Paxos Made Simple

Lamport

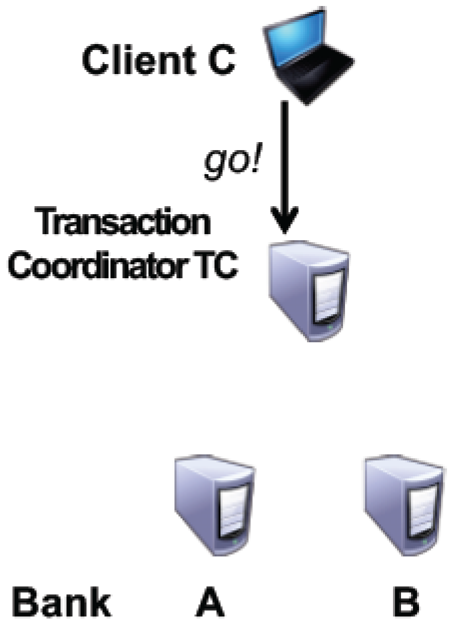
Distributed transactions?

- Partition databases across multiple machines for scalability (A and B might not share a server)
- A transaction might touch more than one partition
- How do we guarantee that all of the partitions commit the transactions or none commit the transactions?

Single-server: ACID

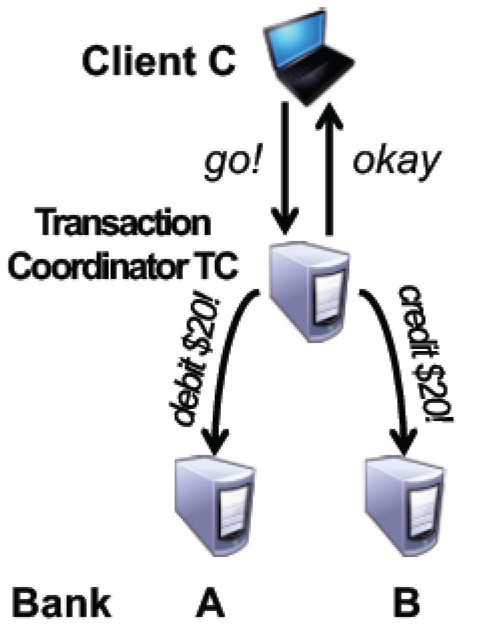
- **Atomicity**: all parts of the transaction execute or none (A's decreases and B's balance increases)
- **Consistency**: the transaction only commits if it preserves invariants (A's balance never goes below 0)
- **Isolation**: the transaction executes as if it executed by itself (even if C is accessing A's account, that will not interfere with this transaction)
- **Durability**: the transaction's effects are not lost after it executes (updates to the balances will remain forever)

Straw Man protocol



1. C → TC: *"go!"*

Straw Man protocol



1. **C → TC**: *“go!”*

2. **TC → A**: *“debit $20!”*
   TC → B: *“credit $20!”*
   TC → C: *“okay”*

- **A, B** perform actions on receipt of messages

Reasoning about the Straw Man protocol

What could **possibly** go wrong?

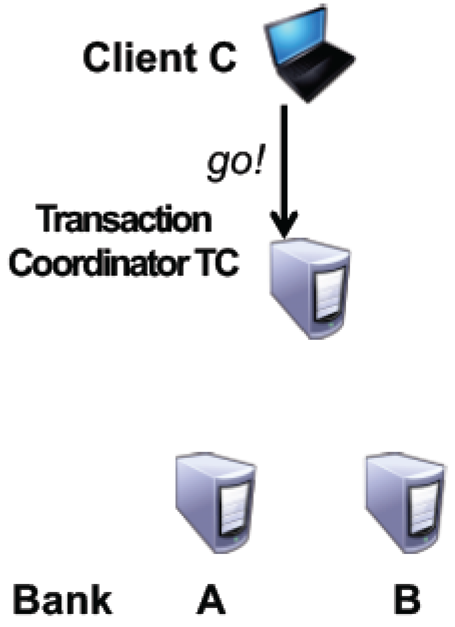
1. Not enough money in **A's** bank account?
2. **B's** bank account no longer exists?
3. **A** or **B** **crashes** before receiving message?
4. The best-effort network to **B** **fails**?
5. **TC** **crashes** after it sends *debit* to **A** but before sending to **B**?

Safety versus liveness

- Note that **TC**, **A**, and **B** each have a notion of committing
- We want two properties:

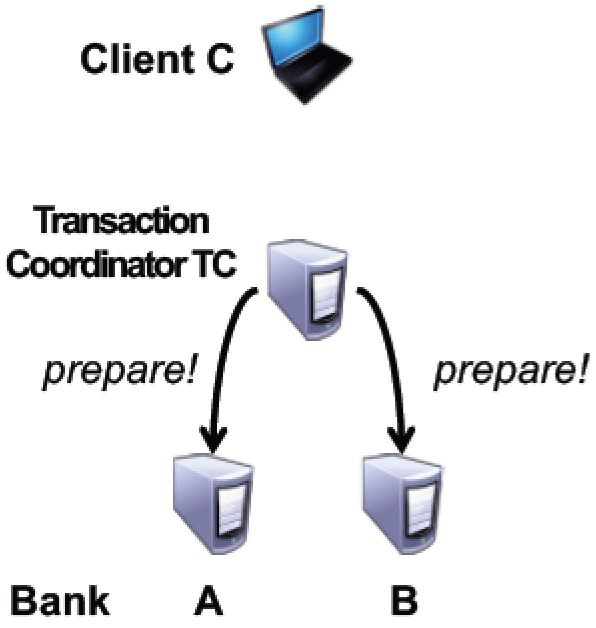
1. Safety
   - If **one commits, no one aborts**
   - If **one aborts, no one commits**
2. Liveness
   - If **no failures** and **A** and **B** can commit, **action commits**
   - If **failures,** reach a conclusion ASAP

A *correct* atomic commit protocol



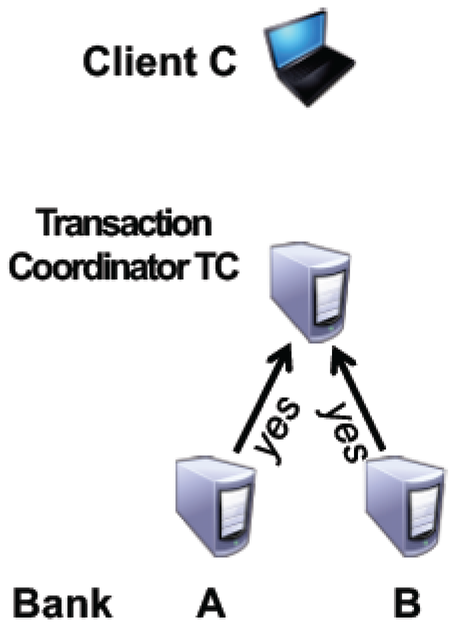
1. C → TC: *"go!"*

A *correct* atomic commit protocol



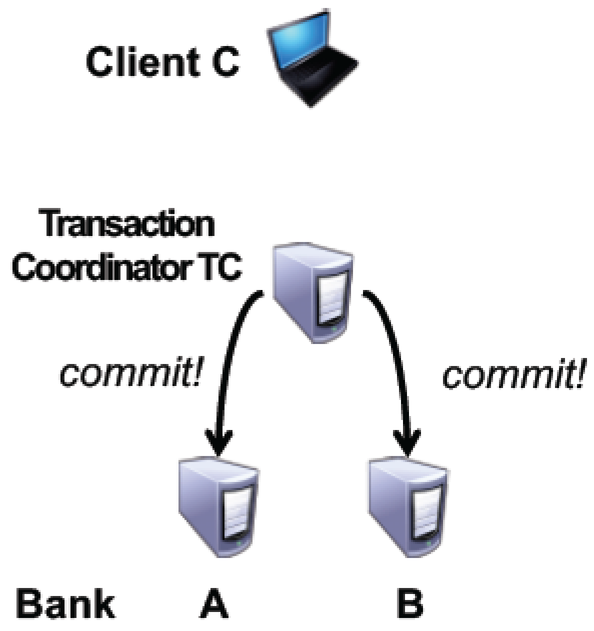
1. C → TC: *"go!"*
2. TC → A, B: *"prepare!"*

A *correct* atomic commit protocol



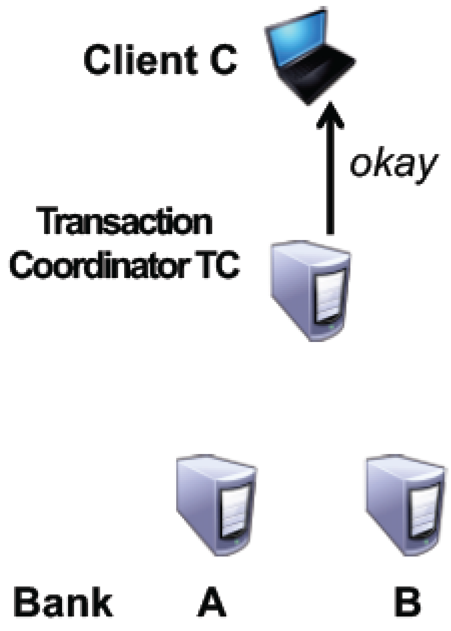
1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → P:** *"yes"* or *"no"*

A *correct* atomic commit protocol



1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → P:** *"yes"* or *"no"*
4. **TC → A, B:** *"commit!"* or *"abort!"*
   - **TC** sends ***commit*** if **both** say *yes*
   - **TC** sends ***abort*** if **either** say *no*

A *correct* atomic commit protocol



1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → P:** *"yes"* or *"no"*
4. **TC → A, B:** *"commit!"* or *"abort!"*
   – **TC** sends ***commit*** if **both** say *yes*
   – **TC** sends ***abort*** if **either** say *no*
5. **TC → C:** *"okay" or "failed"*

