# Optimizing Paxos

Arvind Krishnamurthy
*University of Washington*

# *Leader-Based Paxos*

- Optimized to eliminate Phase 1 messages
  - Phase 1 message cost is amortized for many instances into the future
- What are the costs of leader-based Paxos?
  - Number of messages per instance
  - Latency for a client request
  - Number of messages at bottleneck node

# *Paxos deployment models*

- Datacenter
- Wide-area (across the Internet)
- What are the implications of different types of deployments?

# *Paxos Variants*

- Optimizations:
  - Reduce latency from a client perspective (FastPaxos, SpecPaxos, NoPaxos)
  - Reduce load on the leader (FastPaxos, Mencius, EPaxos)

# *Mencius*

- Approach:
  - Rotating leader
  - Variant of consensus algorithm
  - Various optimizations to make rotations seamless

# *Rotating the leader*

- Each instance of consensus is assigned to a "coordinator"
  - Coordinator is the default leader of that instance
  - Simple assignment: e.g., round-robin
- A server proposes client requests immediately to the next available instance it coordinates
- A server only proposes client requests to instances it coordinates

# *Rule 1*

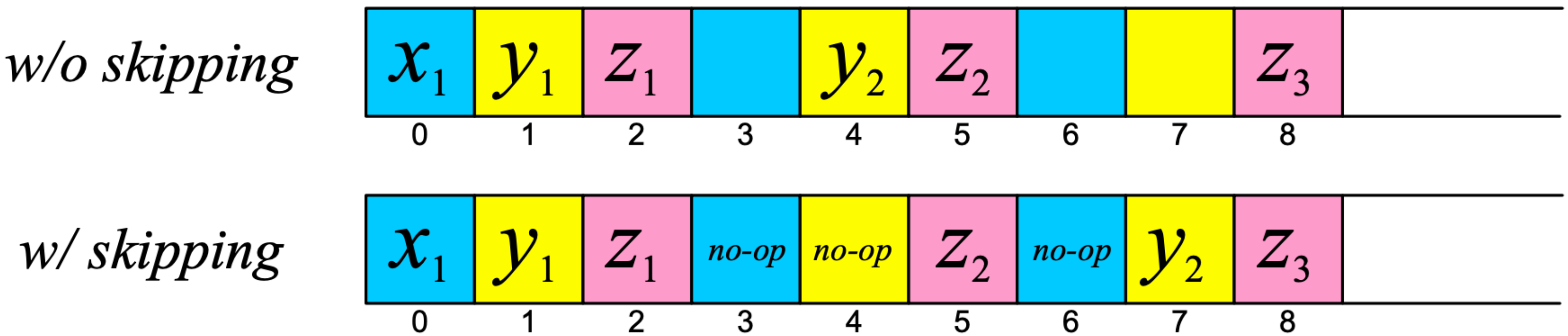
- A server *p* maintains its index $I_p$, i.e., the next consensus instance it coordinates
- Rule 1: Upon receiving a client request *v*, it immediately proposes v to instance $I_p$ and updates its index accordingly

# *Benefits of rotating the leader*

- All servers can now propose requests directly
  - 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

# *Servers with different loads*

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



# *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 *v* and sending back an accept message, *p* updates its index $I_p$ to be greater than *i* and proposes no-ops 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 is proposing values
  - It takes $O(n^2)$ messages to get one value chosen
- Solution: impose constraints on what servers can propose and use these constraints to optimize communication costs

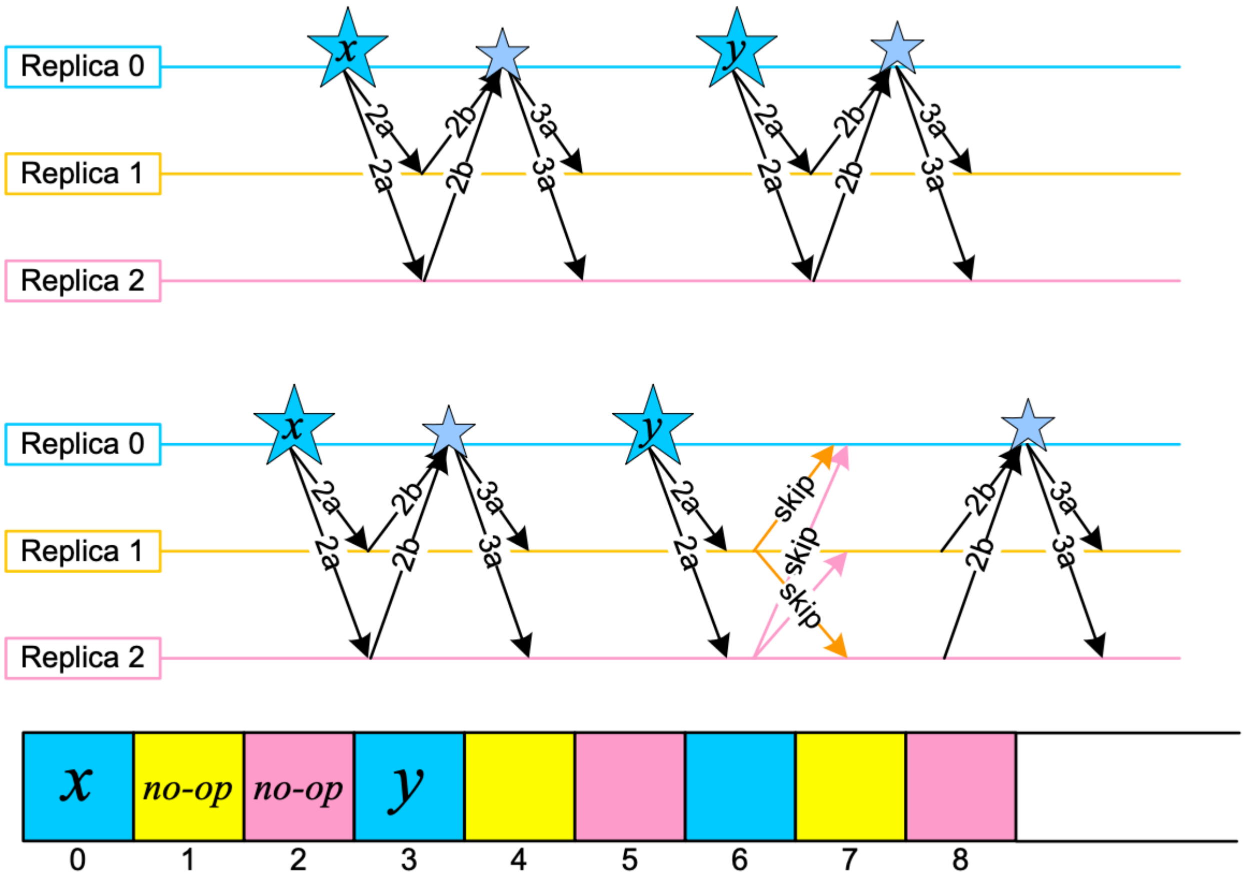
# *"Simple Consensus"*

- Simple consensus constraints:
  - 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-ops to improve efficiency (essentially at no extra cost)

