Consensus

Two-Phase Commit and Paxos

COS 418: Distributed Systems
Precept 3

Themis Melissaris and Daniel Suo

Acknowledgements: Kyle Jamieson, Mike Freedman, Irene Zhang

Plan

- Why are distributed systems hard? (1 min)
- Two-phase commit review (10 min)
- Paxos review (15 min)
  - Activity: working out examples (25 min)
- Assignment 1 poll (1 min)

Why are distributed systems hard?

- We tend to think in a single thread
  - But at each step, so many different cases!
- Hard to learn a new mental model and then immediately try to poke holes
  - But that's exactly what we do; try to understand the fault tolerance
- Solution?
  - Take things slowly; trace one 'run' at a time and save each 'what-if' for the next run you reason through
  - More formal methods you can read about

Two-phase commit review

Goals

Multiple servers agree on some action despite failures with the following 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

The actors

- **Client**: the machine requesting some action to be taken
- **Master**: coordinates multiple nodes via the 2PC protocol
- **Node**: machine that takes the action

The phases

- **Prepare**: master asks if all nodes can commit to an action or not

- **Commit**: if all nodes respond yes during the prepare phase, the master tells all nodes to commit

What could go wrong?

- **No reply**: I was expecting a message, but didn't get it
  – Solve with timeouts

- **Reboot**: I crashed and now must recover
  – Solve with write-ahead logs

Note: from the perspective of other nodes, timeouts and crashes look the same, but a node that crashes has to remember some state

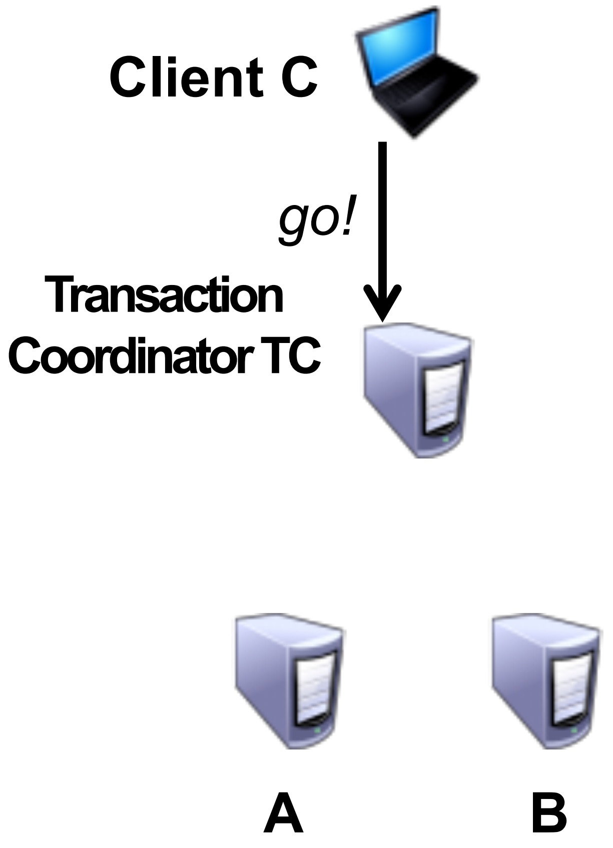
Reasoning about fault tolerance – Take 2

- We’ve identified two kinds of errors and two strategies for resolving them
- However, we still must enumerate all the failure scenarios
- Determine how to use our strategies appropriately to guarantee our properties
- **This is what all the text from Monday was about!**
- Of course, try as we might, there could be error categories we haven’t thought about yet…

Two-phase commit:

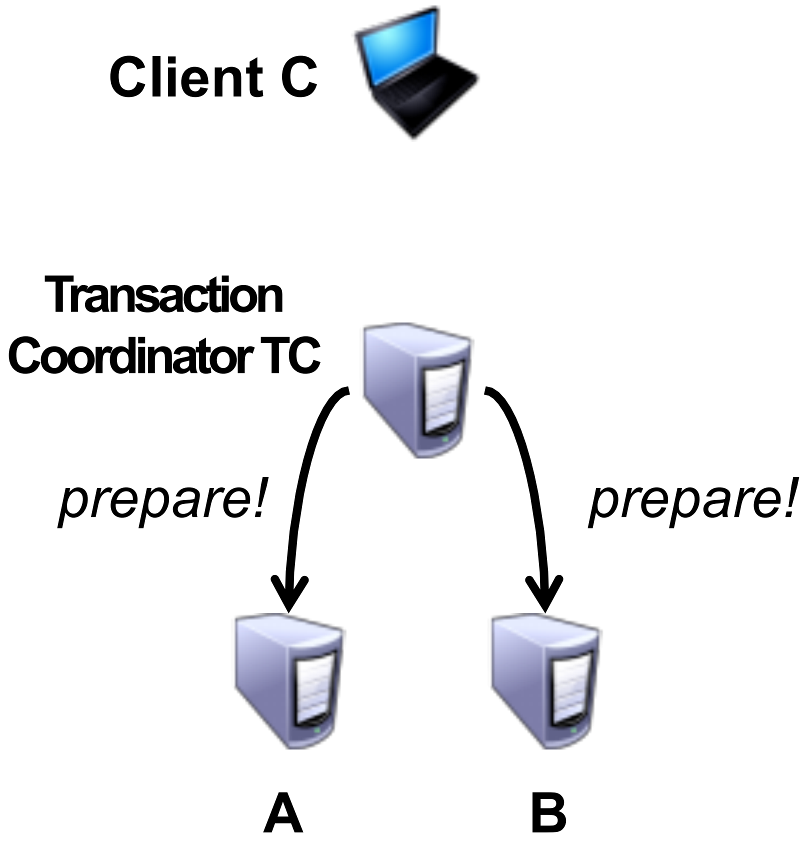
No failure

Client issues request



1. C → TC: *"go!"*

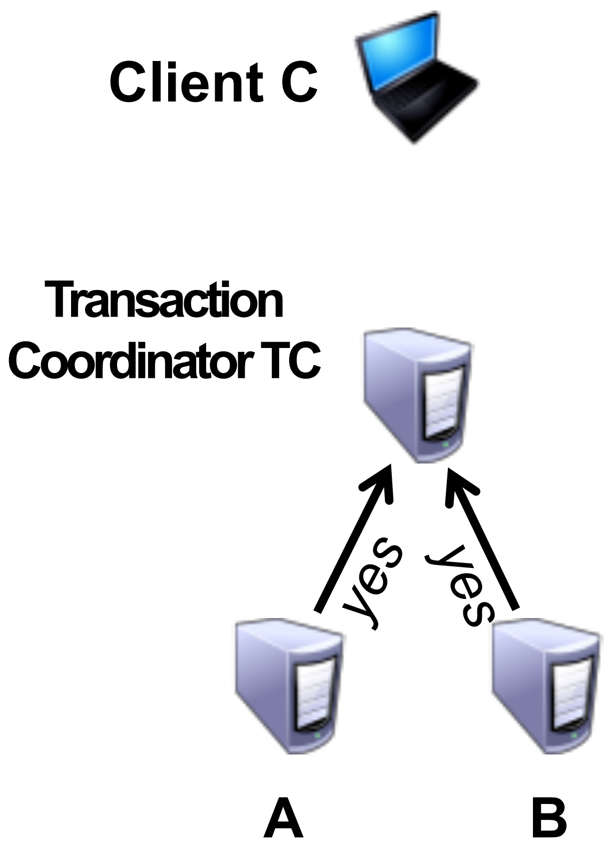
TC tells nodes to prepare



1. **C → TC:** *"go!"*

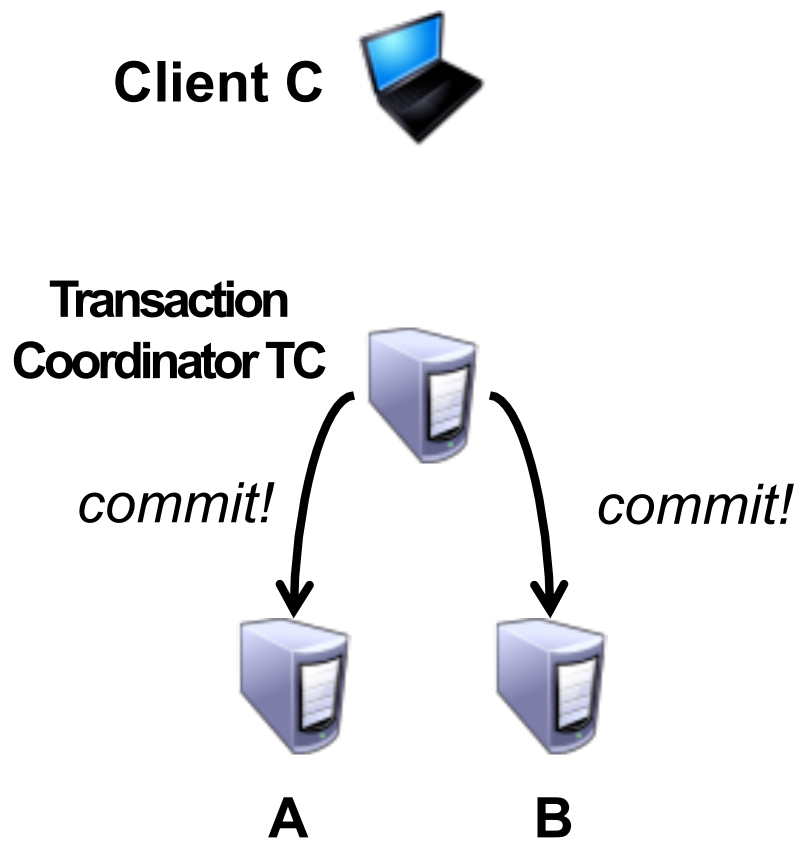
2. **TC → A, B:** *"prepare!"*

Nodes respond that they are ready to commit



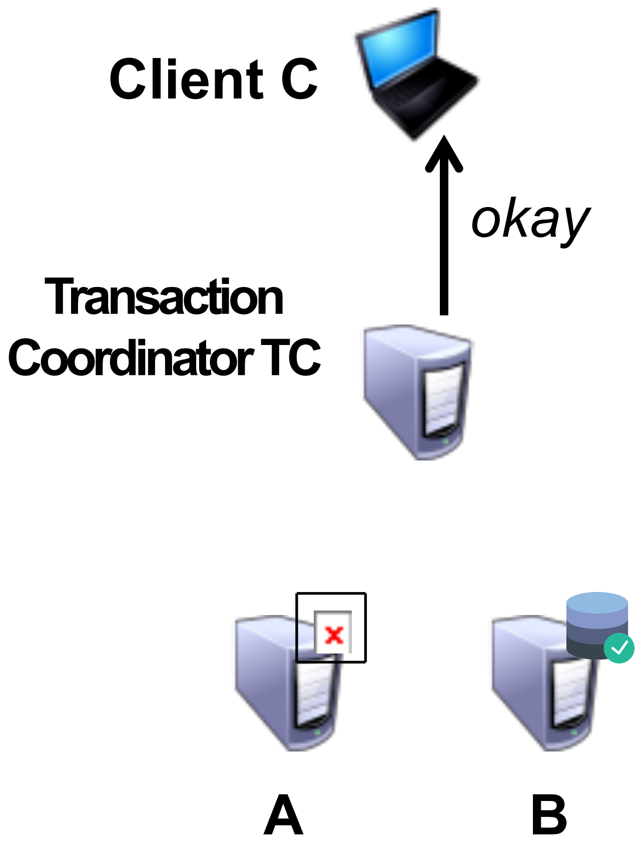
1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → P:** *"yes"* or *"no"*

TC tells nodes to commit



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*

Nodes commit and TC tells client all is well



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"*

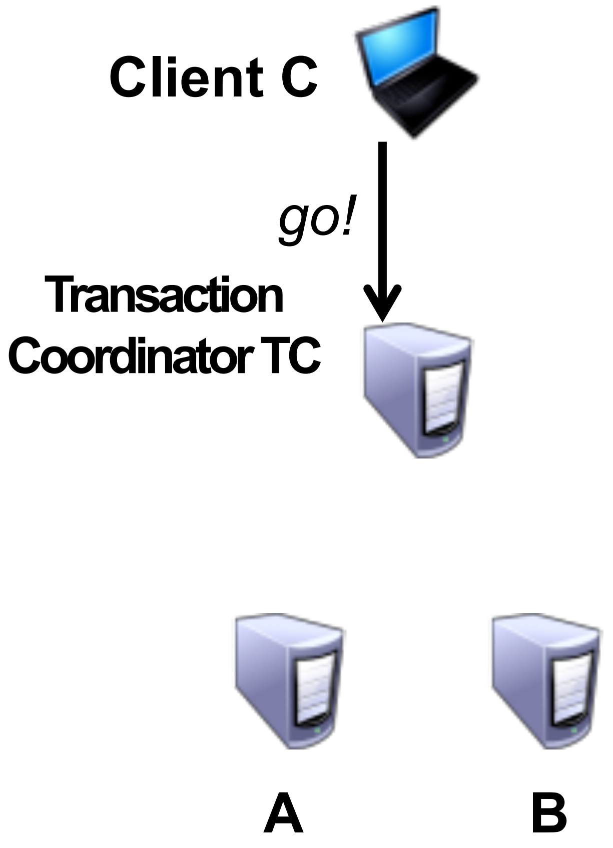
- **A, B** commit on receipt of commit message

Two-phase commit:

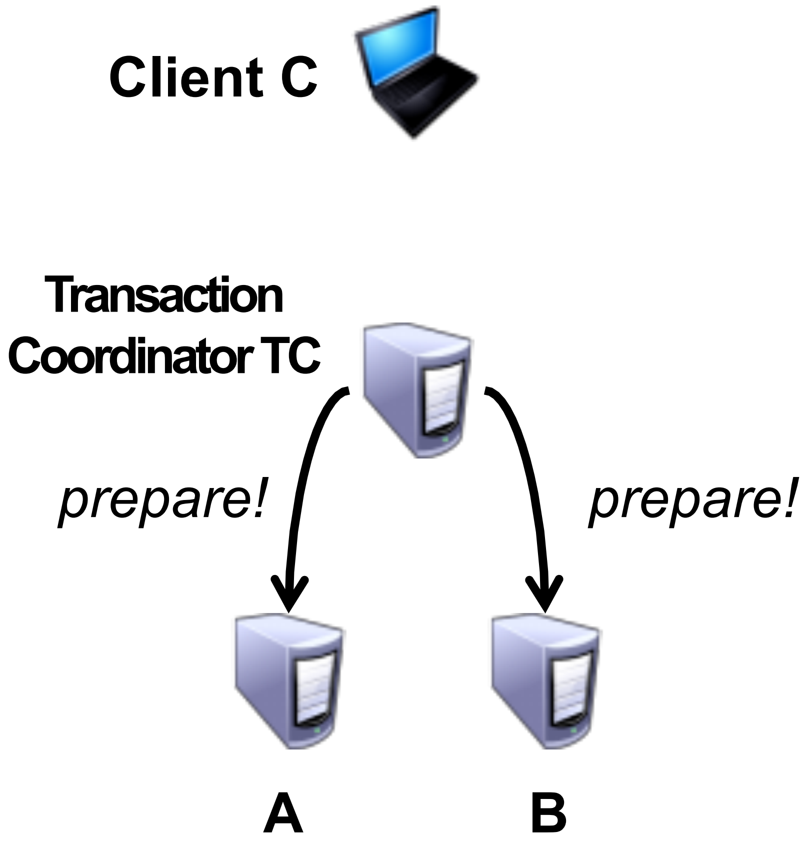
Node crashes before responding

Client issues request

1. **C → TC:** *"go!"*

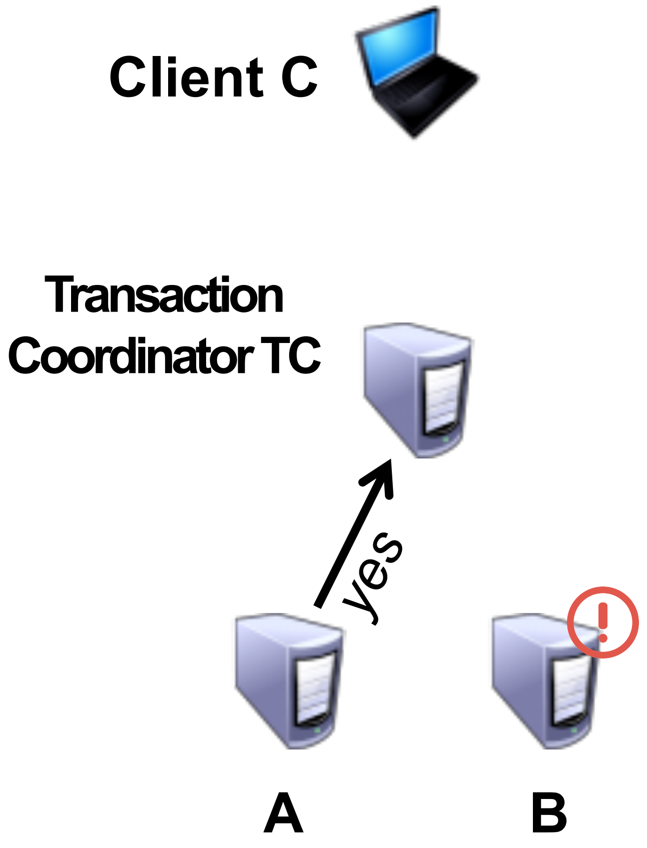


TC tells nodes to prepare



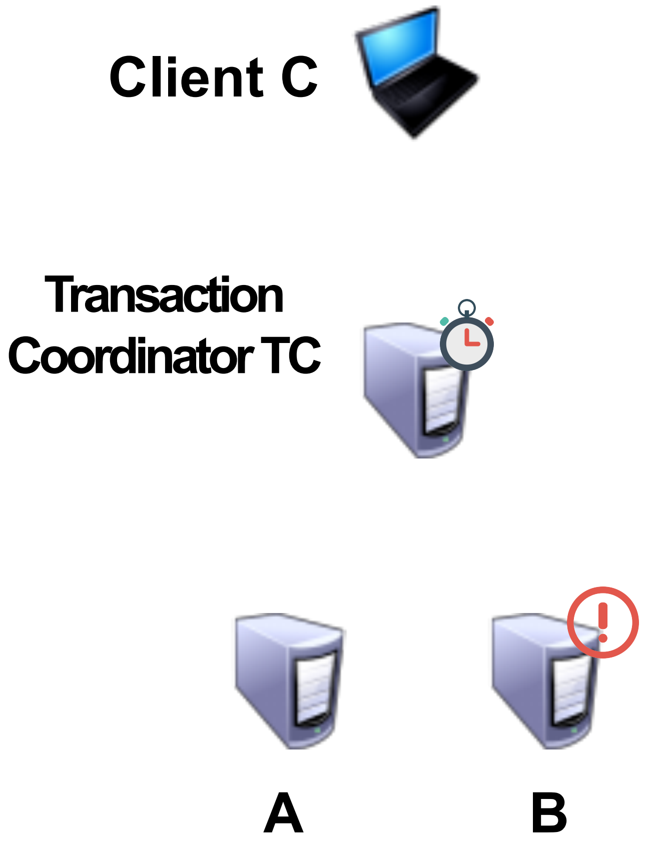
1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*

Node B crashes and only Node A responds (how should we use our two solutions?)



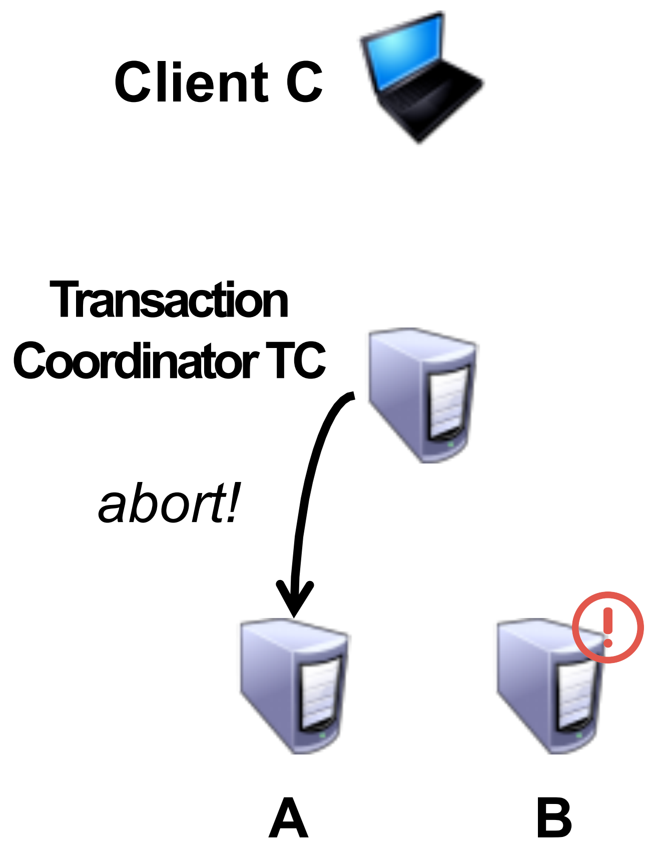
1. C → TC: *"go!"*
2. TC → A, B: *"prepare!"*
3. A → P: *"yes"* or *"no"*, B crashes

TC waits on B until some timeout period



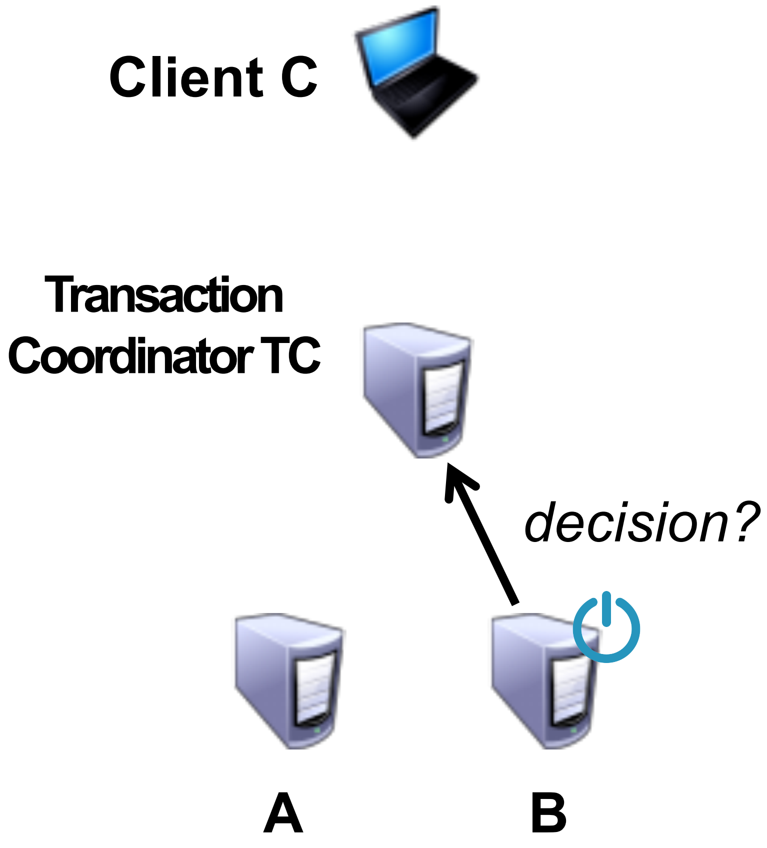
1. **C → TC:** *“go!”*
2. **TC → A, B:** *“prepare!”*
3. **A → P:** *“yes”* or *“no”*, **B crashes**
4. **TC:** times out