- **A, B** commit on receipt of commit message

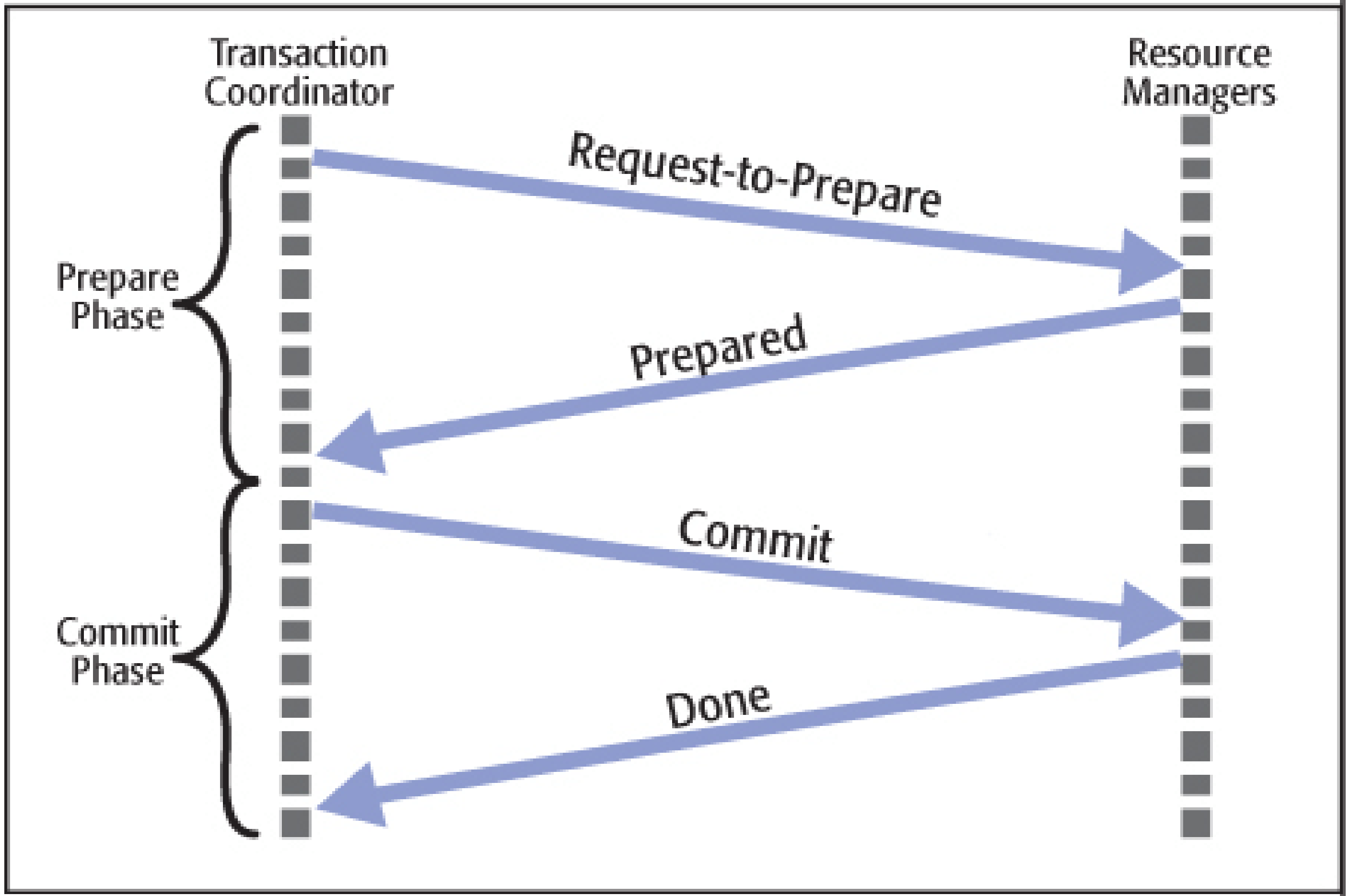
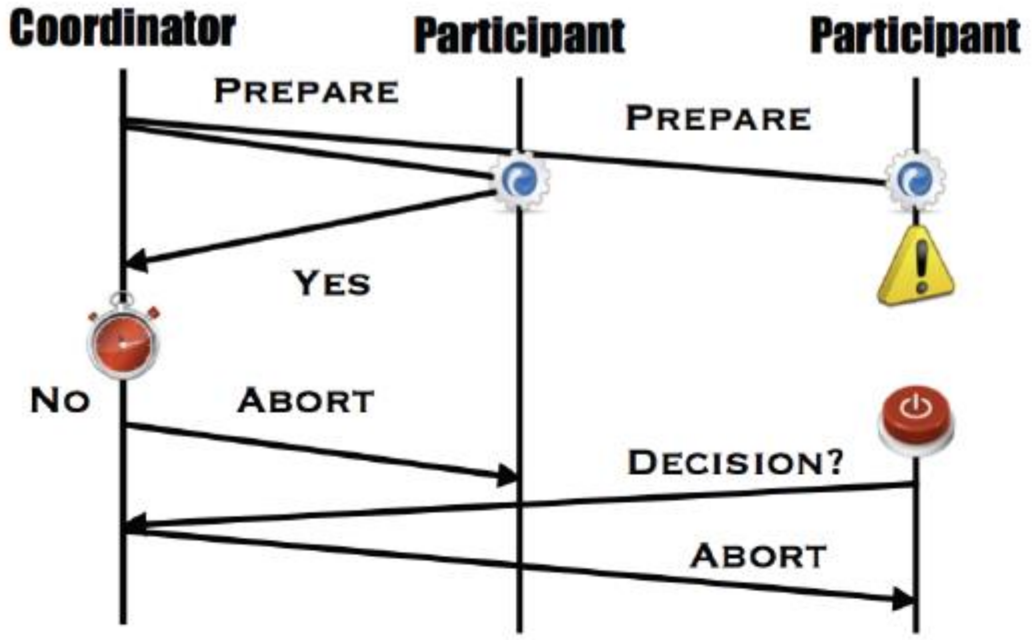
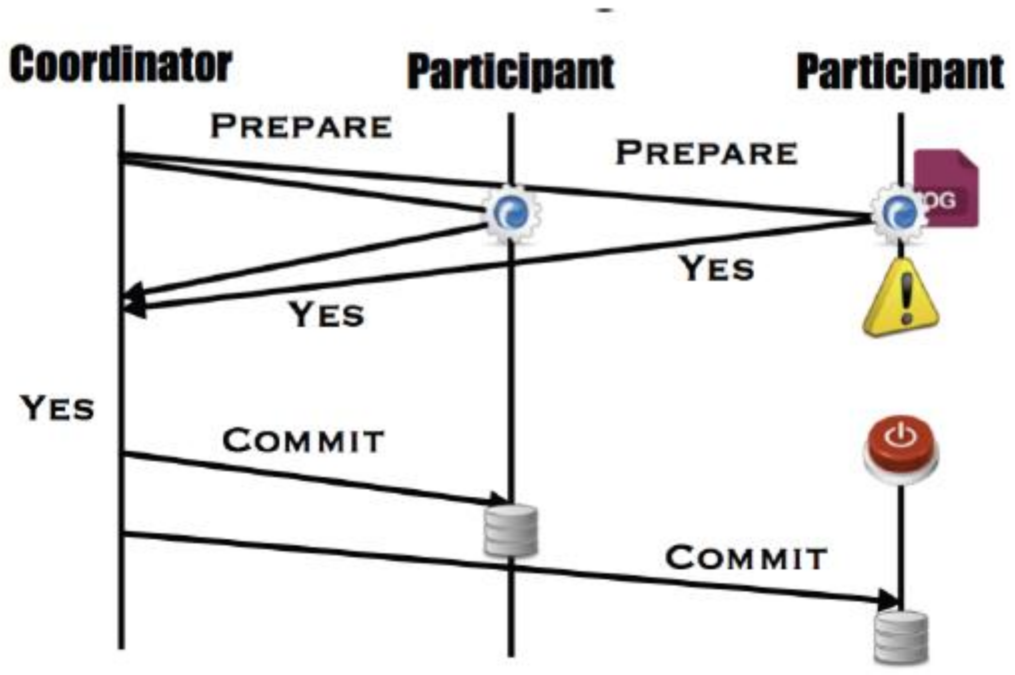


Figure 1 • The two-phase commit protocol

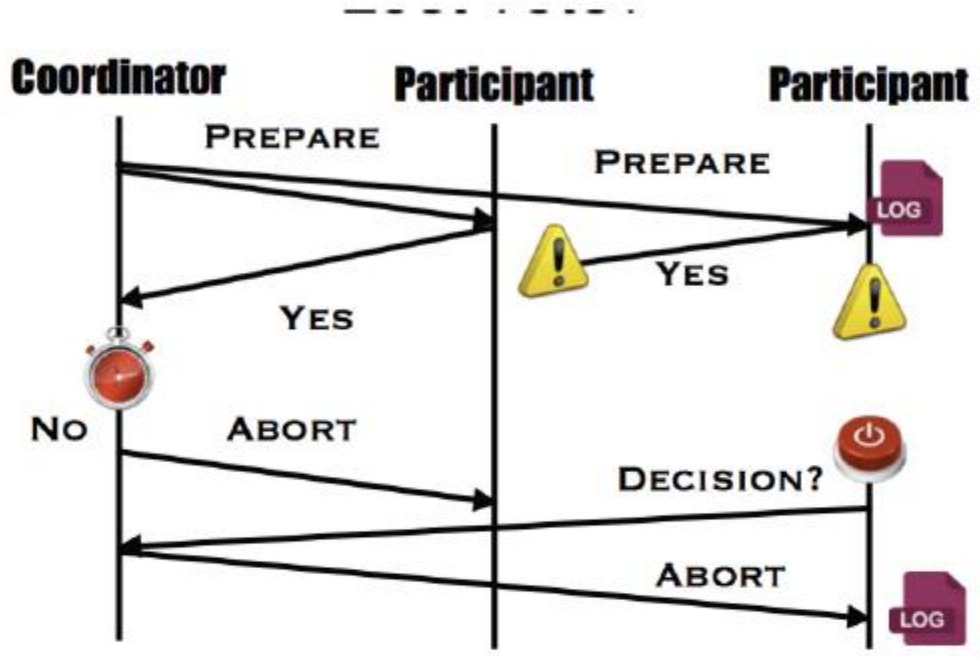
What if participant fails before sending response?



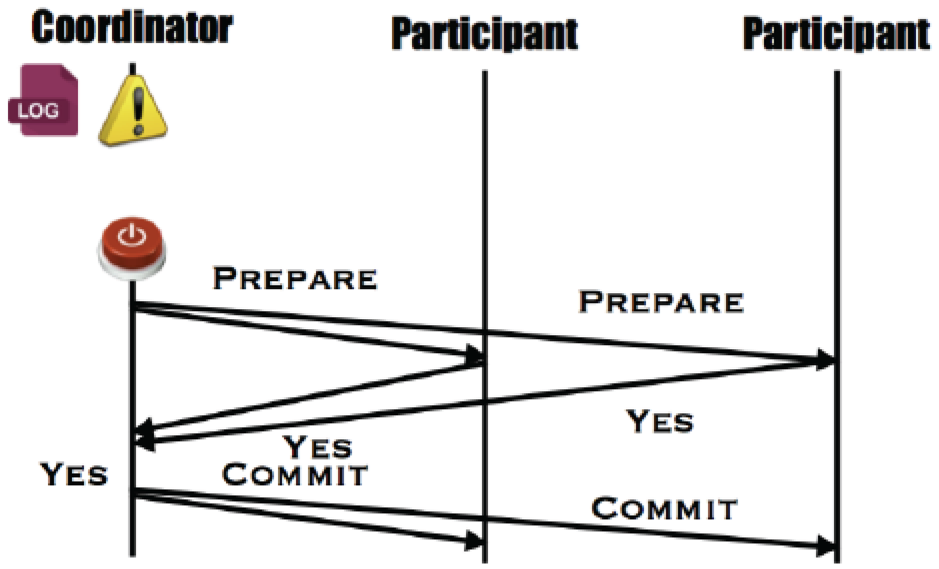
What if participant fails after sending vote



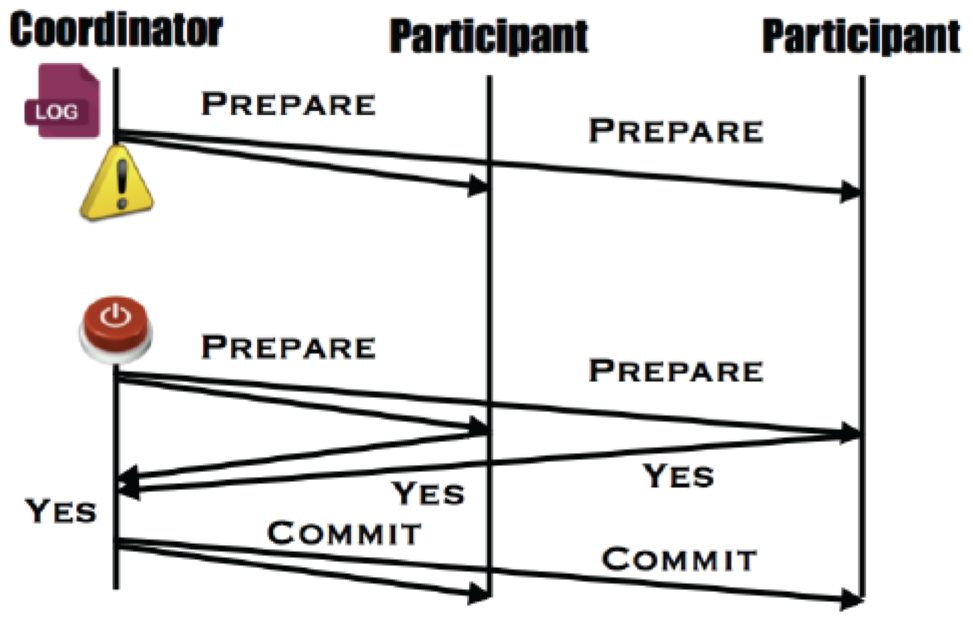
What if participant lost a vote?