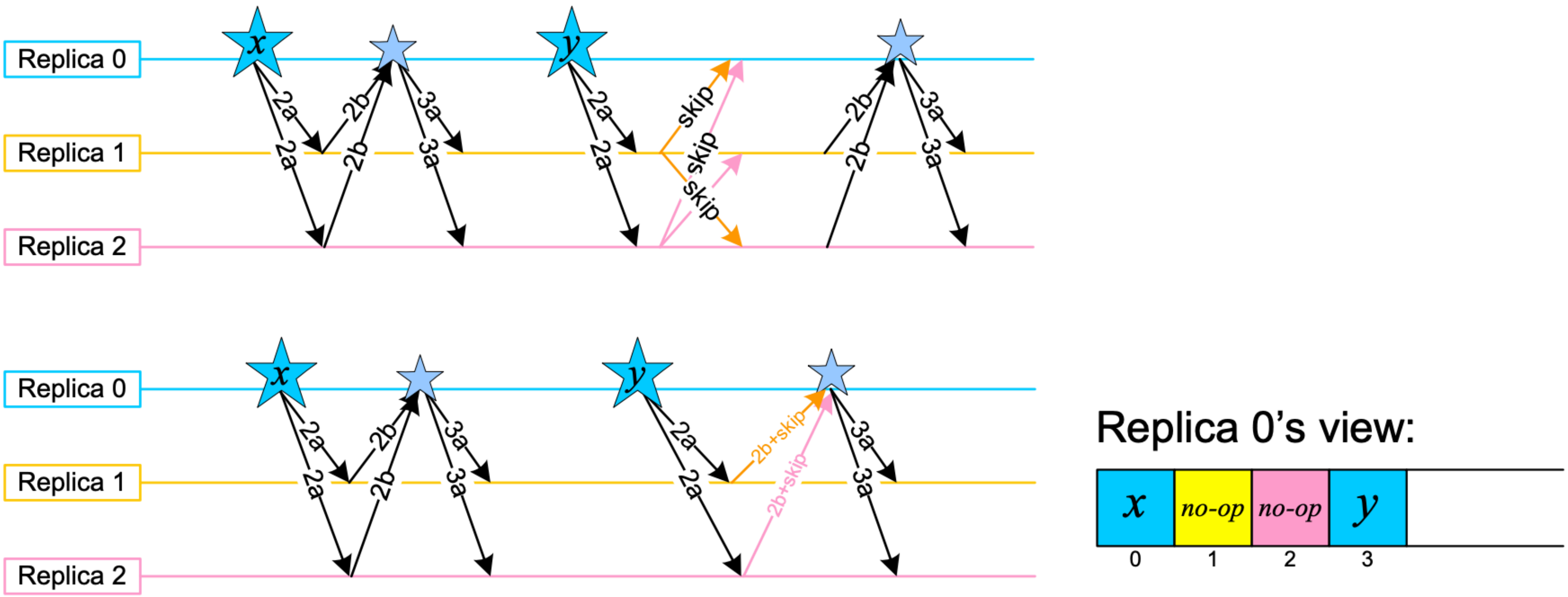
# *Coordinated Paxos*

- Starting state
  - The coordinator is the default leader
  - Start from state as if phase one is done by the coordinator
- Suggest
  - The coordinator proposes a request (Phase 2)
- Skip
  - The coordinator proposes a no-op; fast learning
- Revoke
  - A replica starts the full phase 1 & 2 and proposes a no-op
  - Only needed when failure is suspected

# *Skips with simple consensus*



# *Reduce message complexity*




- Piggyback skip to 2b messages as well as future 2a messages

# *Mencius optimizations*

- 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 requests to instances 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 instances smaller than *i*.

# *Gaps in idle replicas*

- Potentially unbounded number of requests wait to be committed
- 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 $\alpha$, or the message has been deferred for more than some time $\tau$.

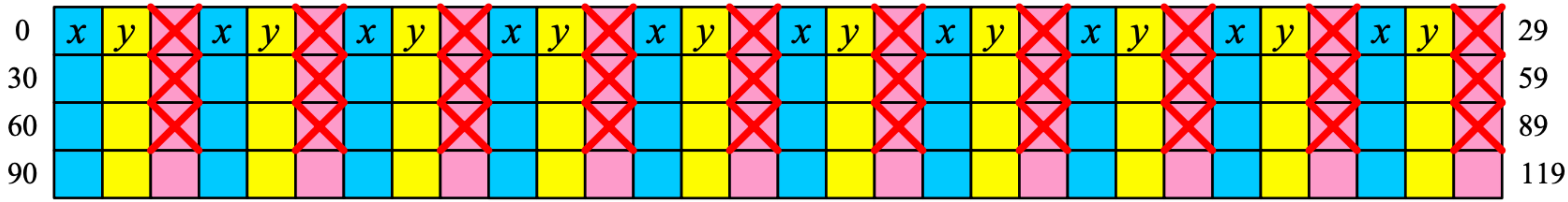
# *Failures*

- Faulty processes cannot skip
- How can we handle faults?

