. Instead, p waits for a future *suggest* message from p to r to indicate that p has promised not to suggest any client requests to instance smaller than i.

Gap in idle replicas

- Potentially unbounded number of requests wait to be committed
  - Only happens to idle replicas. Once the replica sends requests, the pending skips are resolved.
  - Idle replicas can send skips periodically to accelerate resolution

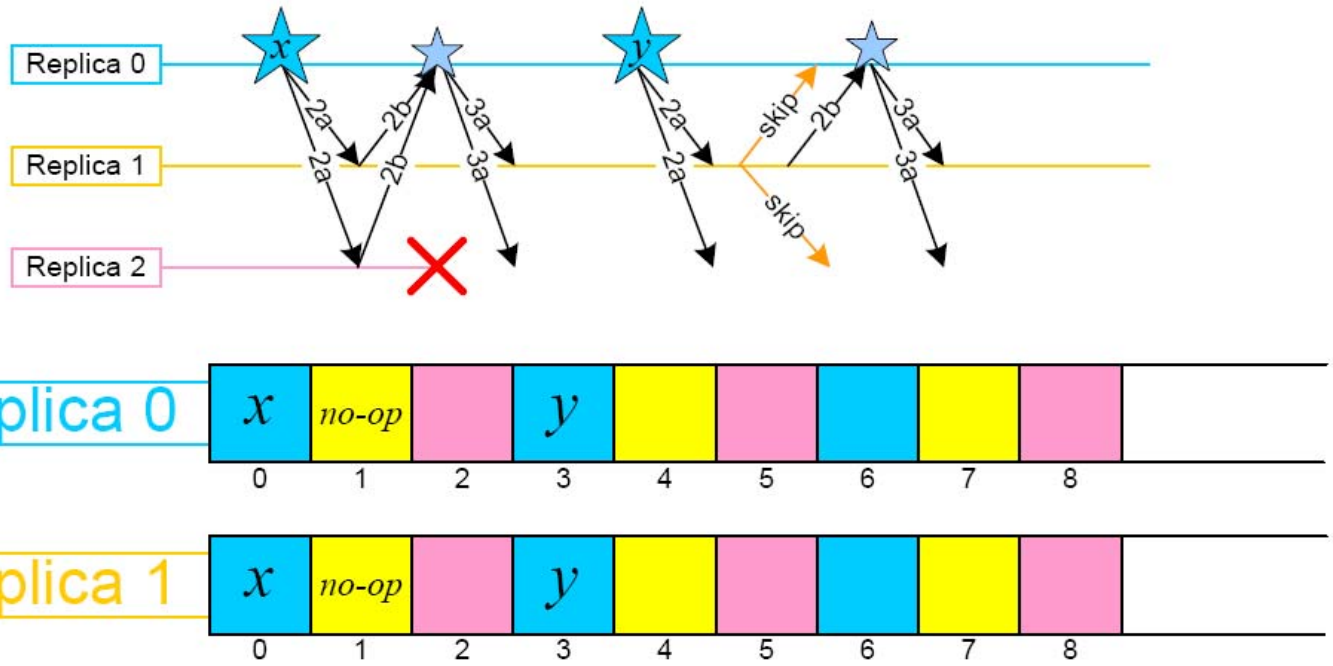


Mencius Accelerator 1

- When a server p receives a suggest message from server q, let r be a server other than p or q.
- **Accelerator 1**: A server p propagates *skip* messages to r if the total number of outstanding skip messages to r is more than some constant α, or the message has been deferred for more than some time τ.

Failures

- Faulty processes cannot skip
  - Leave gap in the command sequence



Mencius: Rule 3

- **Rule 3**: Let q be a server that another server p suspects has failed, and let C_q be the smallest instance that is coordinated by q and not learned by p, p revokes q for all instances in the range $[C_q, I_p]$ that q coordinates.