TC tells nodes to prepare



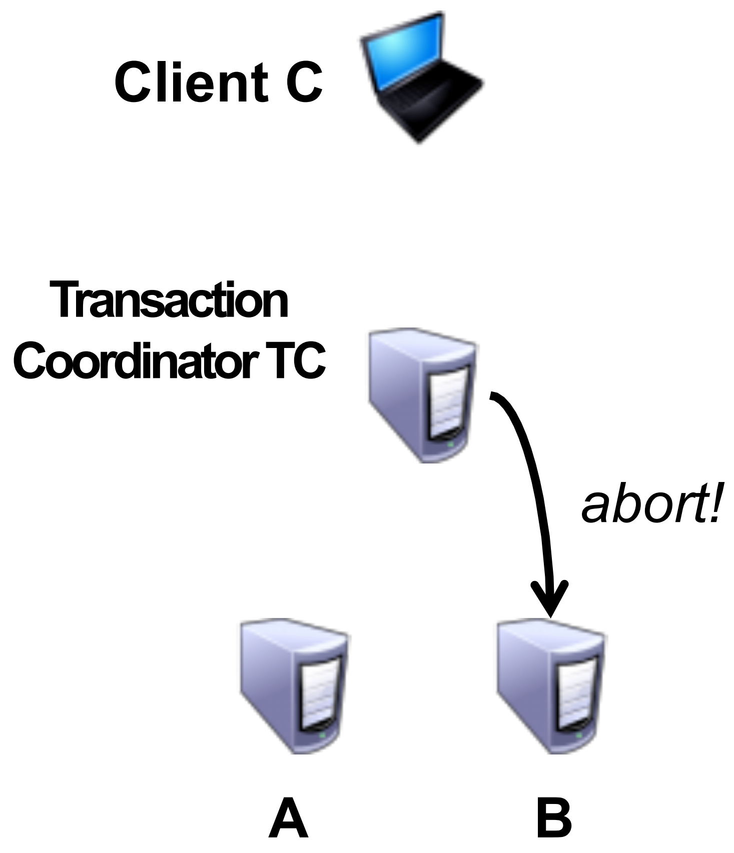
1. **C → TC:** *“go!”*
2. **TC → A, B:** *“prepare!”*
3. **A → P:** *“yes”* or *“no”*, **B crashes**
4. **TC:** times out
5. **TC → A:** *“abort!”*

When B recovers, check with TC for decision (how does B know to check?)



1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A → P:** *"yes"* or *"no"*, **B crashes**
4. **TC:** times out
5. **TC → A:** *"abort!"*
6. ***B* → TC:** *"decision?"*

When B recovers, check with TC for decision



1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A → P:** *"yes"* or *"no"*, **B crashes**
4. **TC:** times out
5. **TC → A:** *"abort!"*
6. ***B* → TC:** *"decision?"*
7. **TC → B:** *"abort!"*

To really understand two-phase commit, you should list the failure cases and convince yourself of what should happen in each

Paxos review

Two ways to learn Paxos

1. Give the algorithm and examine how it satisfies stated properties (deductive)

2. Begin with properties, build up what an algorithm would have to do to satisfy properties (inductive)

- The former is how we treated 2PC and the latter is a more formal method

- Which path?
  - Red pill of painful truth (formal method)
  - Blue pill of blissful ignorance (give the algorithm)?

Common material

Properties

Goal: a collection of processes that can propose values would like to reach a consensus value.

1. **Safety**
   - Only a single value is chosen
   - Only a proposed value can be chosen
   - Only a chosen value is propagated to learners
2. **Liveness**
   - If fewer than half of processes fail, some value eventually chosen
   - If a value is chosen, a process eventually learns it

Some administrative stuff

- Three roles: proposers, acceptors, learners
- A process can have more than one role, but don’t care about that (why?)
- Processes can fail and restart, but have to remember some state
- Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but are not corrupted (non-Byzantine)

Red pill

Strategy

1. List our safety and liveness properties

2. Start with a simple algorithm that satisfies the safety properties and iterate to make more fault tolerant

3. Examine what happens with the final algorithm in different failure scenarios

4. Compare and contrast with two-phase commit

Choosing a value: v1

- Use **one acceptor** – done!
  - Proposers send values to the acceptor and the first value the acceptor receives is the chosen value
  - Learners then learn the value from the acceptor
  - We satisfy all our safety properties (review)

- But if our one acceptor fails permanently, we can't make any more progress

Choosing a value: v2

- Proposers send proposals to **multiple acceptors**
  - Now we aren't dependent on a single acceptor
  - Use a majority to guarantee only one value is accepted
  - If acceptors can only accept one value, there can only be one majority (why?)
  - Learners learn from an acceptor in the majority
  - Once again, safety properties satisfied

- Now we need to know who is in the acceptor group

Invariant P1

- Absent network or node failure, we want to be able to choose a value even if we only have proposer who proposes only one value
- **P1**: An acceptor must accept the first proposal that it receives (why?)
- **Problem**: we might not have a majority! (why?)
- **Solution**: let acceptors accept multiple values (how does this solve the problem?)

Choosing a value: v3

- Proposers send proposals to multiple acceptors
- For book-keeping, let each proposal have a proposal number *n*
  - For now, assume there is some protocol among machines to do this
- Acceptors can differentiate among and **accept multiple proposals**
- **Problem**: we don't satisfy our safety requirements anymore! (why?)

Invariant P2

- How do we make sure we don't have different majorities choosing different values?

- **P2**: Want to make sure once we choose a value, we stick to it!
  - More formally, if our acceptors have accepted value *v* at proposal number *m*, then all subsequent proposals *n*, where $n > m$ propose *v*.

Recap

- v1: Single acceptor
- v2: Multiple acceptors
- v3: Acceptors accept multiple proposals
- Invariant P2 resolves our problem with v3!!! (softball question: why?)
- Of course, just because P2 magically makes v3 work, how do we make sure P2 holds?

Making P2 stronger

- **P2^a**: Once we choose a value, every proposal $n > m$ **accepted by any acceptor** will have the same value (how does this guarantee P2?)

- **P2^b**: Once we choose a value, every proposal $n > m$ **proposed by any proposer** will have the same value (how does this guarantee P2^a?)

- Why are we doing this???

- Remolding P2 so we can actually implement it

Making P2 stronger

- **P2^c**: for proposal *n* with value *v*, there is a majority of acceptors *S* where either
  - No one has accepted a proposal with number < *n*
  - The proposal with highest number < *n* accepted thus far has value *v*

- Sketch* of proof by induction:
  - Suppose proposal *m* < *n* were accepted with value *v*
  - Assume proposal *n – 1* has value *v*
  - Show proposal *n* has value *v*

* Details of the inductive proof in *Paxos Made Simple* by Leslie Lamport. We've simplified the logic, but recommend reading the original paper!

What does P2^c give us?