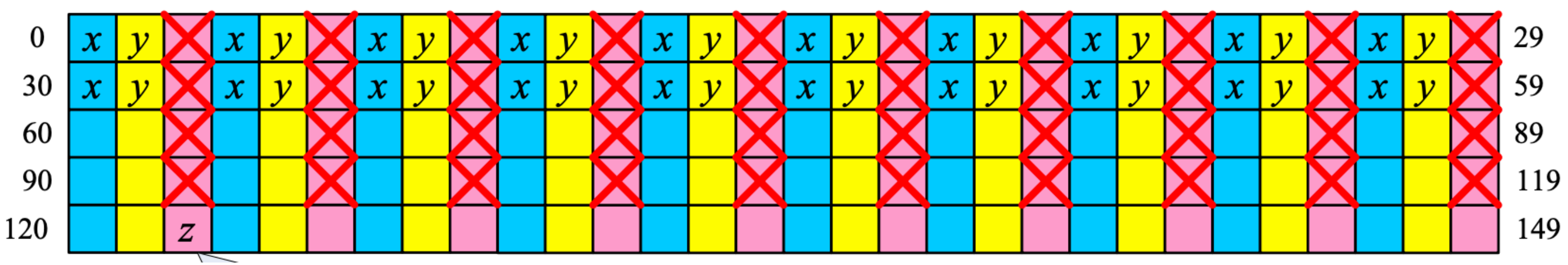
# *Revocation*

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

# Revocation & Recovery

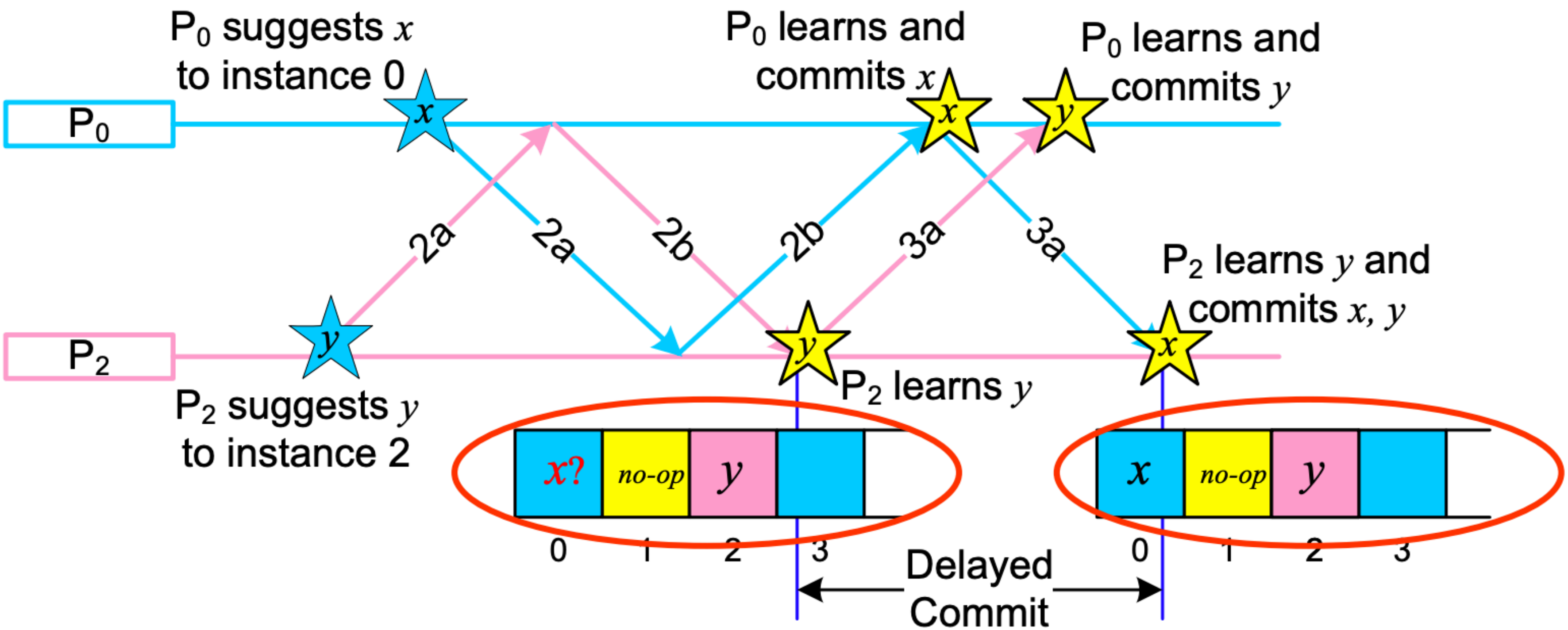



- Node may come back because of failure recovery or false suspicion
  - Find out the next available slots it coordinates
  - Start proposing request to that slot



# *Delayed Commit*

- Up to one RTT delay
- Out-of-order commits for commutable requests



# *Mencius Summary*

- Rotating leader Paxos
  - Easy to ensure safety and a flexible design
  - Simple consensus
- High performance
  - High throughput under high load
  - Low latency under low load
  - Better load balancing

# *Egalitarian Paxos*

- Similar goals:
  - High throughput, low latency
  - Fast failure recovery
  - Load balancing
  - Use closest/fastest replica (geographically distributed)

# *EPaxos Overview*

- Each replica has its own log of operations
  - Replica has “pre-prepared” all slots in its log
- No longer a single sequential log; order is unclear
- Each operation comes with a dependency list
  - Other operations have to run before it