Deal with failure

- Revoke: propose *no-op* on behalf of the faulty processes
  - Problem: Full 3 phases of Paxos are costly
  - Solution: revoke for larger blocks

Mencius: Optimization 3

- **Optimization 3**: Let q be a server that another server p suspects to have failed, and let C_q be the smallest instance that is coordinated by q and not learned by p. For some constant β, p revokes q for all instances in the range $[C_q, I_p+2\beta]$ that q coordinates.

Imperfect failure detector

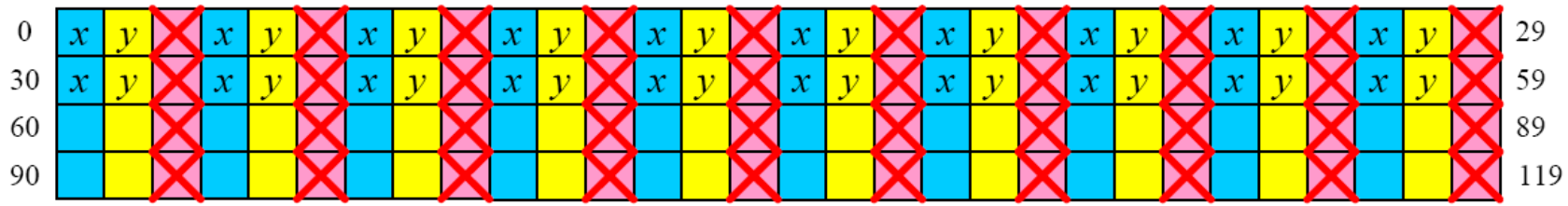
- A false suspicion may lead to *no-op* chosen, even if the coordinator is not crashed and has proposed some value *v*
- Solution: re-suggest *v* when the coordinator learns *no-op*
  - What if false suspicion happens again?
    - Keep trying

Mencius: Rule 4

- **Rule 4**: If server p suggests a value v other than $no\text{-}op$ to instance i, and p learns $no\text{-}op$ is chosen, then p suggests v again.

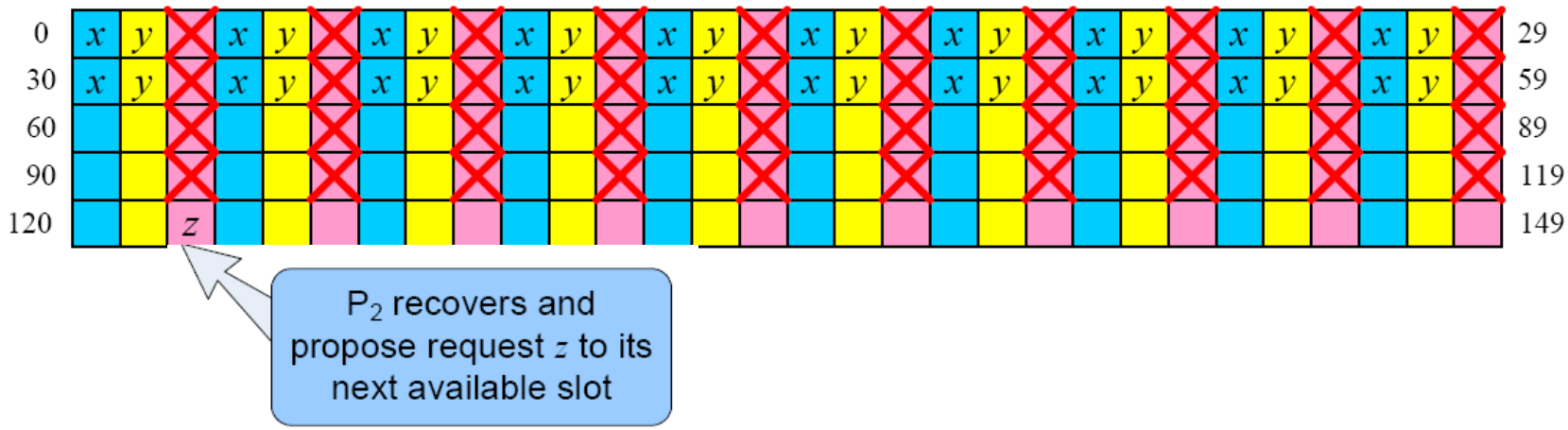
Deal with failure

- Revoke: propose *no-op* on behalf of the faulty processes
  - Problem: Full 3 phases of Paxos are costly
  - Solution: revoke for large block