- To ensure P2^c, if a proposer wants to make proposal *n*
  - Must check for earlier proposal number accepted by majority; take the one with highest number (why?)
  - If none exist, go ahead and propose new value *v*
- Easy to check for proposals already accepted (just ask), but what about proposals about to be accepted?
- Too hard! Force acceptors to make a promise
- P2^c gives us an easy way to implement an algorithm

Choosing a value: v4 proposer

- Everything from v3 (multiple acceptors, proposal numbers, accept multiple proposals)

- **Prepare request**: choose new proposal number *n* and ask acceptors
  - To promise never to accept a proposal with lower proposal number again (why?)
  - To return the proposal with the highest number less than *n* that it has accepted, if any (why?)

Choosing a value: v4 proposer

- **Accept request**: if proposer receives responses from a majority *S*, issue proposal *n* with value *v* where *v* is
  - The value of the accepted proposal with the highest number from among responses (why does this work?)
  - Any value selected by proposer if no responder has accepted an earlier proposal

Choosing a value: v4 acceptor

- Responding to **prepare request**
  - Promise not to accept any proposal with number < *n*
  - Proposal with highest number accepted, if any
- Responding to **accept request**
  - Accept if and only if we haven’t promised not to
- We can always ignore requests without compromising safety (why? What about liveness?)
- How did the condition for accepting differ between v3 and v4?

Walking back: why does this work?

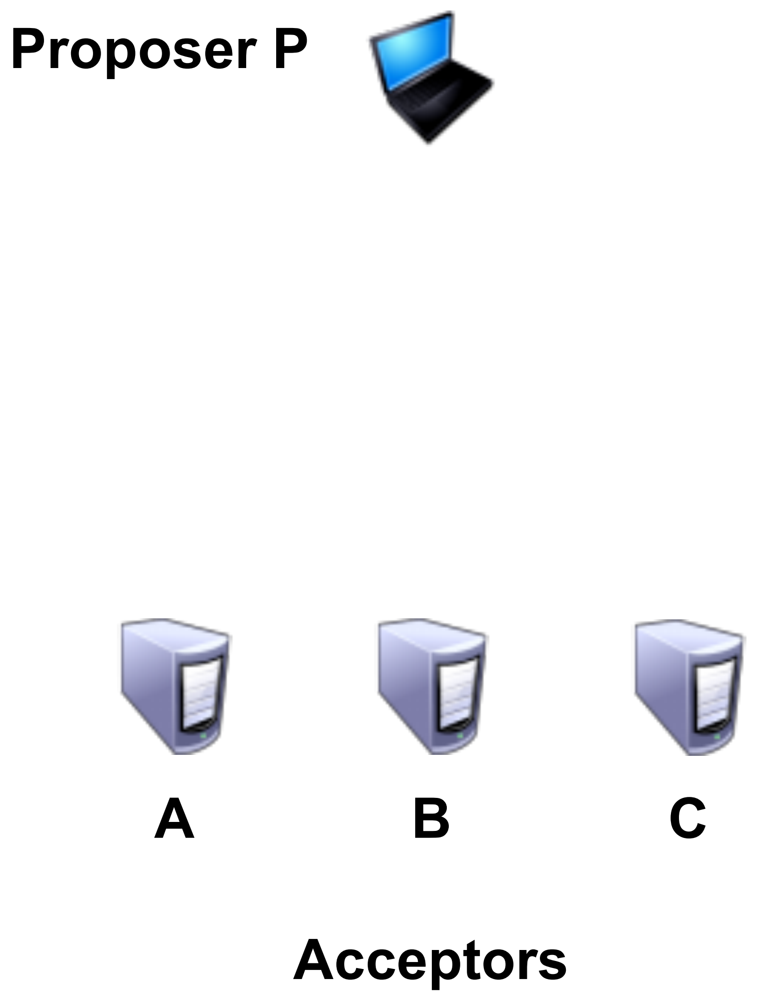
- Our algorithm guarantees $P2^c$ (literally converts its requirements to an algorithm)
- $P2^c$ guarantees P2, which guarantees we stick to a value once we've chosen it
- P2 solves our problem from v3 (how to reach consensus if acceptors accept multiple values)
- v3 solved our liveness issues with v2 and v1

Recap

- **v1**: Single acceptor
- **v2**: Multiple acceptors
- **v3**: Acceptors accept multiple proposals
- **v4**: Proposers only propose new value if there hasn’t already been an accepted value
- Note how the condition for accepting differs between v3 and v4!