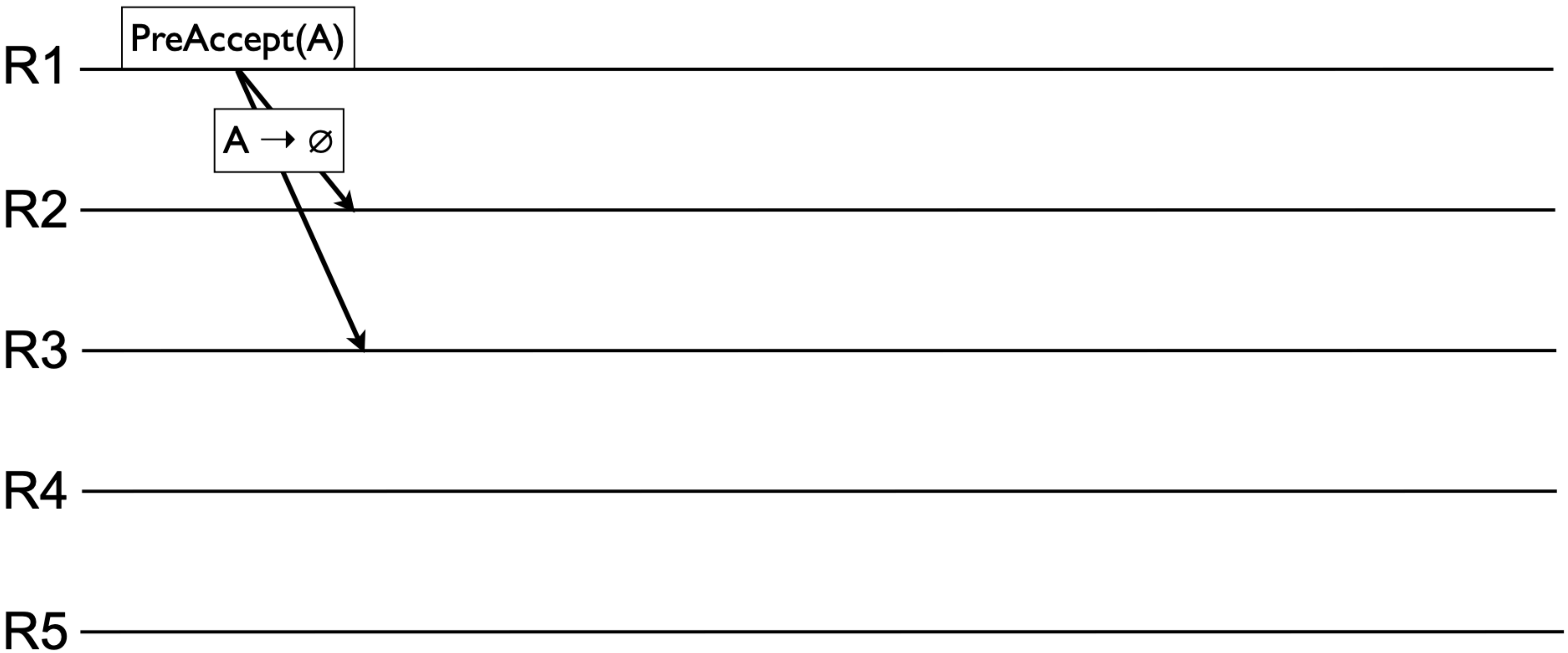
# *Dependencies*

- Issuing replica might know of the most recent conflicting operations
- During first round of messages, acceptors inform issuer about conflicts
  - Look at other instances
- Quorum rule means issuer learns about all conflicts
- First round of messages: pre-accept & ok
  - Ok response includes conflict list
  - If agreement on conflict list, then can commit

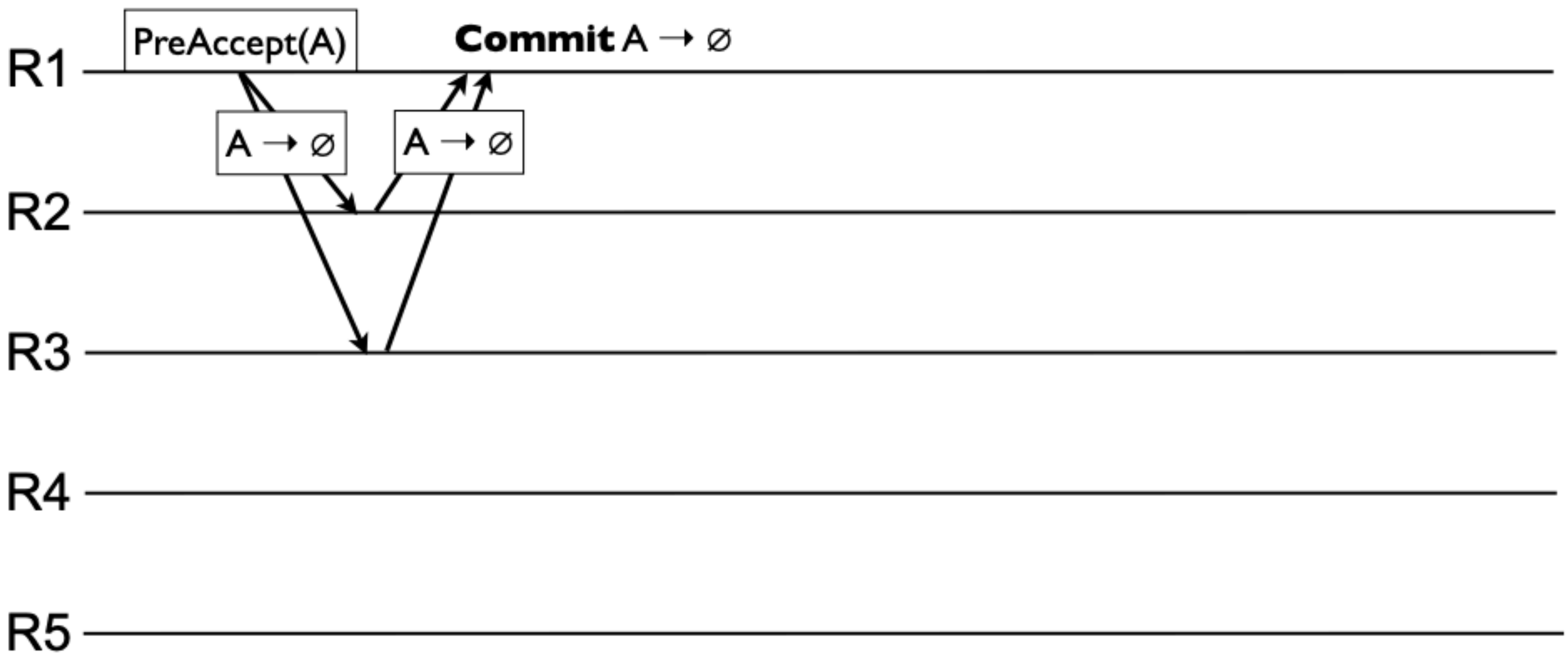
# *When do two commands conflict?*

- Interference is application specified
  - E.g., same key in KV store
  - For different keys, order does not matter