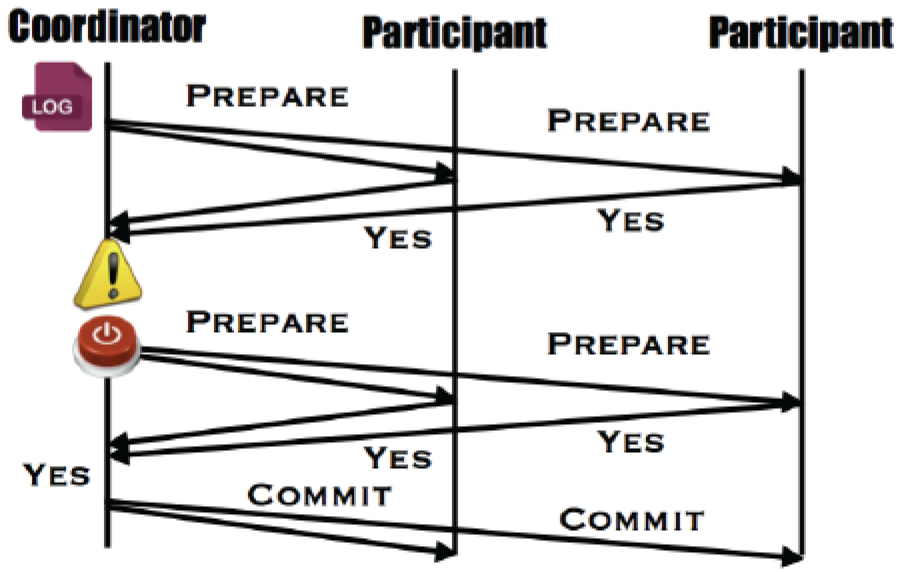
What if coordinator fails before sending prepare?



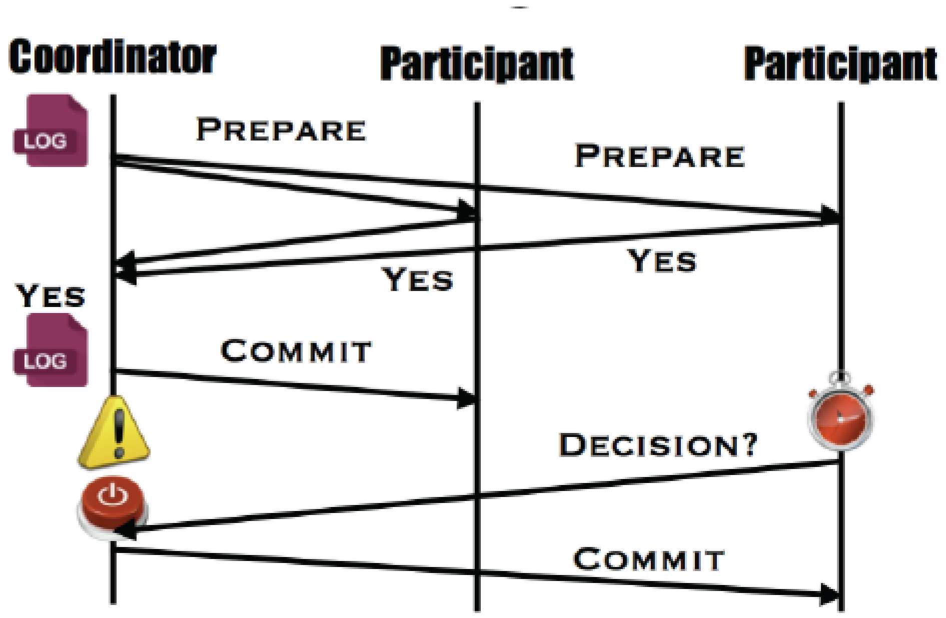
What if coordinator fails after sending prepare?



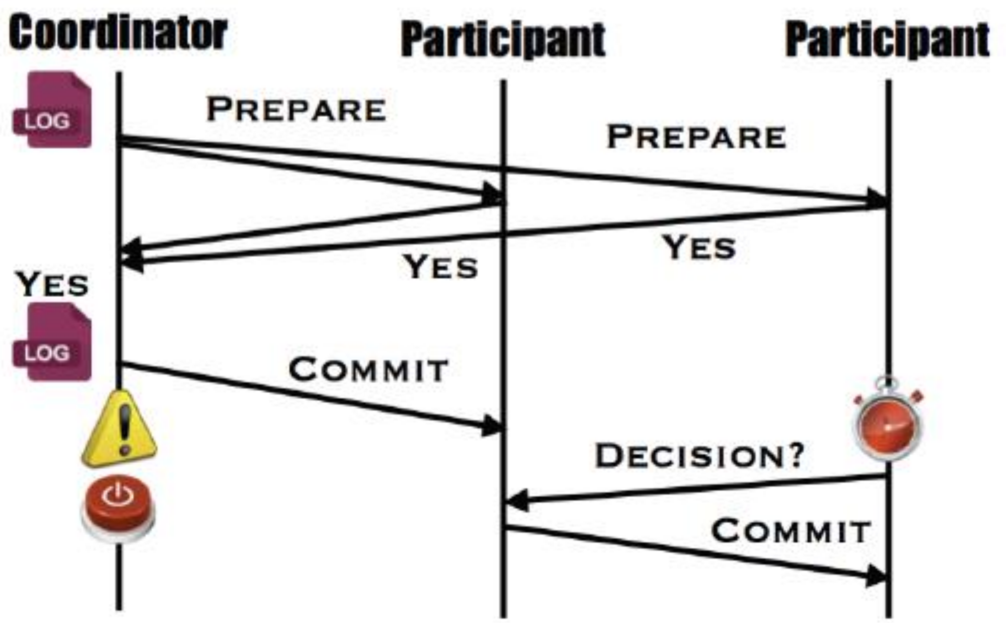
What if coordinator fails after receiving votes



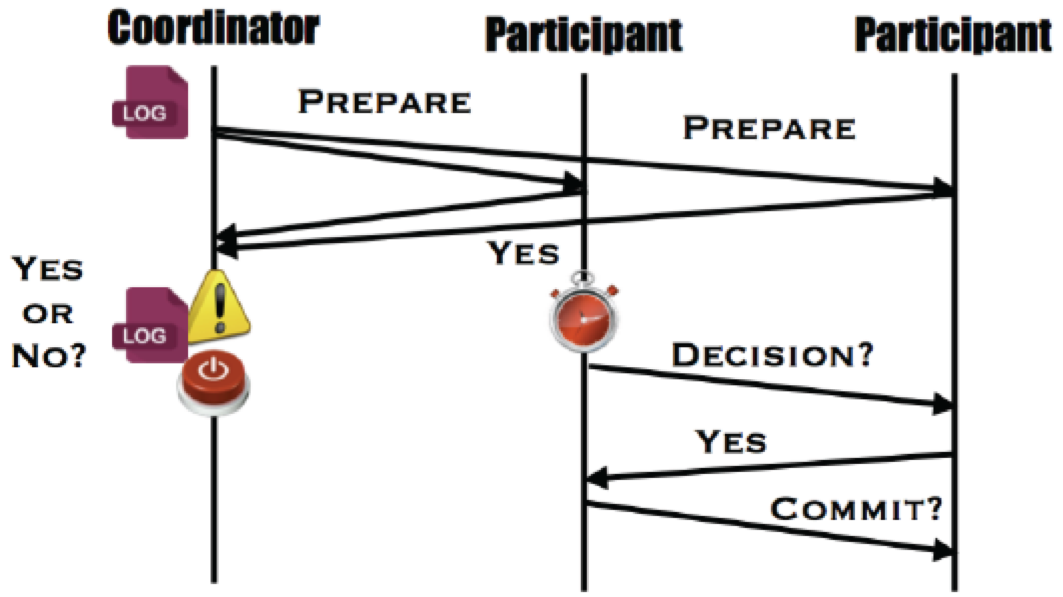
What if coordinator fails after sending decision?



Do we need the coordinator?

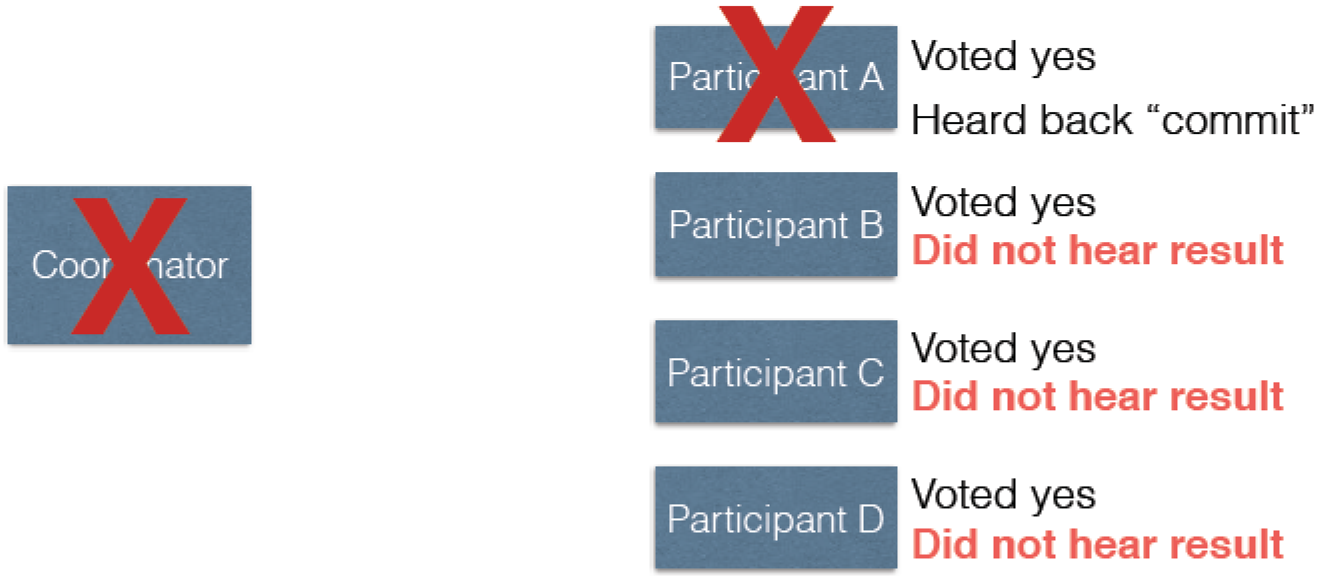


What happens if we don't have a decision?

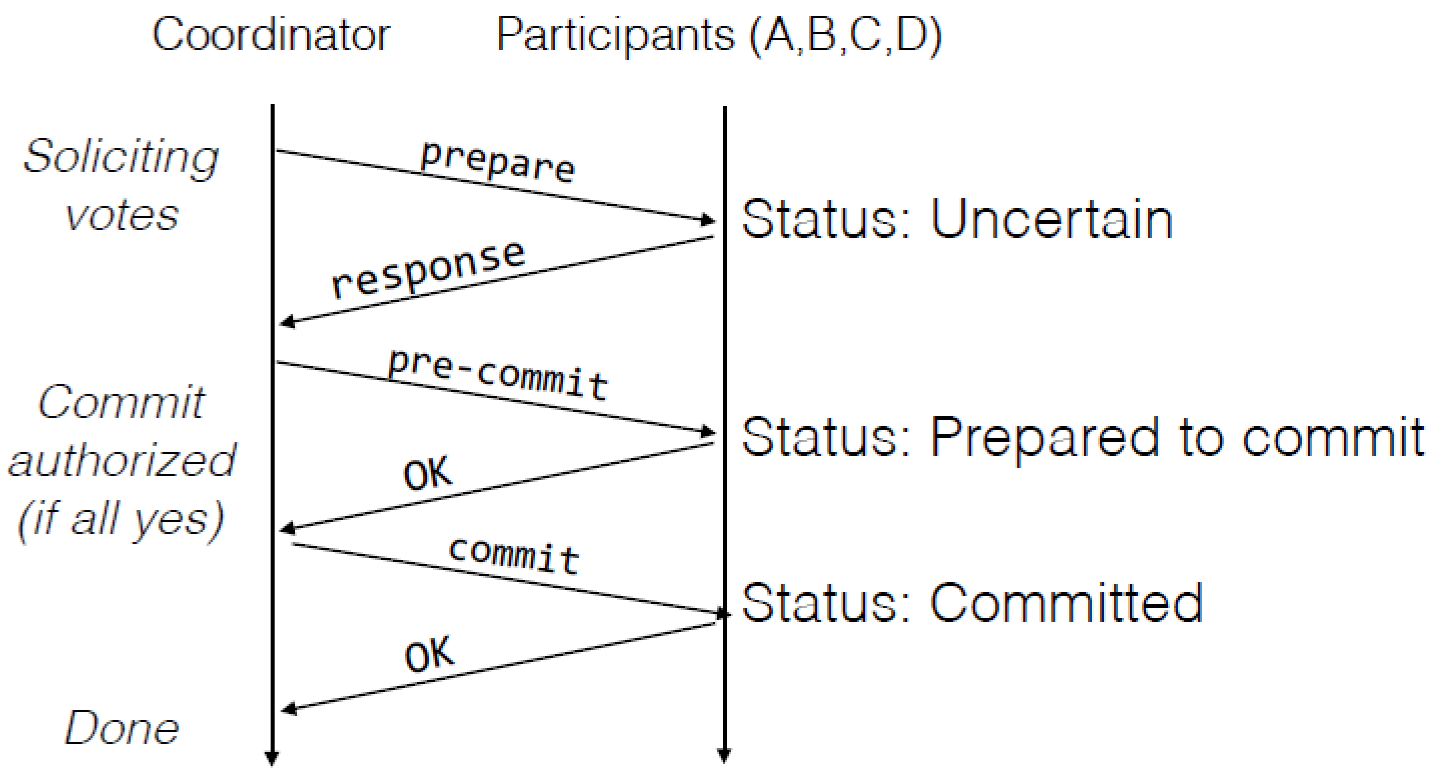


3 Phase Commit

- Goal: Eliminate this specific failure from blocking liveness



3PC Example


Coordinator
Participants (A,B,C,D)
Soliciting votes
prepare
Status: Uncertain
response
pre-commit
Commit authorized (if all yes)
Status: Prepared to commit
OK
commit
Status: Committed
OK
Done