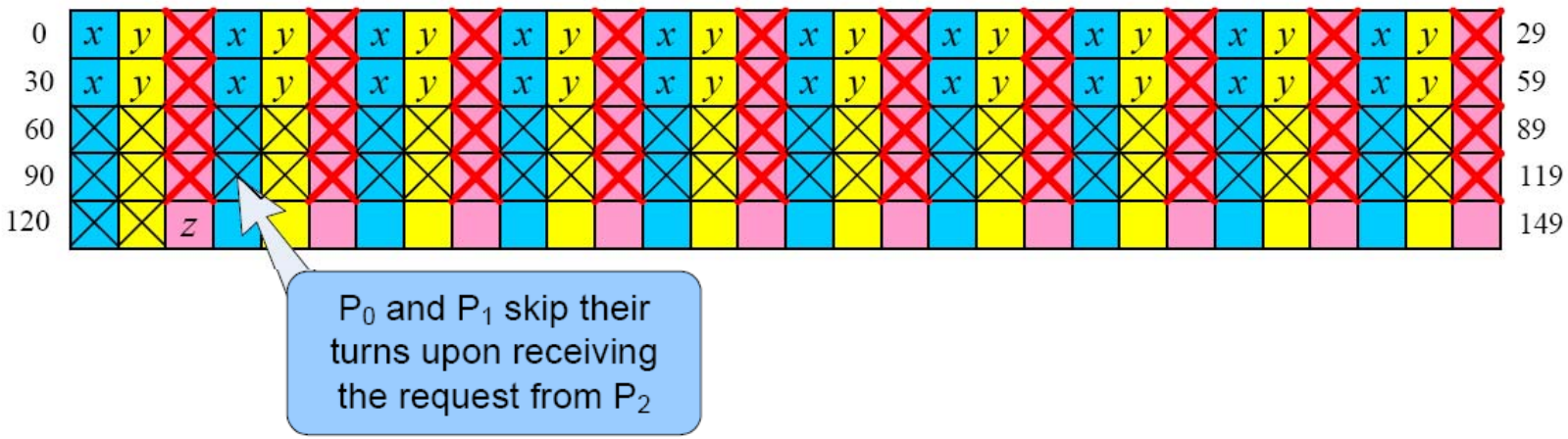
Mencius failure recovery

- P_2 may come back because of failure recovery or false suspicion
  - Find out the next available slots it coordinates
  - Start proposing request to that slot



Mencius failure recovery

- P_2 may come back because of failure recovery or false suspicion
  - Find out the next available slots it coordinates
  - Start proposing request to that slot
  - Other servers catch up with P_2 by skipping