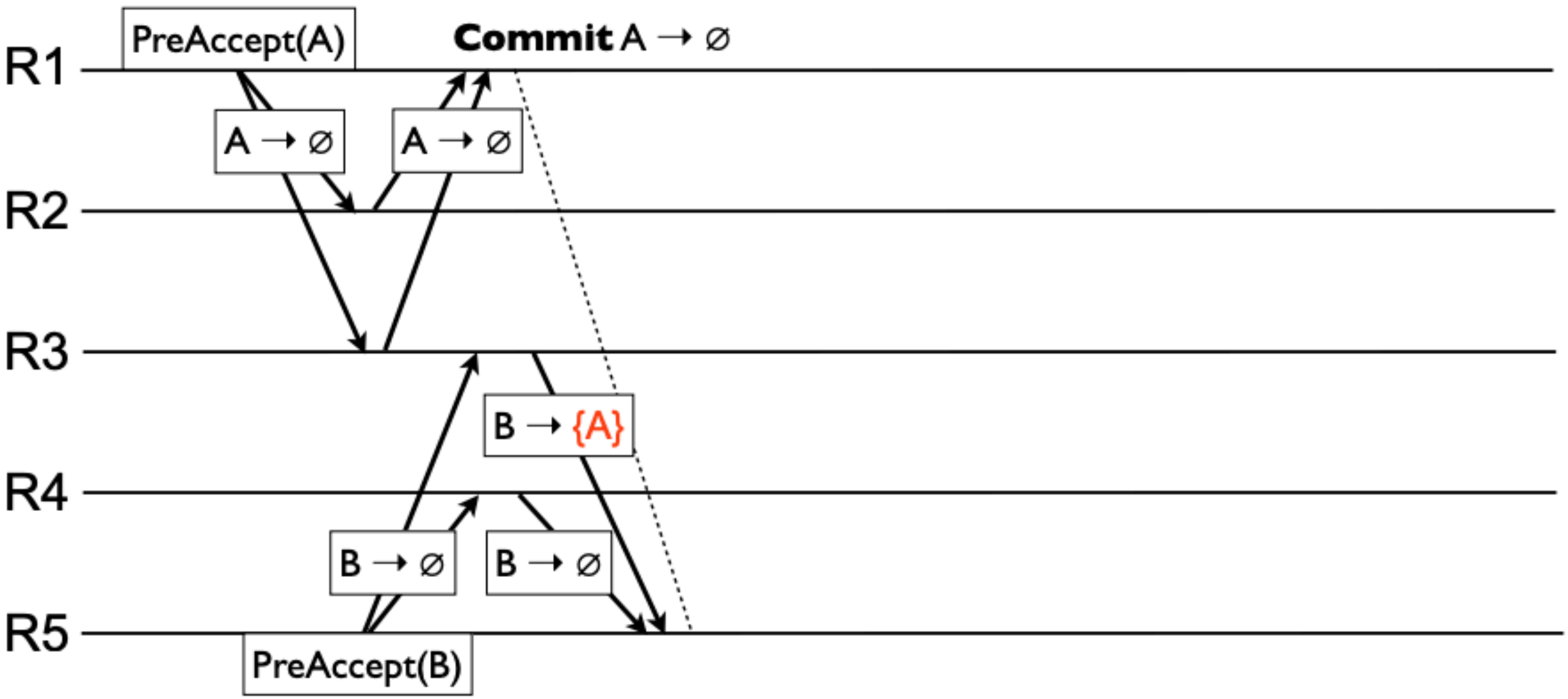
# *Protocol*



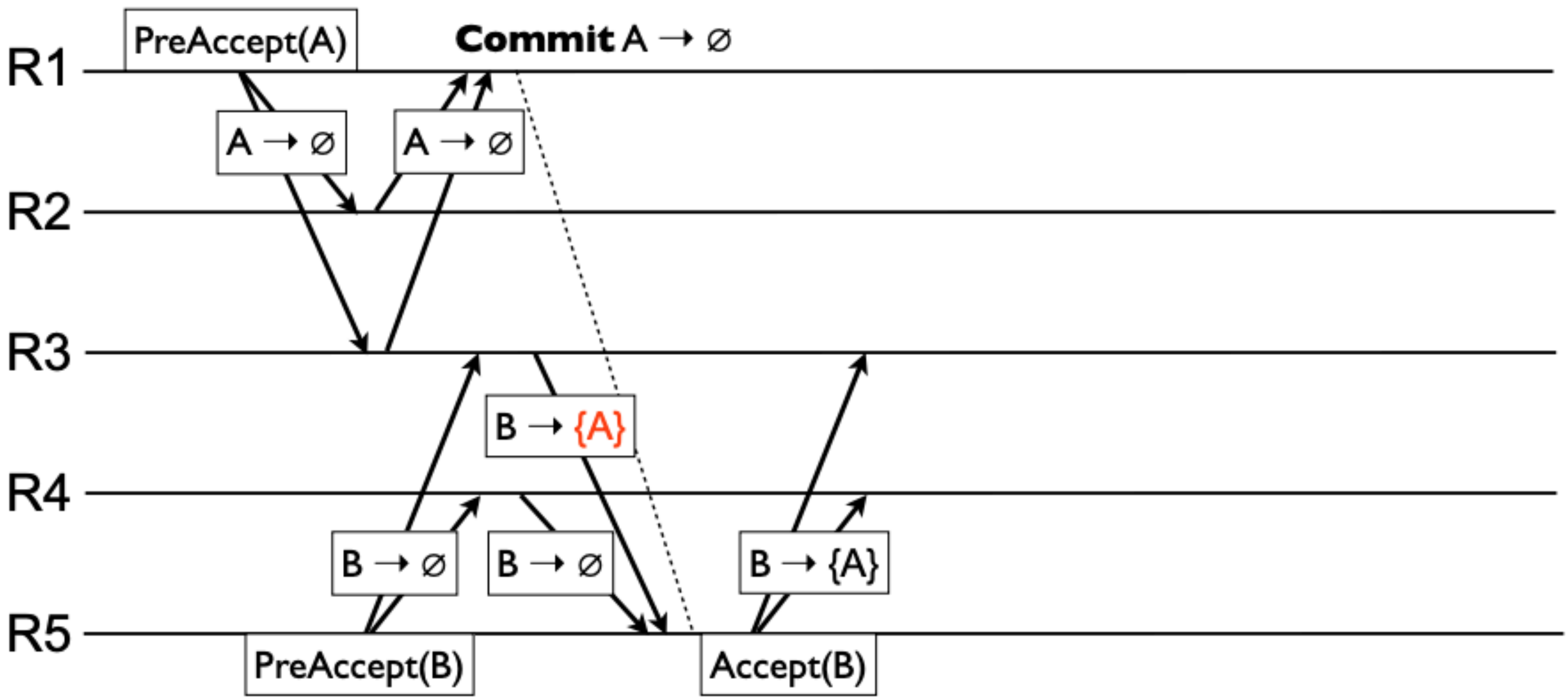
# *Protocol*



# *Protocol*