3 Phase Commit

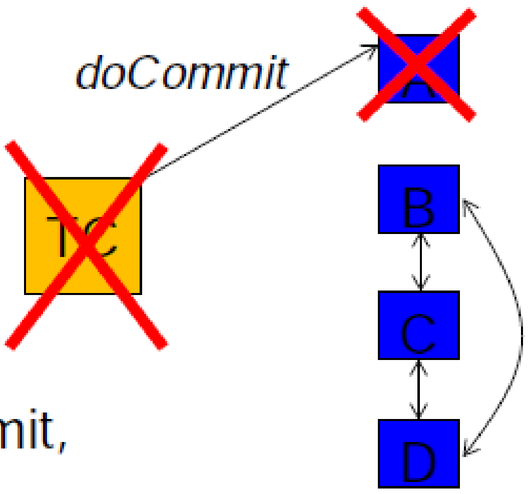
- Goal: Avoid blocking on node failure
- How?
  - Think about how 2PC is better than 1PC
    - 1PC means you can never change your mind or have a failure after committing
    - 2PC **still** means that you can't have a failure after committing (committing is irreversible)
- 3PC idea:
  - Split commit/abort into 2 sub-phases
    - 1: Tell everyone the outcome
    - 2: Agree on outcome

Can 3PC Solve Blocking 2PC Ex.?

- Assuming same scenario as before (TC, A crash), can B/C/D reach a safe decision when they time out?

1. If one of them has received preCommit, they can all commit
   - This is safe if we assume that A is DEAD and after coming back it runs a recovery protocol in which it requires input from B/C/D to complete an uncommitted transaction
   - This conclusion was impossible to reach for 2PC b/c A might have already committed and exposed outcome of transaction to world
2. If none of them has received preCommit, they can all abort
   - This is safe, b/c we know A couldn't have received a doCommit, so it couldn't have committed

Phase 3: Commit



3PC is safe for node crashes (including TC+participant)

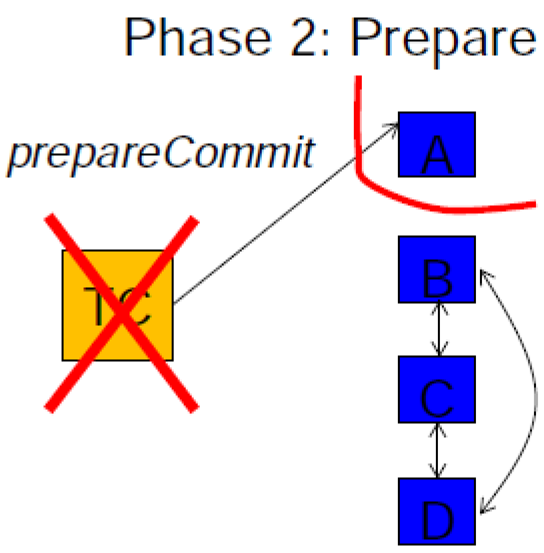
But Does 3PC Achieve Consensus?

- Liveness (availability): Yep
  - Doesn't block, it always makes progress by timing out
- Safety (correctness): Nope
  - Can you think of scenarios in which original 3PC would result in inconsistent states between the replicas?
- Two examples of unsafety in 3PC:
  - A hasn't crashed, it's just offline
  - TC hasn't crashed, it's just offline

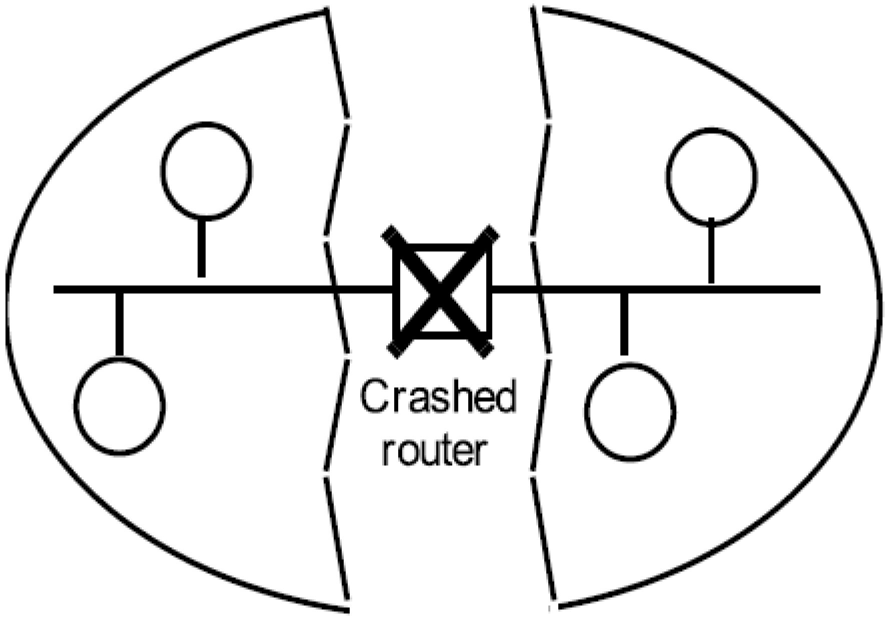
Network partitions

3PC with Network Partitions

- One example scenario:
  - A receives prepareCommit from TC
  - Then, A gets partitioned from B/C/D and TC crashes
  - None of B/C/D have received prepareCommit, hence they all abort upon timeout
  - A is prepared to commit, hence, according to protocol, after it times out, it unilaterally decides to commit
- Similar scenario with partitioned, not crashed, TC



A network partition



Safety vs. Liveness

- So, 3PC is doomed for network partitions
  - The way to think about it is that this protocol's design trades safety for liveness
- Remember that 2PC traded liveness for safety
- Can we design a protocol that's both safe and live?
- Well, it turns out that it's impossible in the most general case!

Fischer-Lynch-Paterson [FLP'85] Impossibility Result

- It is impossible for a set of processors in an asynchronous system to agree on a binary value, even if only a single process is subject to an unannounced failure
  - We won't show any proof here – it's too complicated
- The core of the problem is asynchrony
  - It makes it impossible to tell whether or not a machine has crashed (and therefore it will launch recovery and coordinate with you safely) or you just can't reach it now (and therefore it's running separately from you, potentially doing stuff in disagreement with you)
- For synchronous systems, 3PC can be made to guarantee both safety and liveness!
  - When you know the upper bound of message delays, you can infer when something has crashed with certainty

FLP – Translation

- What FLP says: you can't guarantee both safety and progress when there is even a single fault at an inopportune moment
- What FLP doesn't say: in practice, how close can you get to the ideal (always safe and live)?
- Next: Paxos algorithm, which in practice gets close

Paxos

Background

- Jim Gray proposes 2PC in the 1970s, but it blocks on single node failures.
- Dale Skeen proposes 3PC in the 1980s, but it produces incorrect results in some situations.
- Leslie Lamport proposes Paxos in 1998; the original paper describes the ancient Greek civilization on the Paxos island.

Paxos Is Everywhere

- Widely used in both industry and academia
- Examples:
  - Google: Chubby (Paxos-based distributed lock service)
    - Most Google services use Chubby directly or indirectly
  - Yahoo: Zookeeper (Paxos-based distributed lock service)
  - MSR: Frangipani (Paxos-based distributed lock service)
    - The YFS labs contain a Paxos assignment, hopefully we'll get to it
  - UW: Scatter (Paxos-based consistent DHT)
  - Open source:
    - libpaxos (Paxos-based atomic broadcast)
    - Zookeeper is open-source and integrates with Hadoop

Paxos Properties

- Safety
  - If agreement is reached, everyone agrees on the same value
  - The value agreed upon was proposed by some node
- Fault tolerance (i.e., as-good-as-it-gets liveness)
  - If less than half the nodes fail, the rest nodes reach agreement *eventually*
- No guaranteed termination (i.e., imperfect liveness)
  - Paxos may not always converge on a value, but only in very degenerate cases that are improbable in the real world
- Ah, and lots of awesomeness ☺
  - Basic idea seems natural in retrospect, but why it works in any detail is incredibly complex!

Challenges Addressed in Paxos

- What if multiple nodes become proposers simultaneously?
- What if the new proposer proposes different values than an already decided value?
- What if there is a network partition?
- What if a proposer crashes in the middle of solicitation?
- What if a proposer crashes after deciding but before announcing results?
- …

- Similar concerns occur in the party example – go over scenario

Core Differentiating Mechanisms

1. Proposal ordering
   - Lets nodes decide which of several concurrent proposals to accept and which to reject

2. Majority voting
   - 2PC needs all nodes to vote Yes before committing
     - As a result, 2PC may block when a single node fails
   - Paxos requires only a majority of the acceptors (half+1) to accept a proposal
     - As a result, in Paxos nearly half the nodes can fail to reply and the protocol continues to work correctly
     - Moreover, since no two majorities can exist simultaneously, network partitions do not cause problems (as they did for 3PC)

Paxos

- We want to choose a value and have every node in the cluster agree on the value.
- Three classes of agents: proposers, acceptors, learners.
- Failures are possible, but non-Byzantine.

Paxos: High Level

- One (or more) nodes decide to be leader (proposer)
- Leader proposes a value, solicits acceptance from the rest of the nodes
- Leader announces chosen value, or tries again if it failed to get all nodes to agree on that value
- Lots of tricky corners (failure handling)
- In sum: requires only a majority of the (non-leader) nodes to accept a proposal for it to succeed

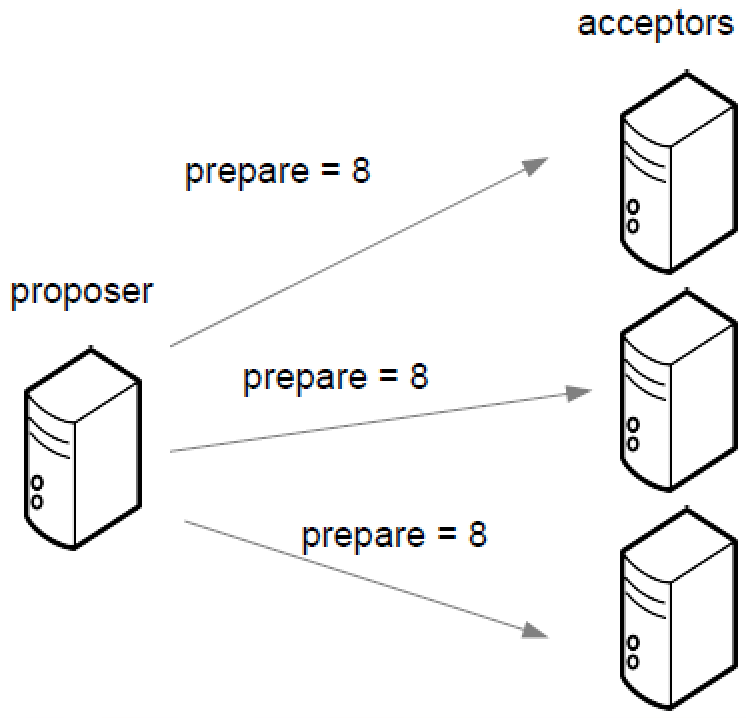
Algorithm

- A proposer selects a proposal number *n* and sends a request to the acceptors.
- If an acceptor has not already accepted a proposal with number greater than *n*, it responds that it can accept this proposal.