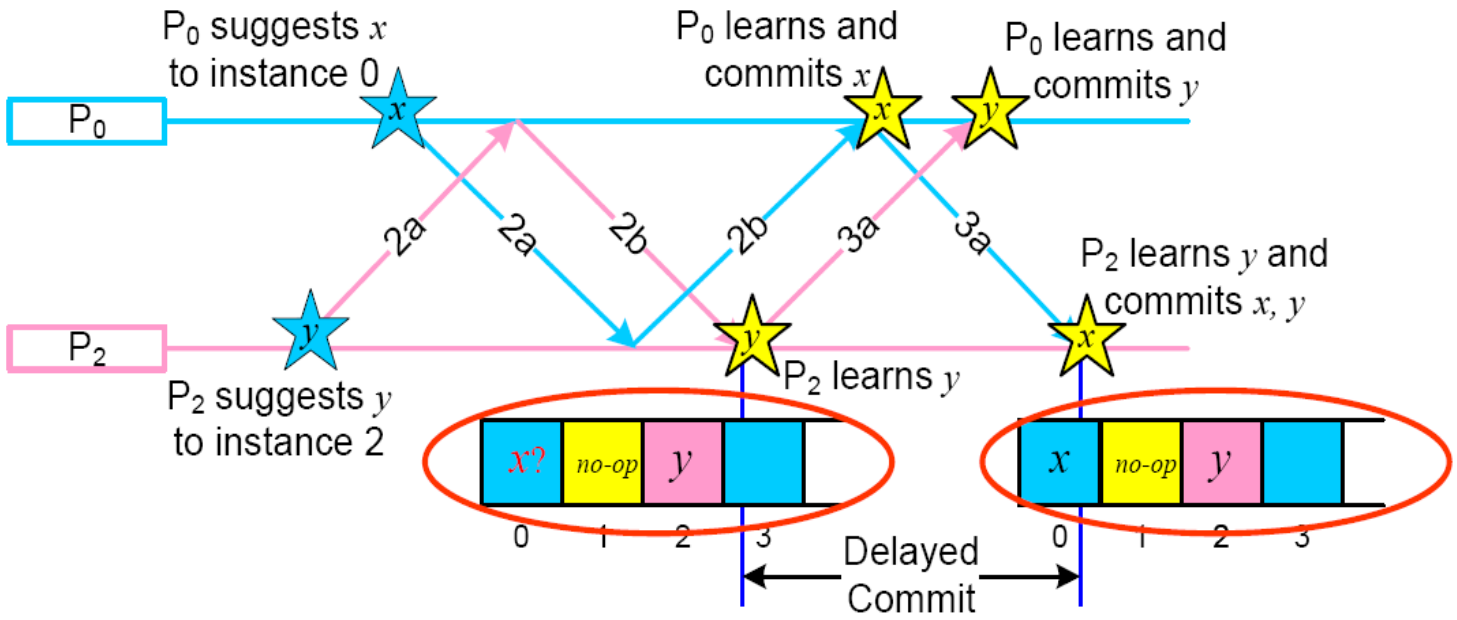
Mencius summary

- Rule 1
  - Suggest client request immediately to the next consensus instance a server coordinates
- Rule 2
  - Update index and skip unused instances when accepting a value
- Rule 3
  - Revoke suspected servers
- Rule 4
  - Re-suggest a client request after a false suspicion

Mencius summary continued

- Optimization 1
  - Piggyback skips to accept (2b) messages
- Optimization 2
  - Piggyback skips to future suggest (2a) messages
- Accelerator 1
  - Put a bound before starting flush skip messages
- Optimization 3
  - Issue revocation in large blocks to amortize the cost