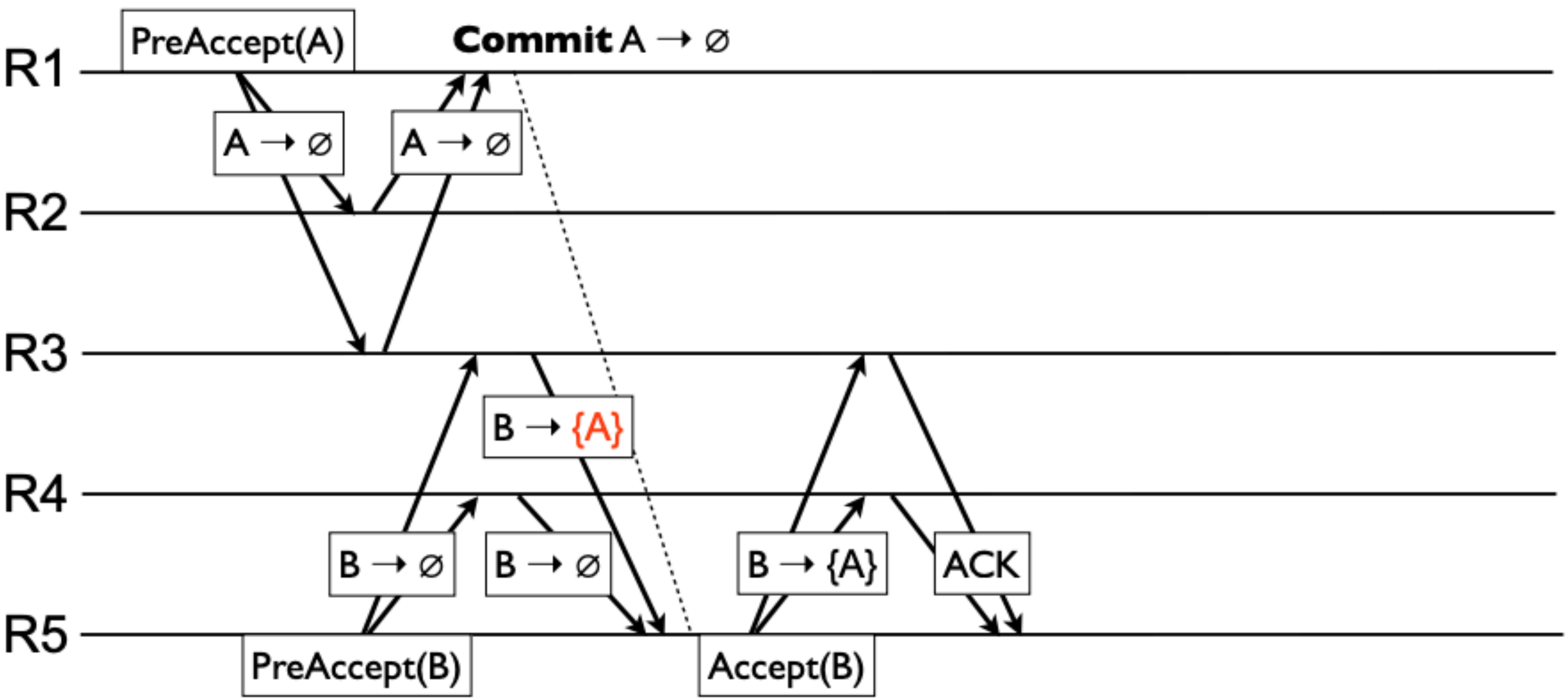


# Protocol



PreAccept(A)
Commit A → ∅
R1
A → ∅
A → ∅
R2
R3
B → {A}
R4
B → ∅
B → ∅
B → {A}
R5
PreAccept(B)
Accept(B)