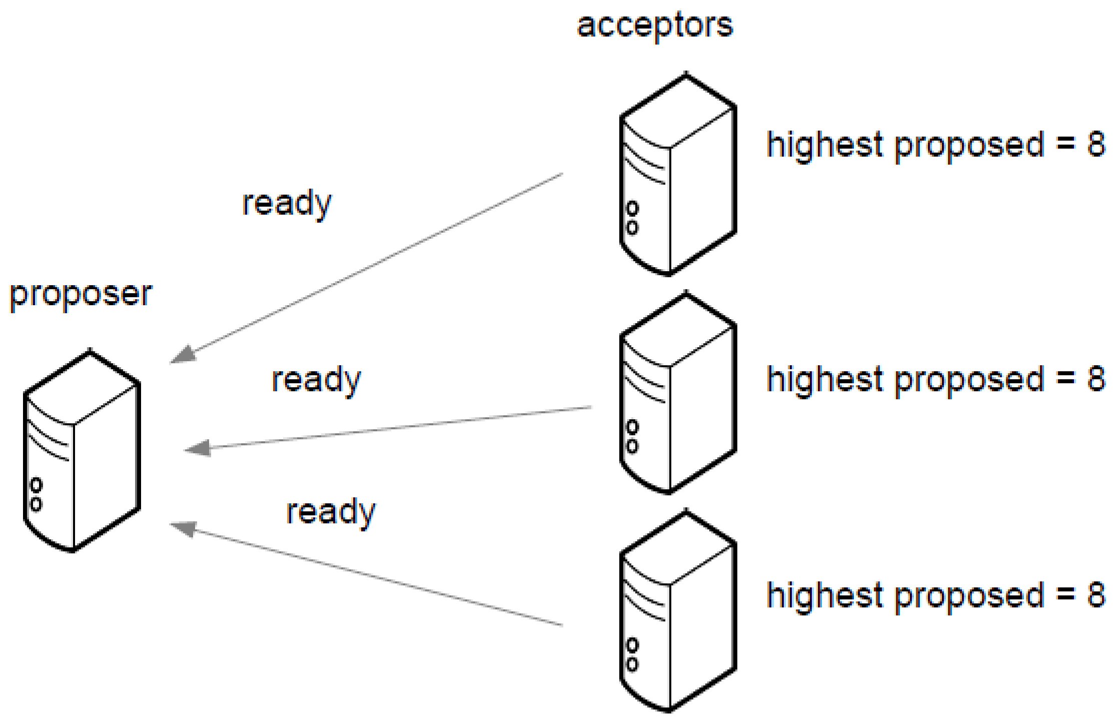
Algorithm (cont)

- If a proposer receives ready responses from a majority of acceptors, it sends an accept message.
- An acceptor that receives an accept message accepts the proposal unless it has responded to a prepare with a number greater than *n*.

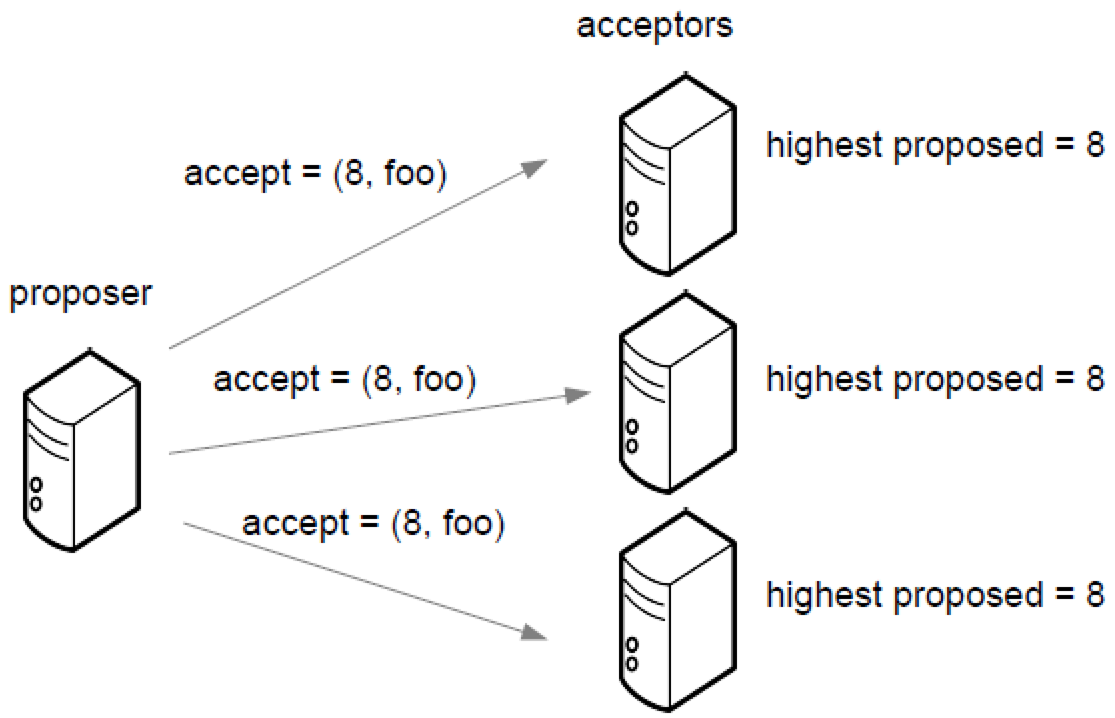
Paxos Example



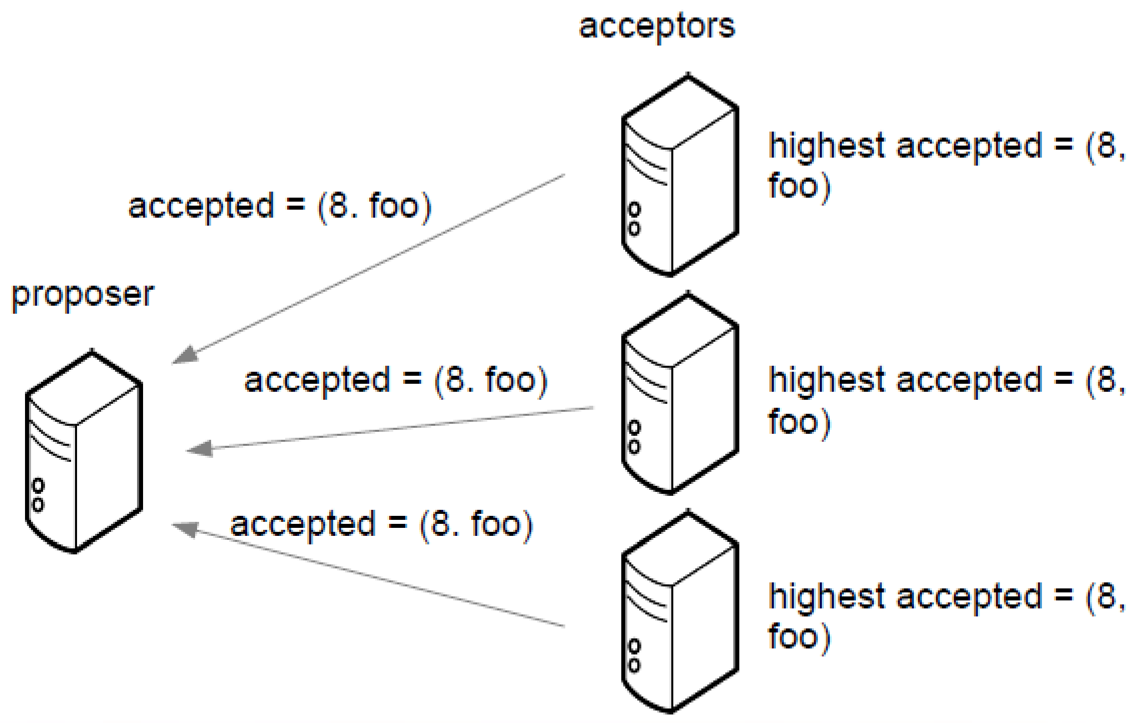
Paxos Example



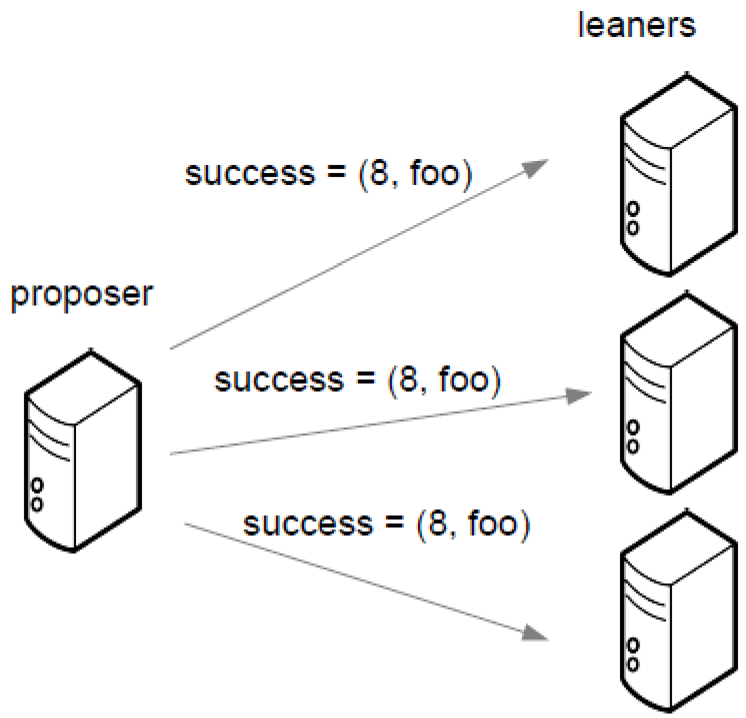
Paxos Example



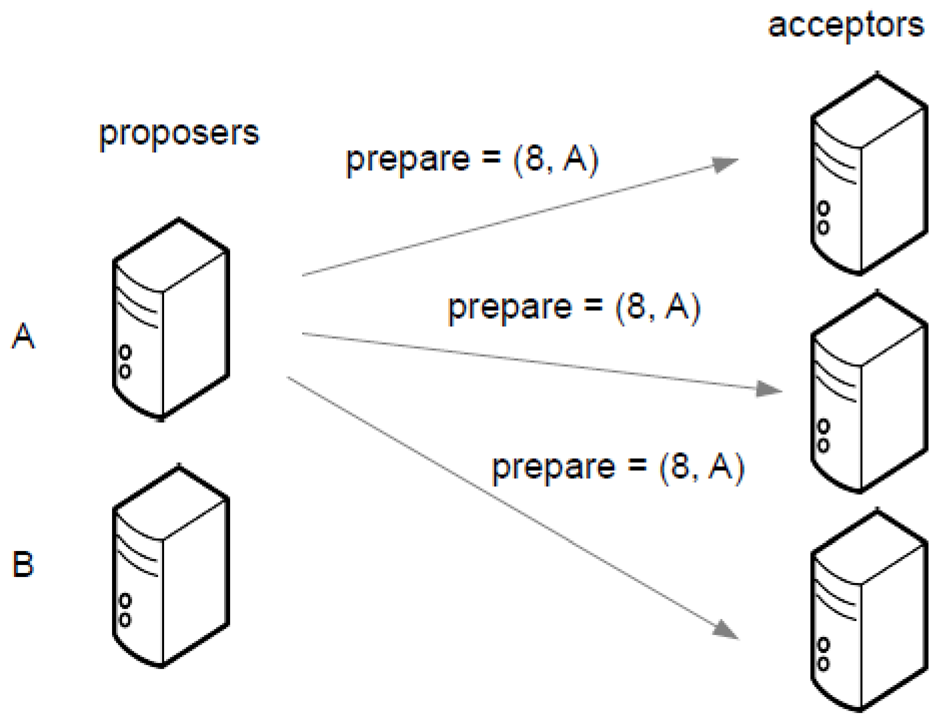
Paxos Example


acceptors
highest accepted = (8, foo)
accepted = (8. foo)
proposer
highest accepted = (8, foo)
accepted = (8. foo)
accepted = (8. foo)
highest accepted = (8, foo)