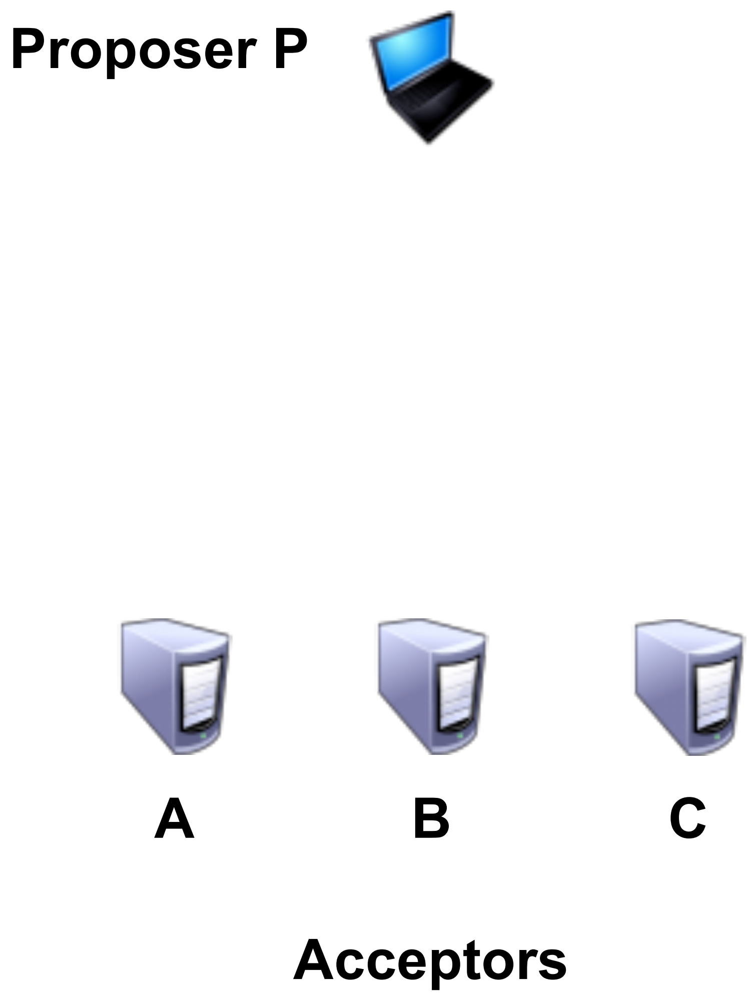
Blue pill

Choose proposal number



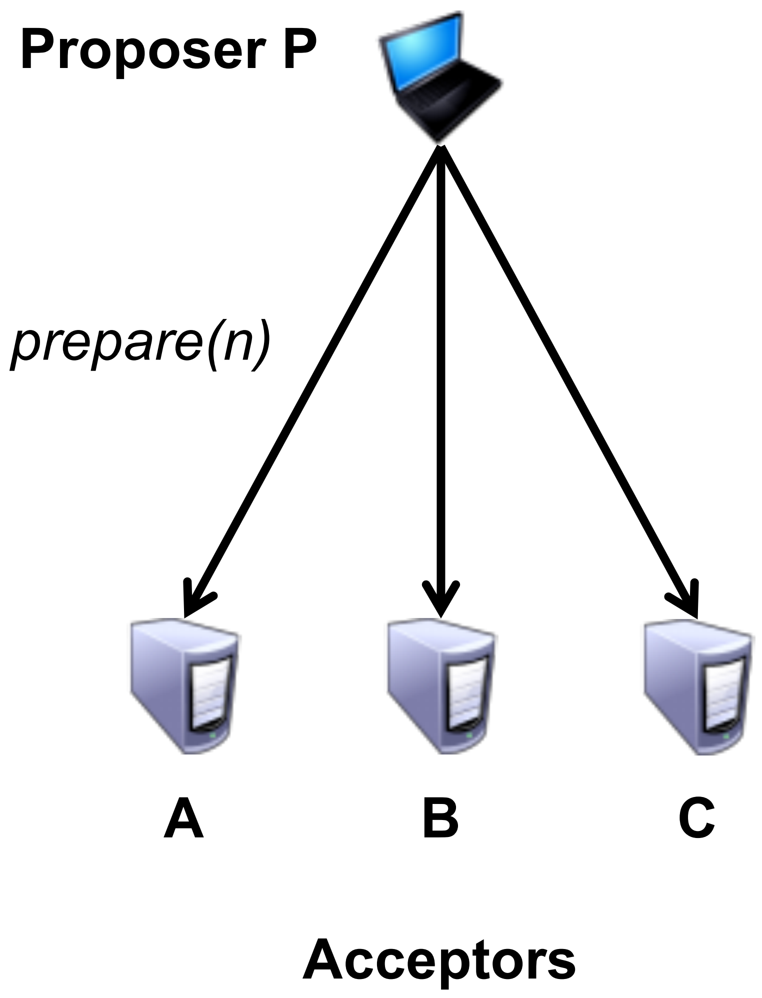
1. **P** chooses proposal number *n*

Acceptors have some initial state



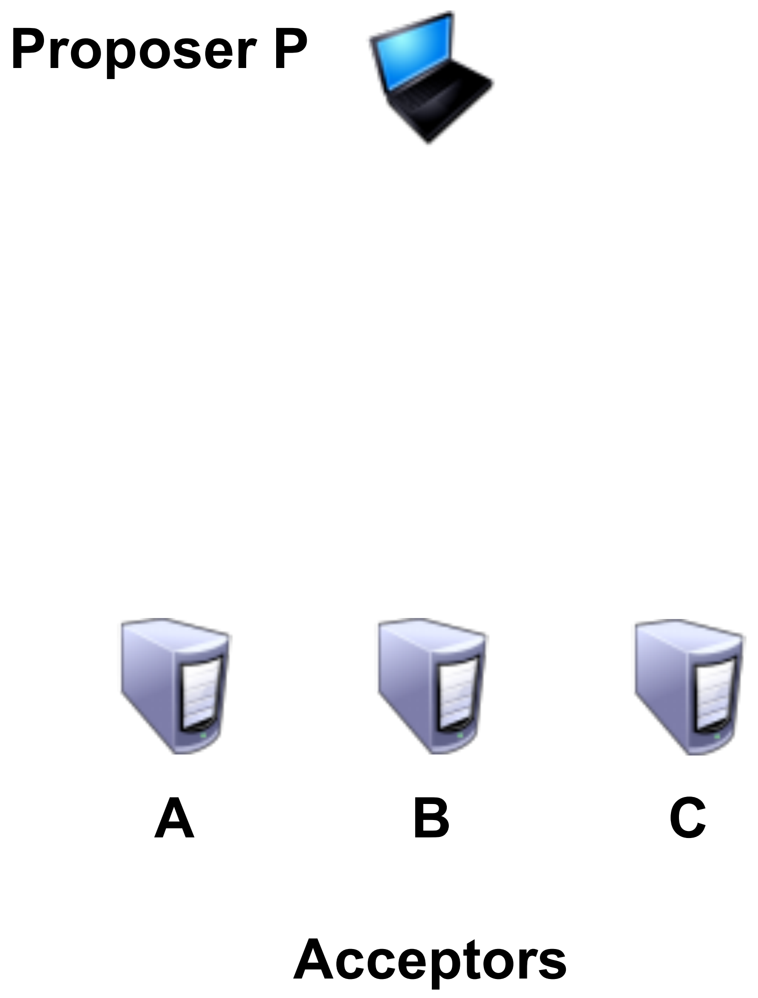
1. **P** chooses proposal number *n*
2. **A, B, C:**
   - **minProposalNum:** some proposal number below which I will never accept. Initialized to 0.
   - **acceptedProposal**: the proposal tuple (n,v) with highest n that I have accepted. Initialize to null

Send prepare message to all acceptors



1. **P** chooses proposal number *n*
2. **P → A, B, C:** *"prepare(n)"*

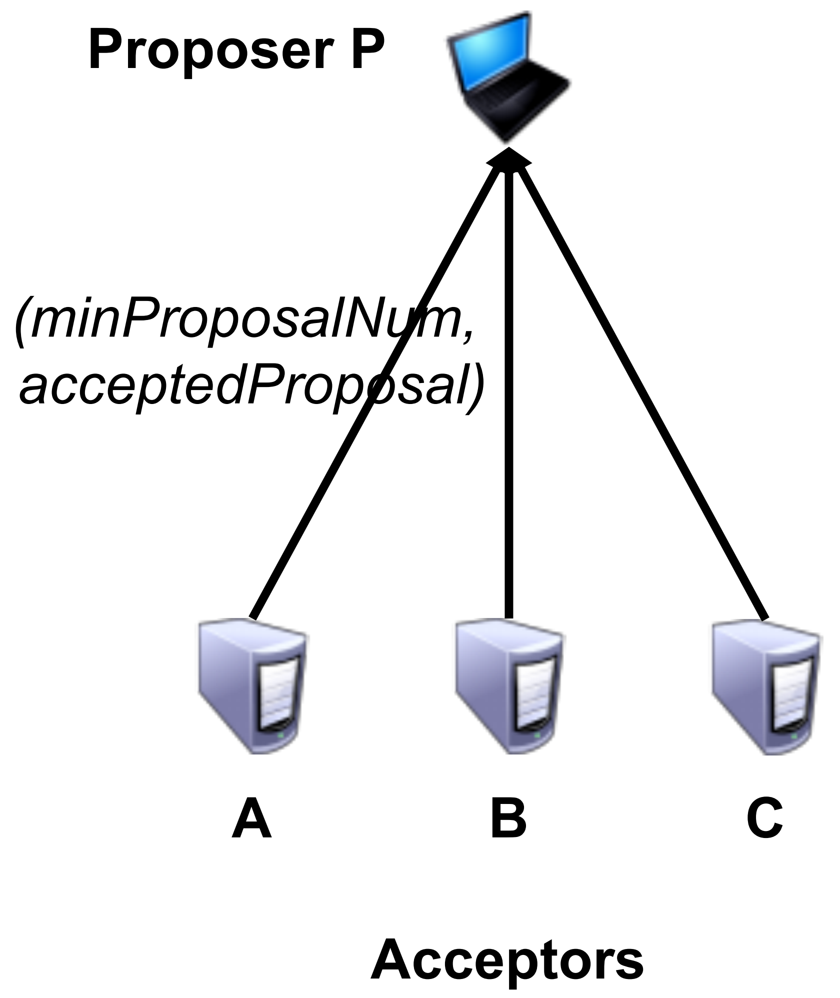
Nodes update state if needed



1. **P** chooses proposal number *n*
2. **P → A, B, C:** *"prepare(n)"*
3. **A, B, C:** if this is the highest proposal number I've seen, remember it