# Protocol

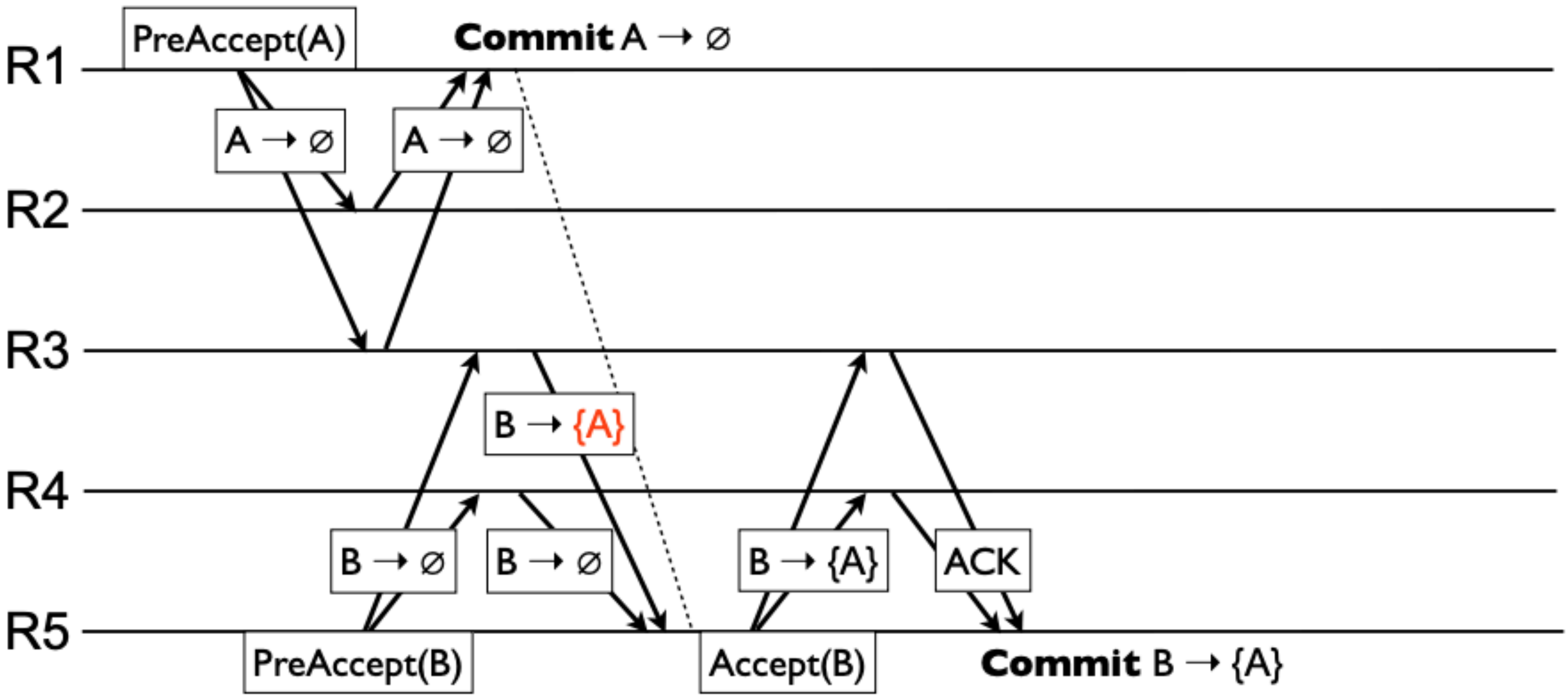


PreAccept(A)
Commit A → ∅
R1
A → ∅
A → ∅
R2
R3
B → {A}
R4
B → ∅
B → ∅
B → {A}
ACK
R5
PreAccept(B)
Accept(B)

# *What is a quorum?*

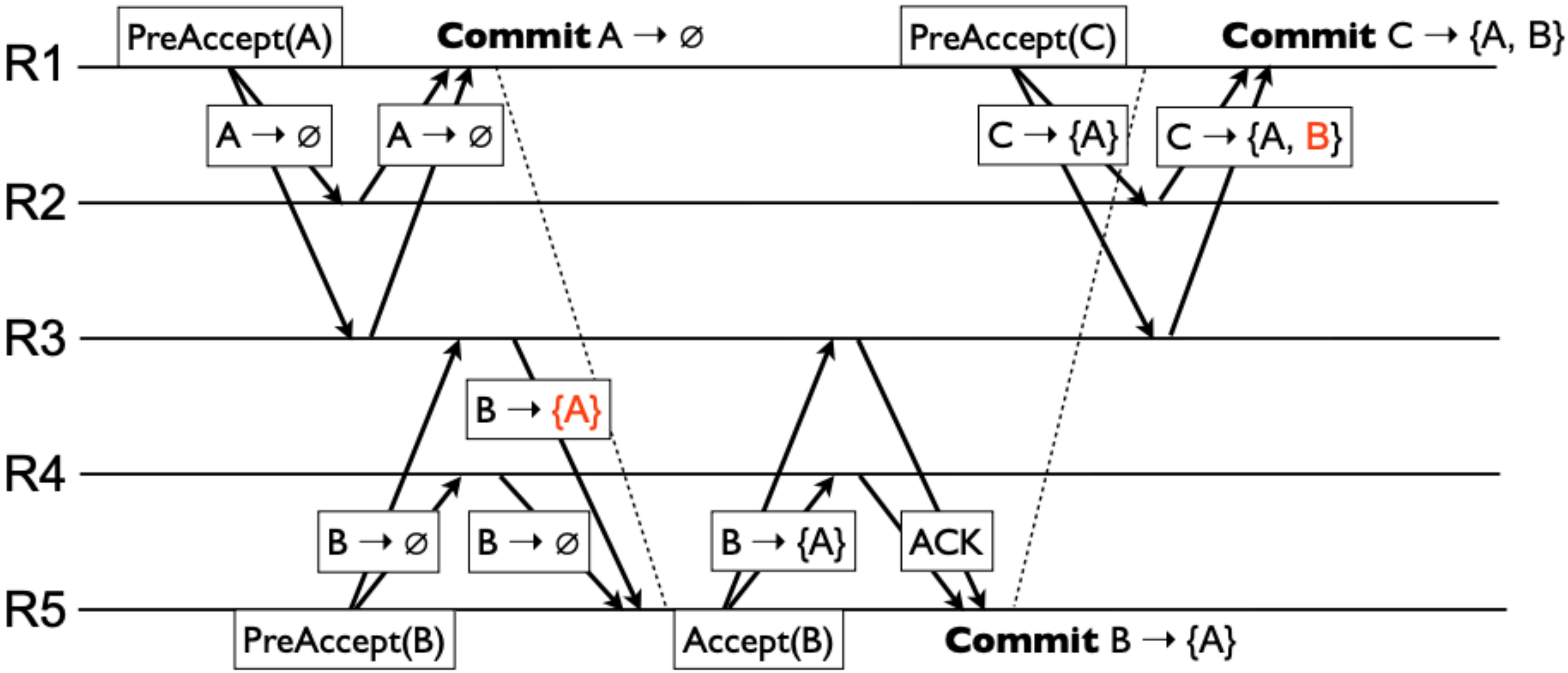
- For fast path, it is f + floor((f+1)/2)
- For slow path, it is f + 1
- Quorum sizes are dependent on the recovery protocol
  - What happens when the proposer fails?
  - Can we reconstruct the state that the proposer might have obtained?