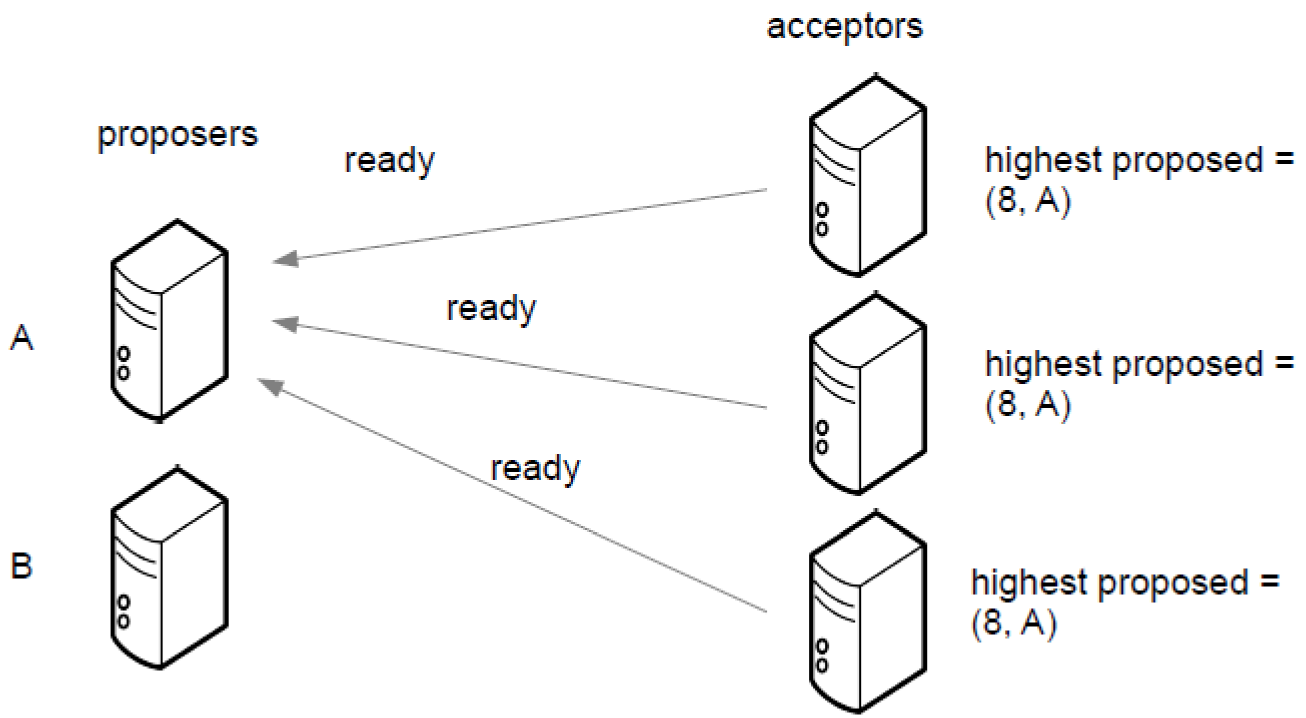
Paxos Example



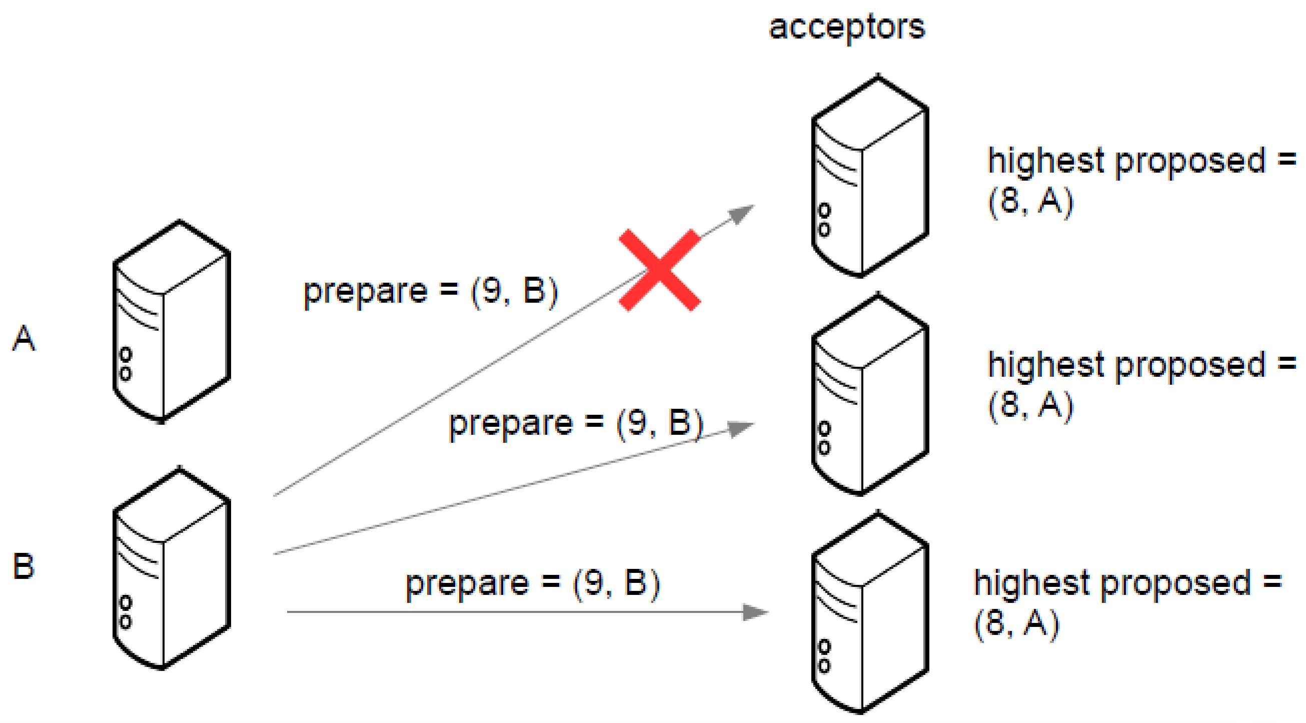
Paxos Example (2)



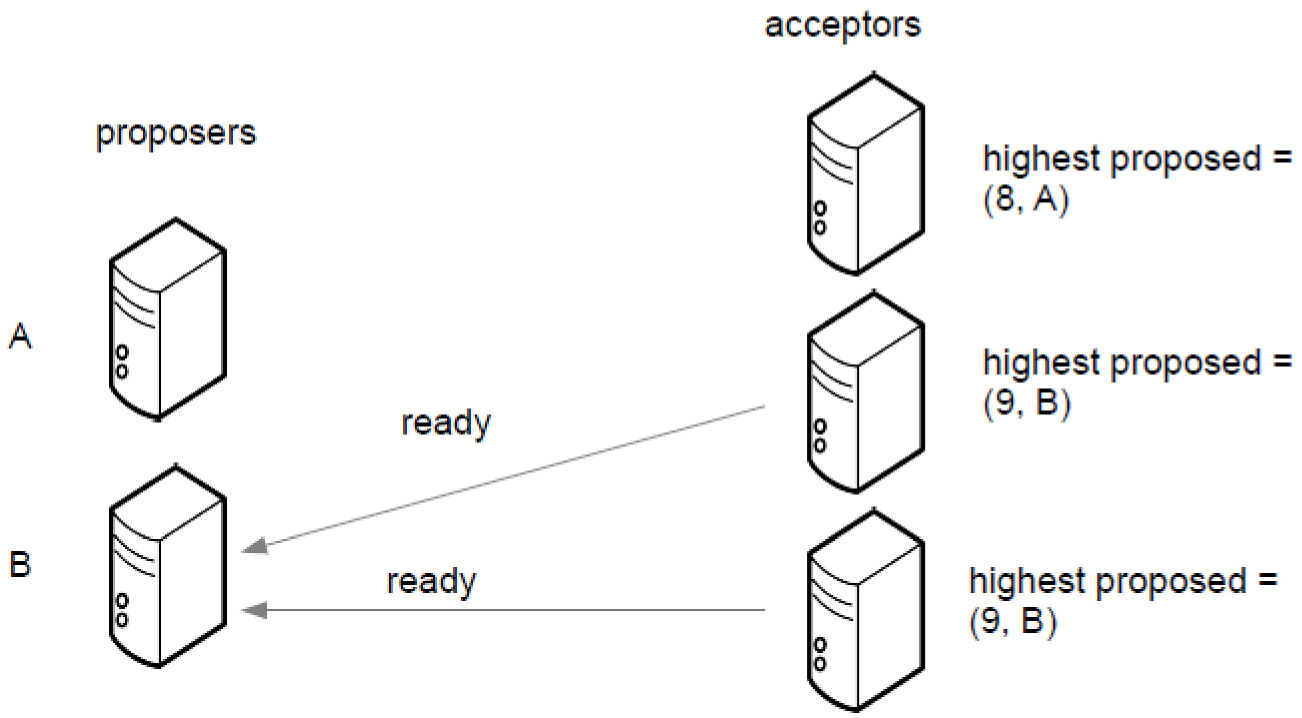
Paxos Example (2)


acceptors
proposers
ready
ready
ready
A
B
highest proposed = (8, A)
highest proposed = (8, A)
highest proposed = (8, A)

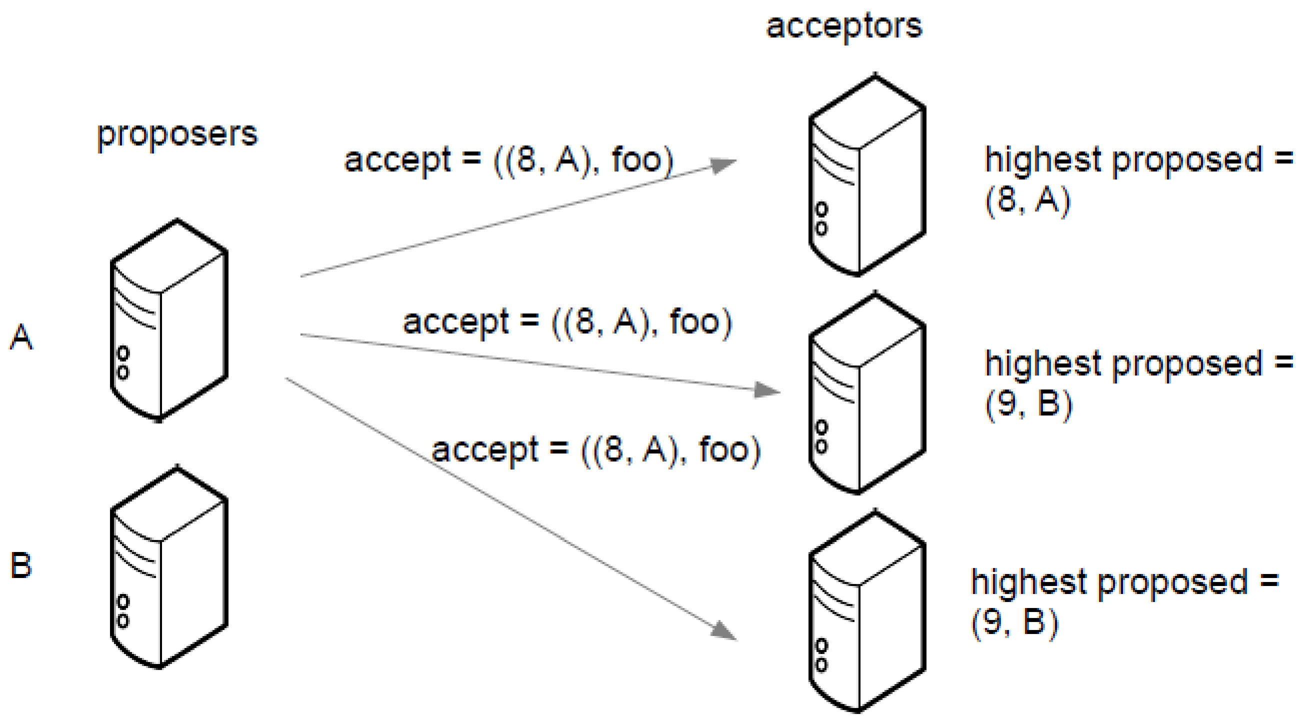
Paxos Example (2)


acceptors
A
B
prepare = (9, B)
prepare = (9, B)
prepare = (9, B)
highest proposed = (8, A)
highest proposed = (8, A)
highest proposed = (8, A)