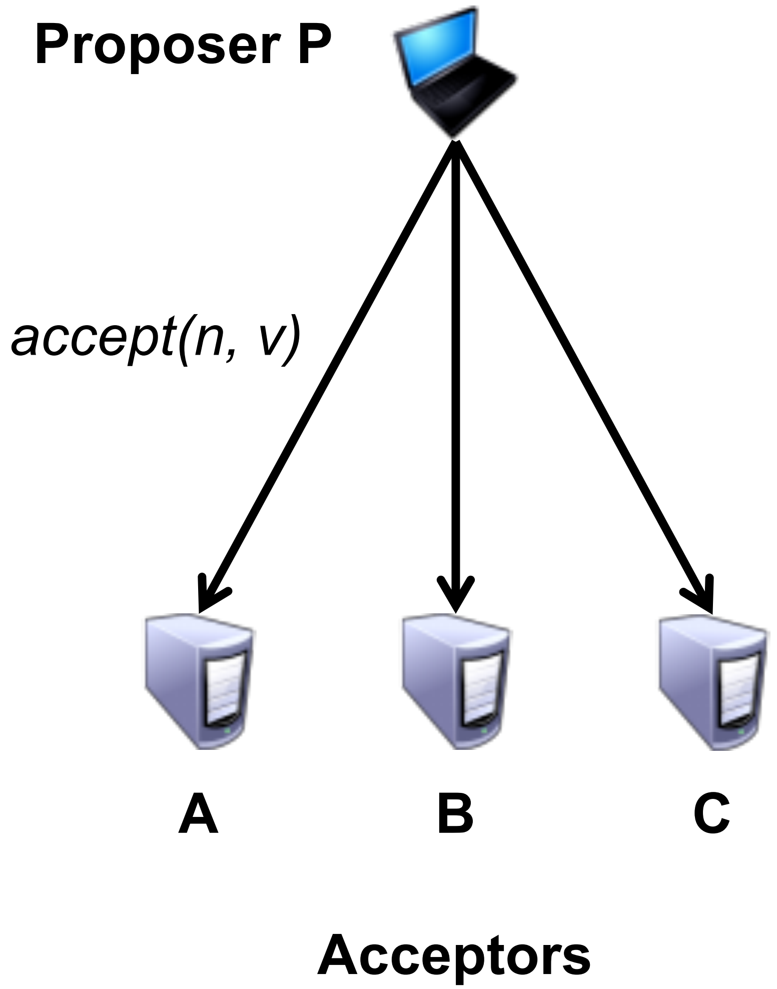
```
if n > minProposalNum then
minProposalNum = n
```

Respond to prepare message



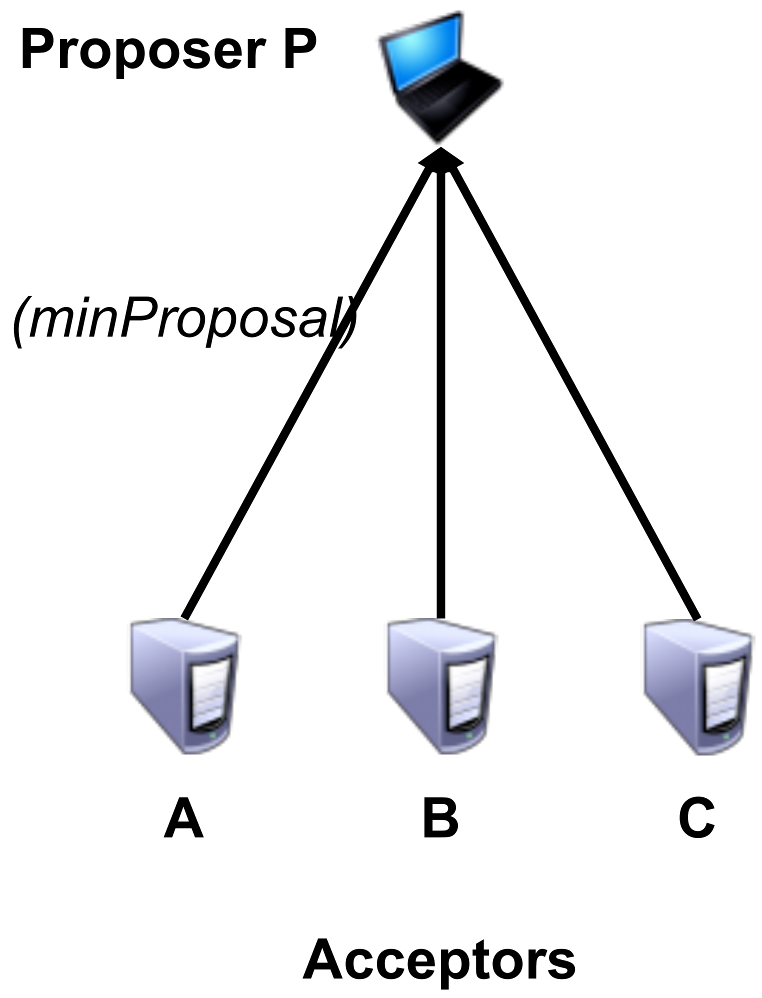
1. **P** chooses proposal number *n*
2. **P → A, B, C:** *"prepare(n)"*
3. **A, B, C:** if this is the highest proposal number I've seen, remember it
4. **A, B, C → P:**
   – promise I won't accept any proposal less than minProposalNum
   – return the highest proposal number accepted and accepted value, if any (none so far)

If P receives response from majority, send accept request



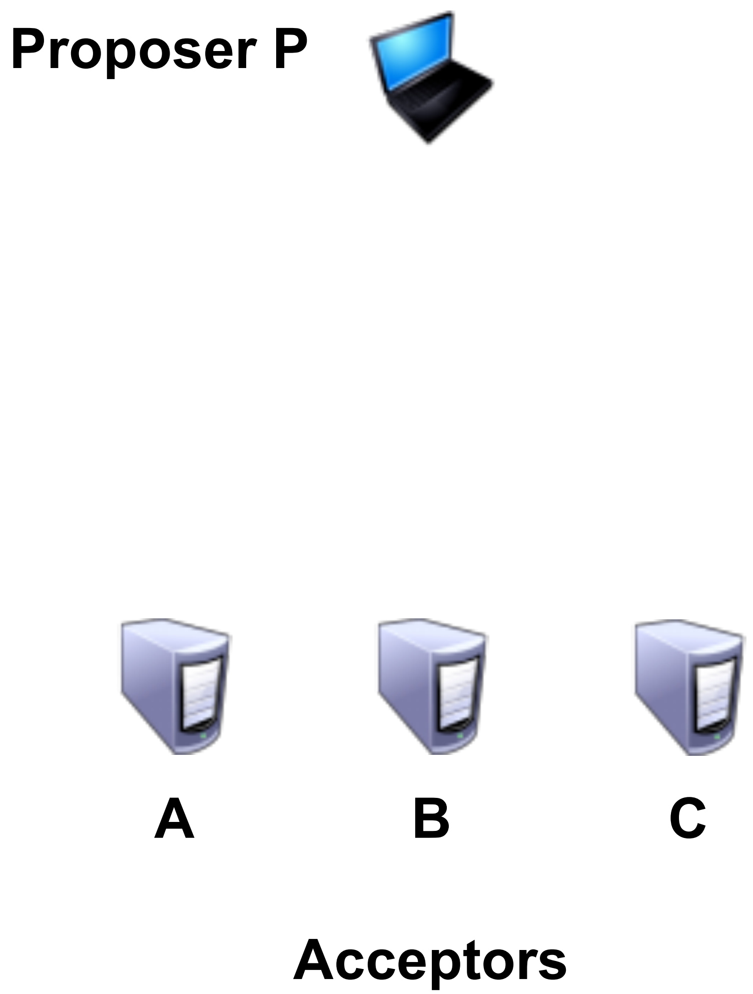
5. **P → A, B, C:** *"accept(n, v)"*, where *v* is the value from the highest accepted proposal number I've seen

Respond to accept message



5. **P → A, B, C:** *"accept(n, v)"*, where *v* is the value from the highest accepted proposal number I've seen
6. **A, B, C → P:**
   - If n is greater than minProposalNum, accept the value and update acceptedProposal
   - Return the proposal number accepted

Choosing a value



7. If we received any rejection (i.e., minProposal returned > n), **repeat protocol**
8. Otherwise, **choose value**

Common material

What about learning values?