# *Protocol*



R1
R2
R3
R4
R5
PreAccept(A)
Commit A → ∅
A → ∅
A → ∅
B → {A}
B → ∅
B → ∅
B → {A}
ACK
PreAccept(B)
Accept(B)
Commit B → {A}

# Protocol



PreAccept(A)
Commit A → ∅
PreAccept(C)
Commit C → {A, B}
R1
A → ∅
A → ∅
C → {A}
C → {A, B}
R2
R3
B → {A}
R4
B → ∅
B → ∅
B → {A}
ACK
R5
PreAccept(B)
Accept(B)
Commit B → {A}

# *When is second phase needed?*

- If all responses are same, quorum of nodes remember the pre-accept-ok response
  - If proposer crashes, recovery ensures that operation will go into the slot
- If some differ, need to agree on what is the slot

# *Order only interfering commands*

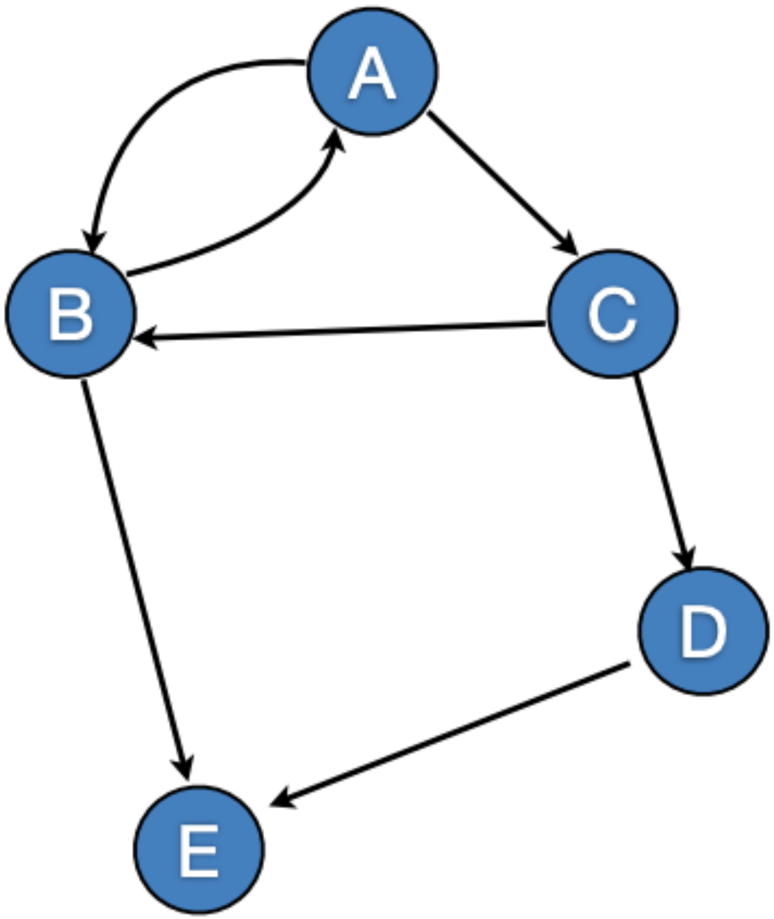
- 1 RTT
  - Non-concurrent commands
  - OR non-interfering commands

- 2 RTTs
  - Concurrent and interfering commands

# *Ordering with dependencies*

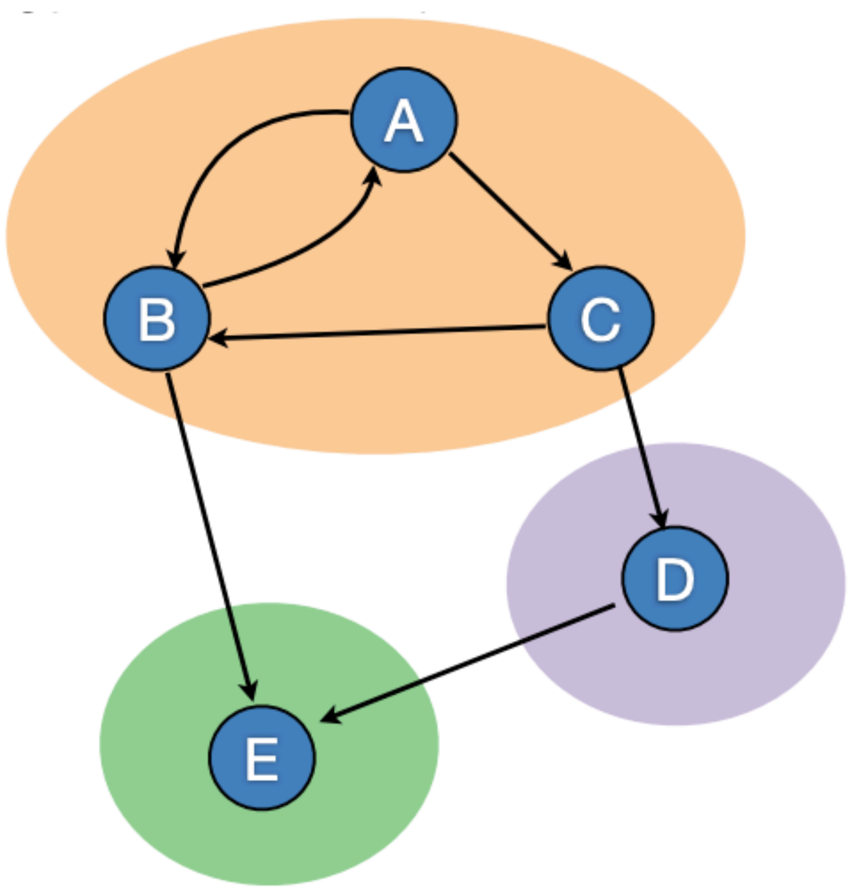
- If two conflicting commands are committed, at least one has a dependency on the other
- Execute dependencies of a command first
  - But, if commands are mutually dependent, use a deterministic algorithm to compute their order

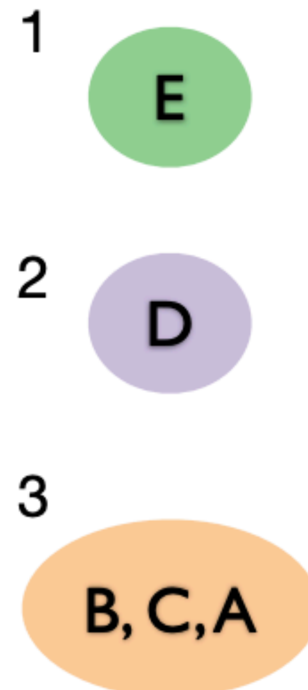
# Execution



# *Execution*

- Order strongly connected components




Commands in SCC ordered by replica sequence numbers

# *Other details*

- Dependency list size: safe to include just the latest conflicting operation
- EPaxos could potentially execute a transaction — needs to take into account dependencies of all operations
- Recovery could be complex

# *Performance Gains*

- Balanced load
- Client can use the closest replica as the coordinator
- Replica can use closest replicas as its quorum