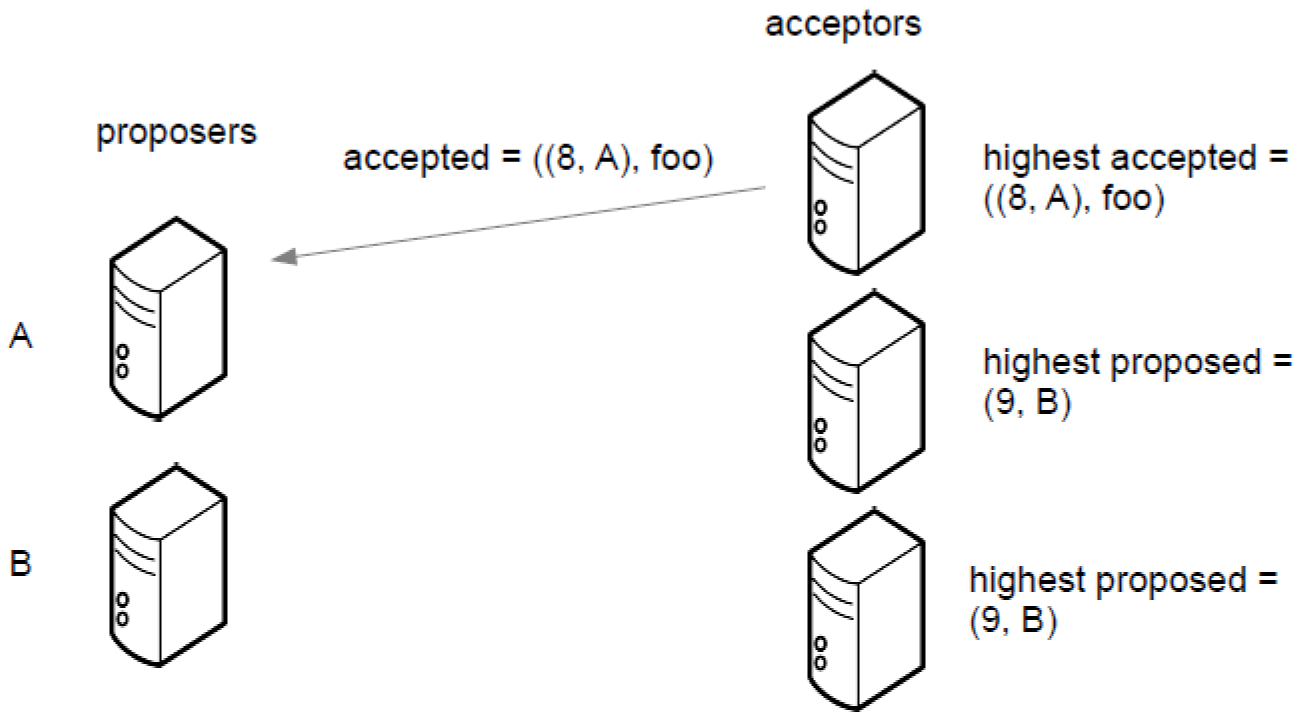
Paxos Example (2)


acceptors
proposers
A
B
ready
ready
highest proposed = (8, A)
highest proposed = (9, B)
highest proposed = (9, B)

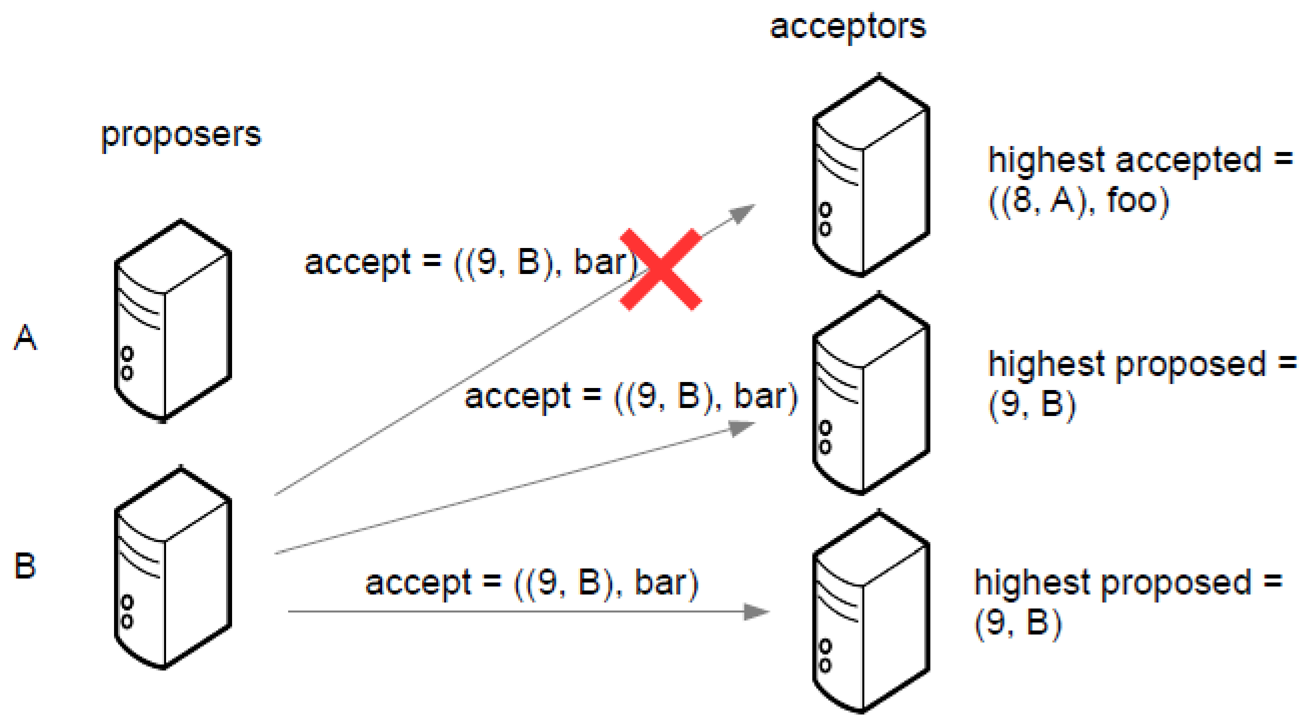
Paxos Example (2)



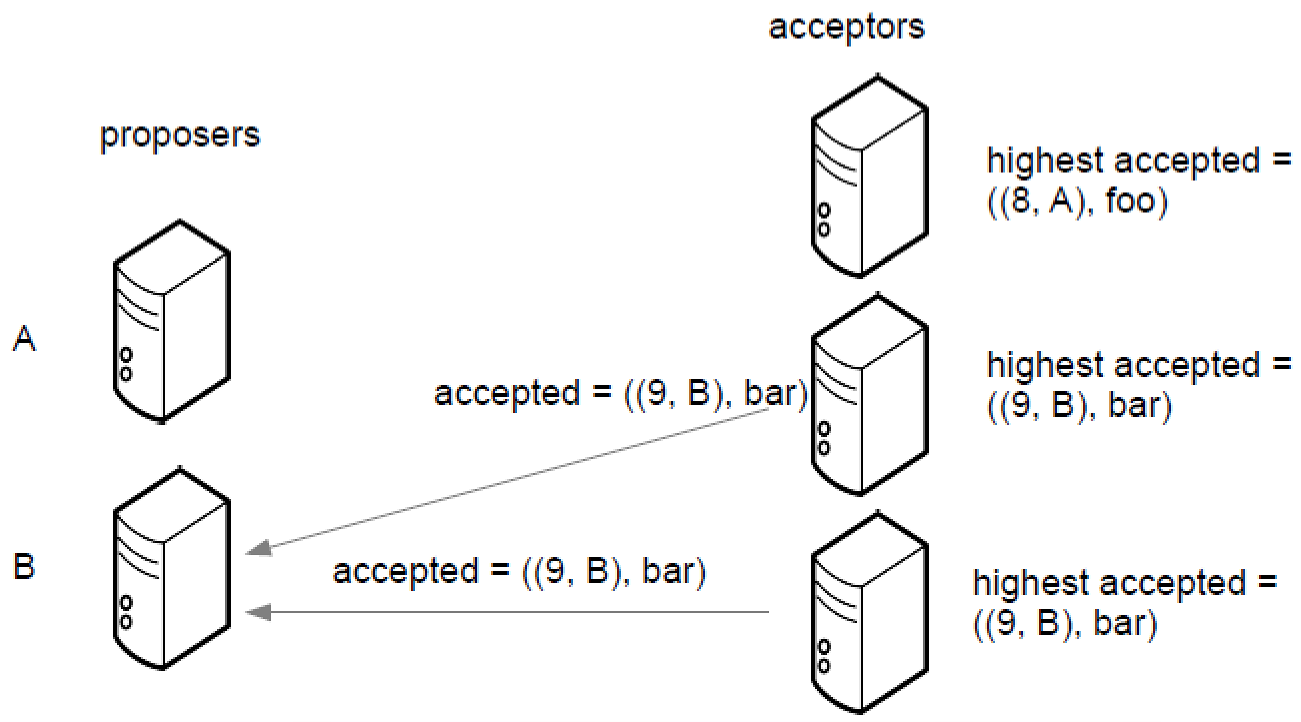
Paxos Example (2)


acceptors
proposers
accepted = ((8, A), foo)
highest accepted = ((8, A), foo)
A
highest proposed = (9, B)
B
highest proposed = (9, B)

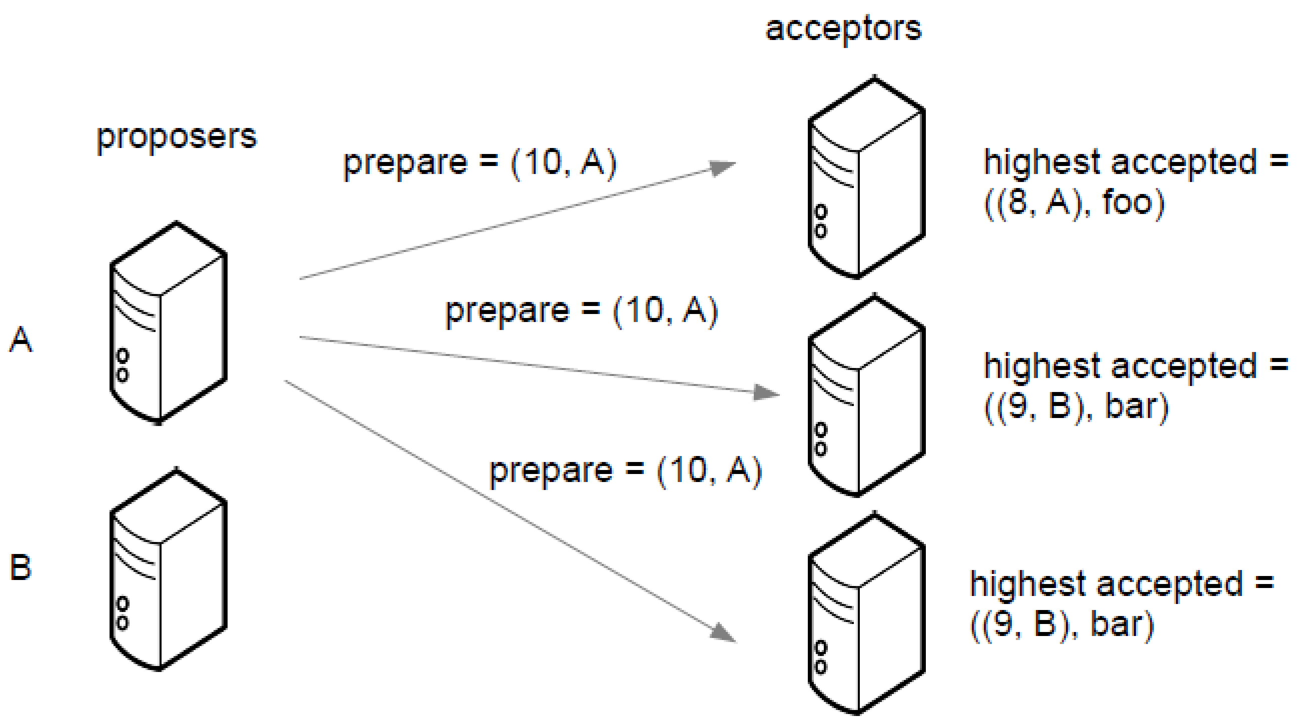
Paxos Example (2)