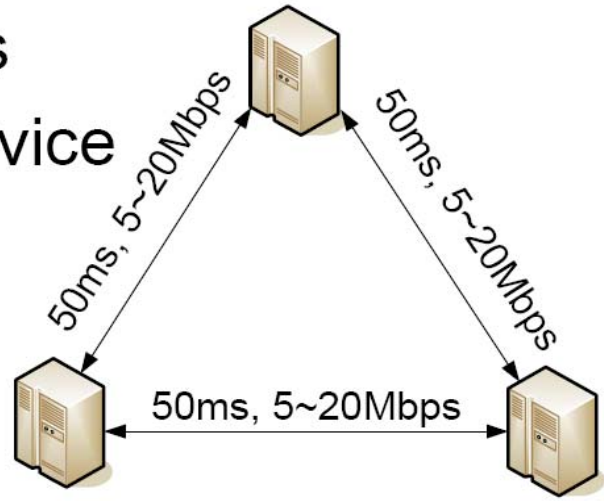
Delayed commit



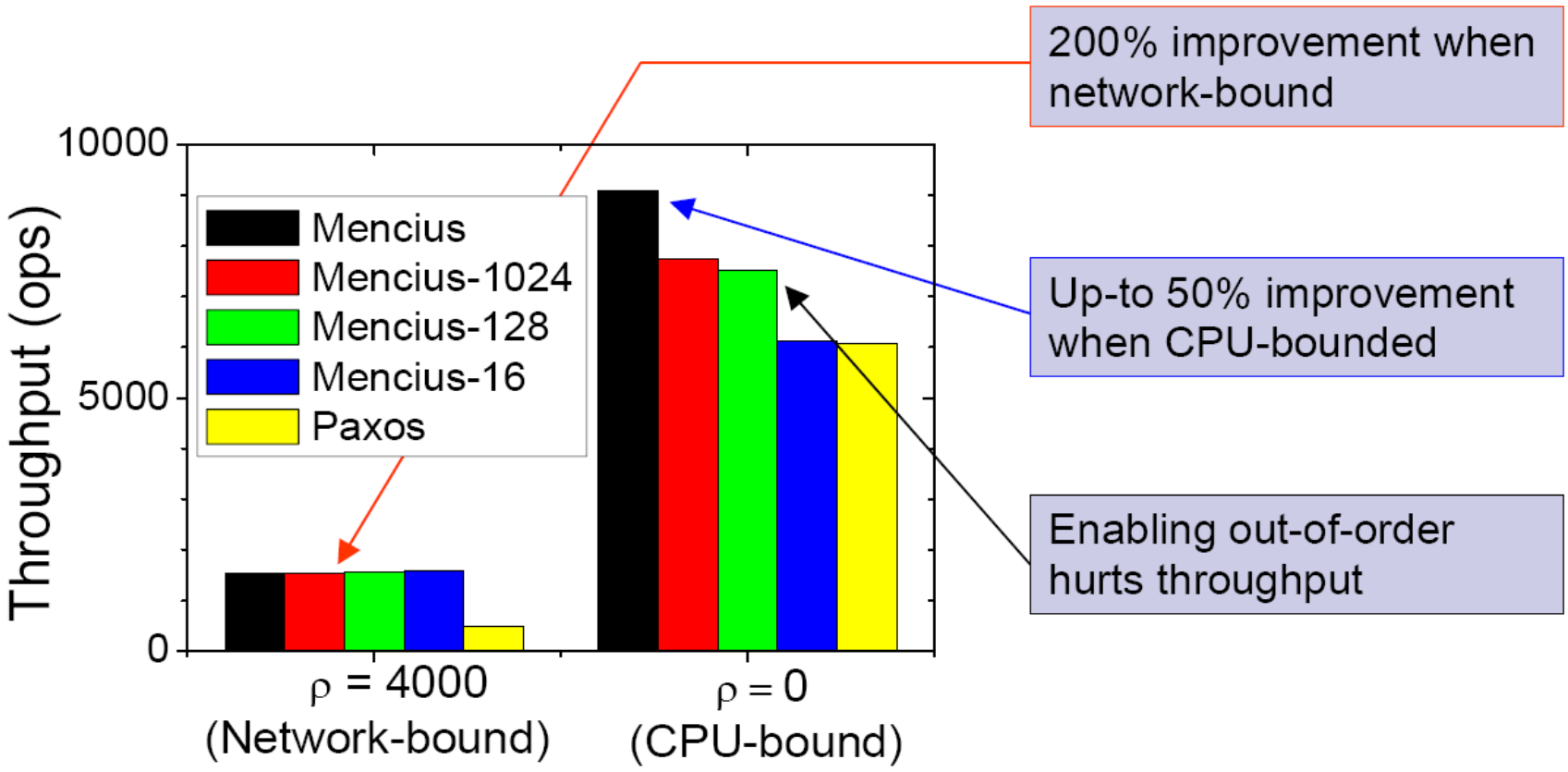
- Up to one round-trip extra delay
- *Out-of-order commit* reduces the extra delay when requests are commutable
  - A benefit of using simple consensus

Experimental setup

- Mencius vs. Paxos
- A clique topology for three sites
- A simple replicated register service
  - k total registers: Mencius-k
  - ρ bytes dummy payload
    - ρ = 4000 : Network-bound
    - ρ = 0 : CPU-bound
- Out-of-order commit option