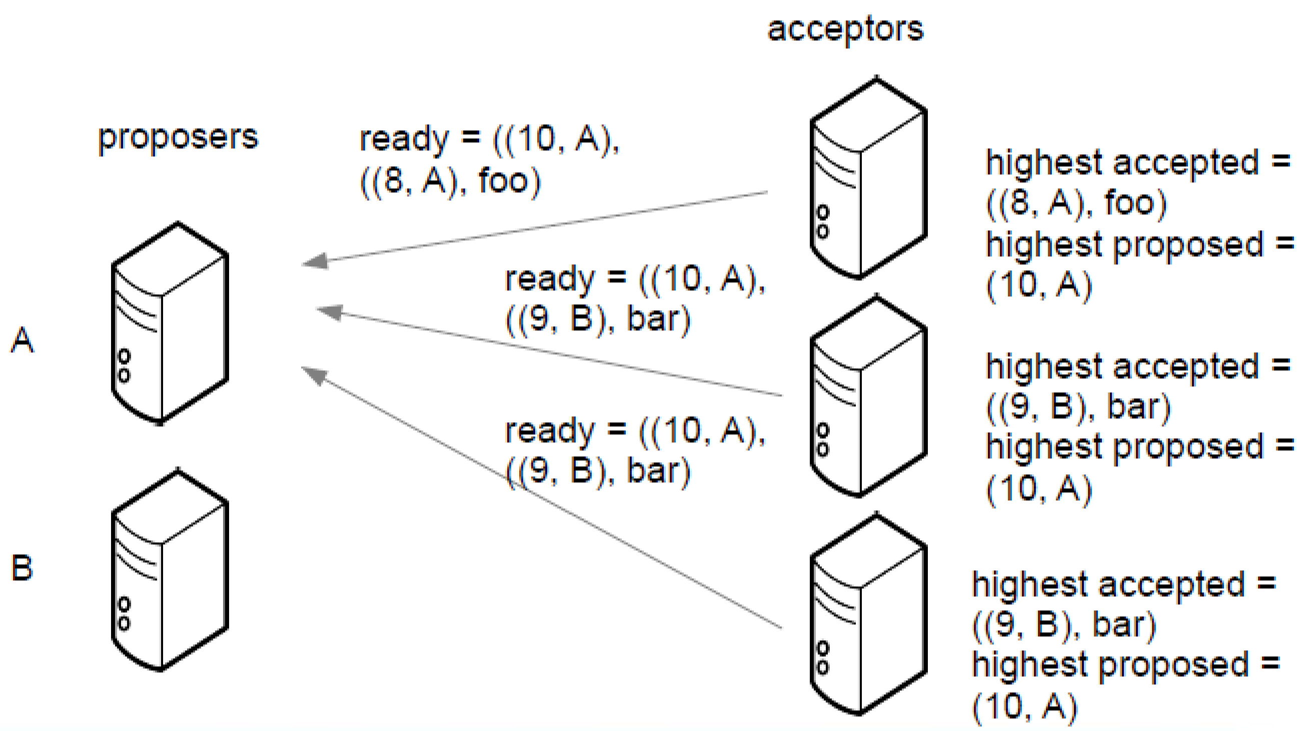
Paxos Example (2)


acceptors
proposers
highest accepted = ((8, A), foo)
A
accepted = ((9, B), bar)
highest accepted = ((9, B), bar)
B
accepted = ((9, B), bar)
highest accepted = ((9, B), bar)

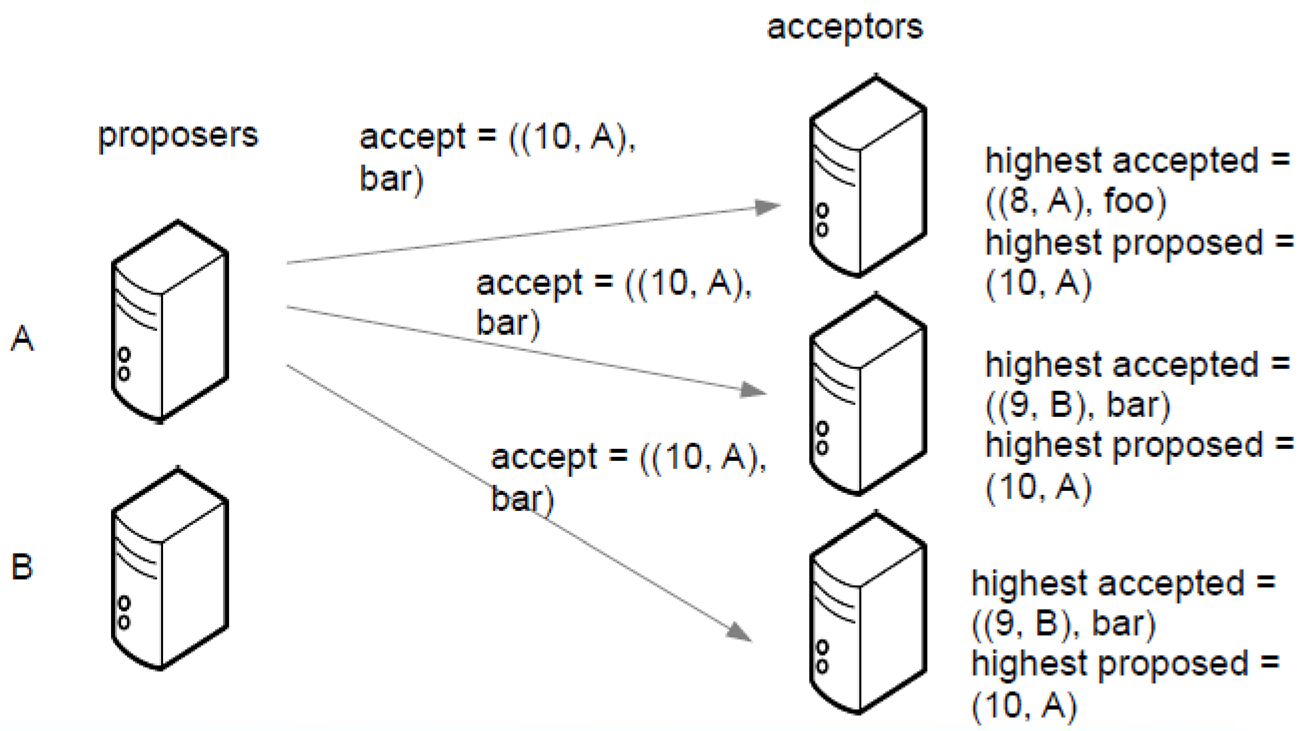
Paxos Example (2)


acceptors
proposers
prepare = (10, A)
highest accepted = ((8, A), foo)
A
prepare = (10, A)
highest accepted = ((9, B), bar)
prepare = (10, A)
B
highest accepted = ((9, B), bar)

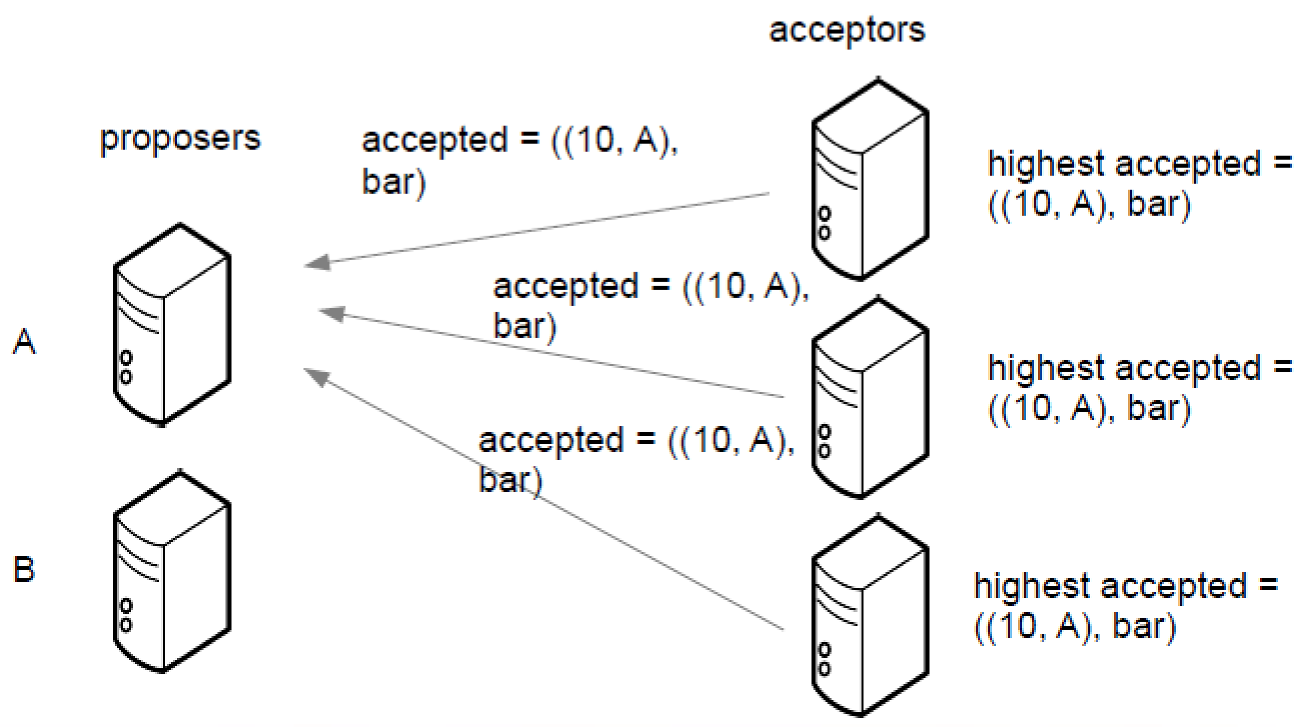
Paxos Example (2)


acceptors
proposers
ready = ((10, A), ((8, A), foo)
ready = ((10, A), ((9, B), bar)
ready = ((10, A), ((9, B), bar)
A
B
highest accepted = ((8, A), foo)
highest proposed = (10, A)
highest accepted = ((9, B), bar)
highest proposed = (10, A)
highest accepted = ((9, B), bar)
highest proposed = (10, A)

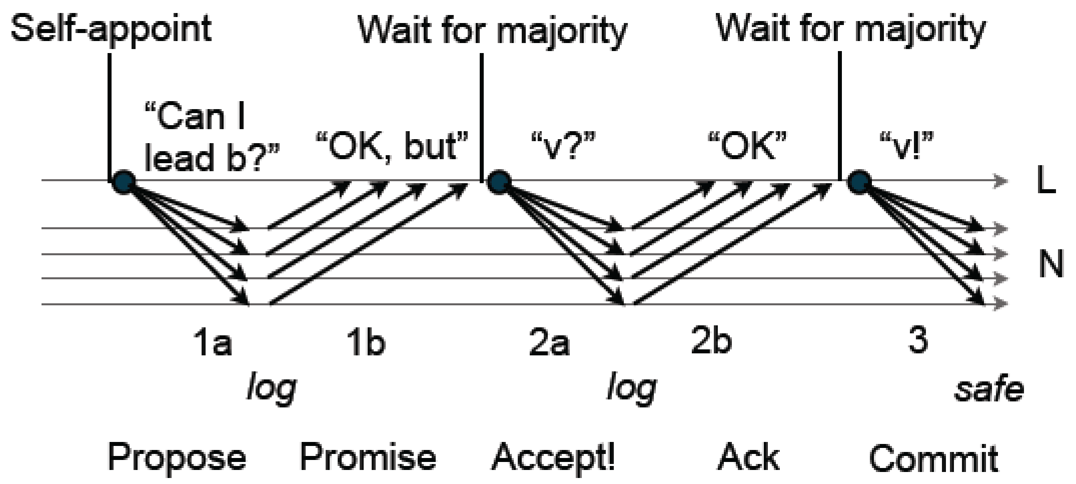
Paxos Example (2)


acceptors
proposers
accept = ((10, A), bar)
accept = ((10, A), bar)
accept = ((10, A), bar)
A
B
highest accepted = ((8, A), foo)
highest proposed = (10, A)
highest accepted = ((9, B), bar)
highest proposed = (10, A)
highest accepted = ((9, B), bar)
highest proposed = (10, A)

Paxos Example (2)


acceptors
proposers
accepted = ((10, A), bar)
highest accepted = ((10, A), bar)
A
accepted = ((10, A), bar)
highest accepted = ((10, A), bar)
accepted = ((10, A), bar)
B
highest accepted = ((10, A), bar)

A Paxos Round



Progress

- You can imagine “dueling” proposers that continually propose higher and higher proposal numbers without any ever being accepted.
- Solution: distinguished proposer.

Failures

- What if an acceptor fails: No problem as long as majority are up.

- What is a proposer fails: If you do not listen for a while, act as a new proposal and run another instance of paxos.