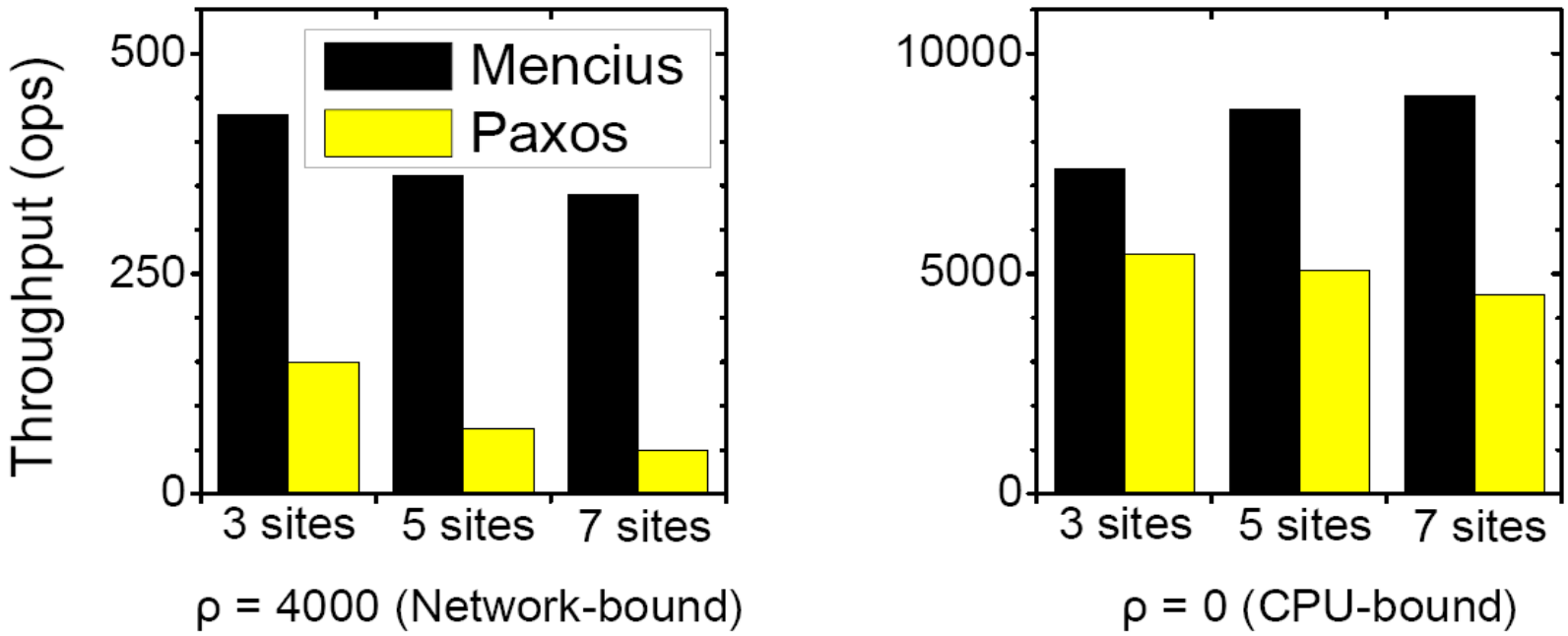
Throughput



Take away: Mencius has good throughput

Fault scalability

Star topology: 10 Mbps link from each site to the central node



Take away: Mencius scales better than Paxos

More evaluation results

- Lower latency at all servers under low contention
  - True, even without out-of-order commit
- Adaptable to available bandwidth
  - Adaptation happens automatically
  - Easy to take advantage of all available bandwidth

Summary

- Mencius
  - Rotating leader
    - Load balance, no single bottleneck
  - Simple consensus
    - Skipping with no overhead
- High performance
  - High throughput under high load
  - Low latency under low load
  - Low latency under medium to high load when enabling out-of-order commit
  - Better load balancing
  - Better scalability

Questions?