- This part of the protocol is important, but not really that hard

- Trivially: each acceptor sends accepted proposal to each learner (why might this be inefficient?)

What about failures?

- **No reply**: I was expecting a message, but didn’t get it
  - Does it matter? When?

- **Reboot**: I crashed and now must recover
  - Solve with write-ahead logs

Paxos vs. 2PC

- 2PC can be viewed as a special case of Paxos
  - Transaction coordinator is the only proposer
  - Nodes are all acceptors
  - All acceptors must accept to choose value

- Paxos improves on 2PC
  - Much less likely to block than 2PC
  - But still can!

Paxos by example

(video walkthrough here)

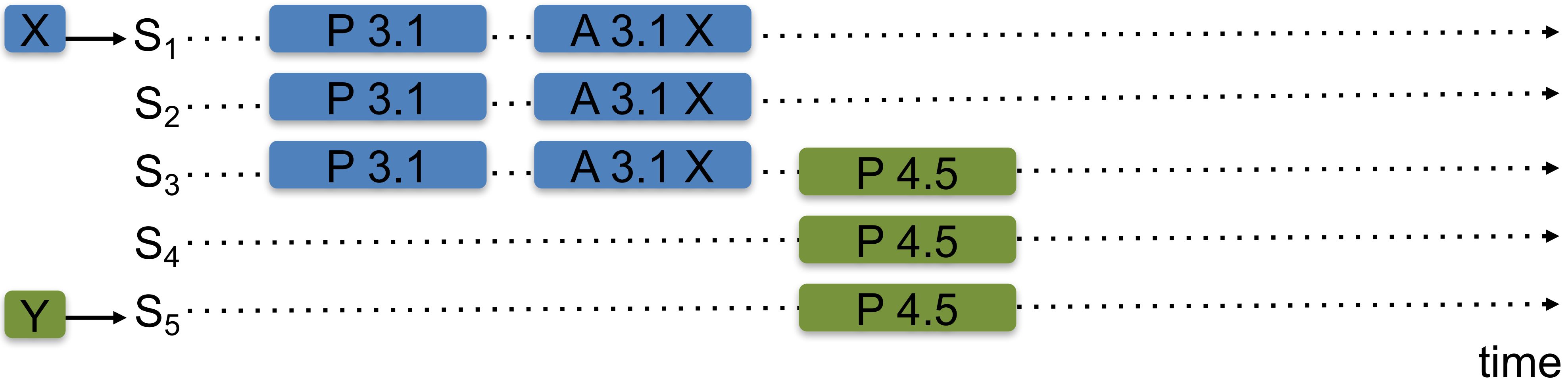
The setup

- Five servers: S1 through S5
- Two proposers
- Three acceptors

Paxos Examples

1. Previous value already chosen:

- Server 5 sends prepare requests (green) to the majority of the servers. There is going to be some overlap. What is happening at the accept stage?

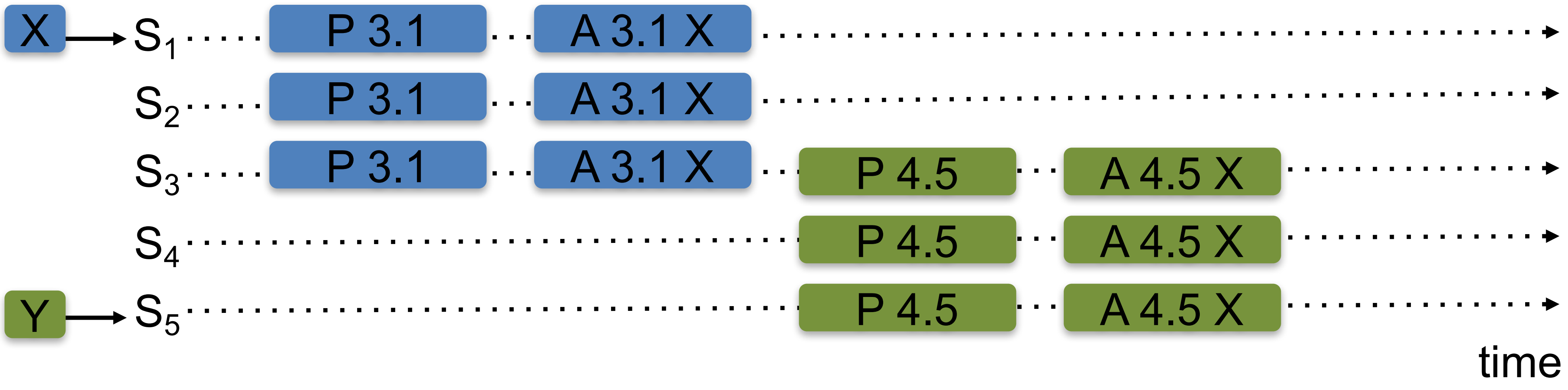


- P 3.1 = “Prepare proposal with #3 from server 1”,
- A 4.5 X = “Accept proposal with #4 from server 5 with value X”

Paxos Examples

1. Previous value already chosen:

- Server 5 sends prepare requests (green) to the majority of the servers. There is going to be some overlap. What is happening at the accept stage?

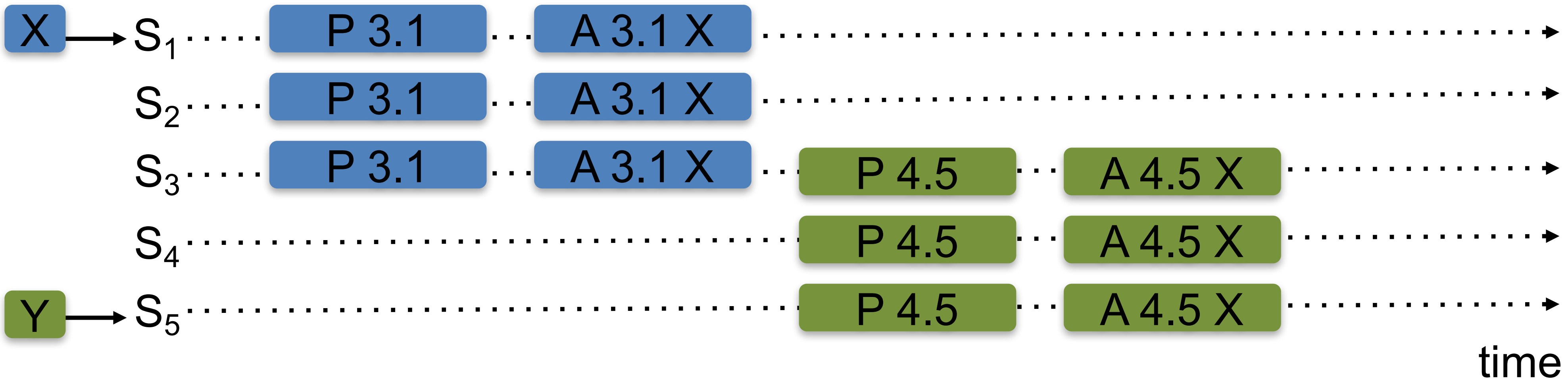


- New proposer will find it and use it

Paxos Examples

1. Previous value already chosen:

- Server 5 sends prepare requests (green) to the majority of the servers. There is going to be some overlap. What is happening at the accept stage?



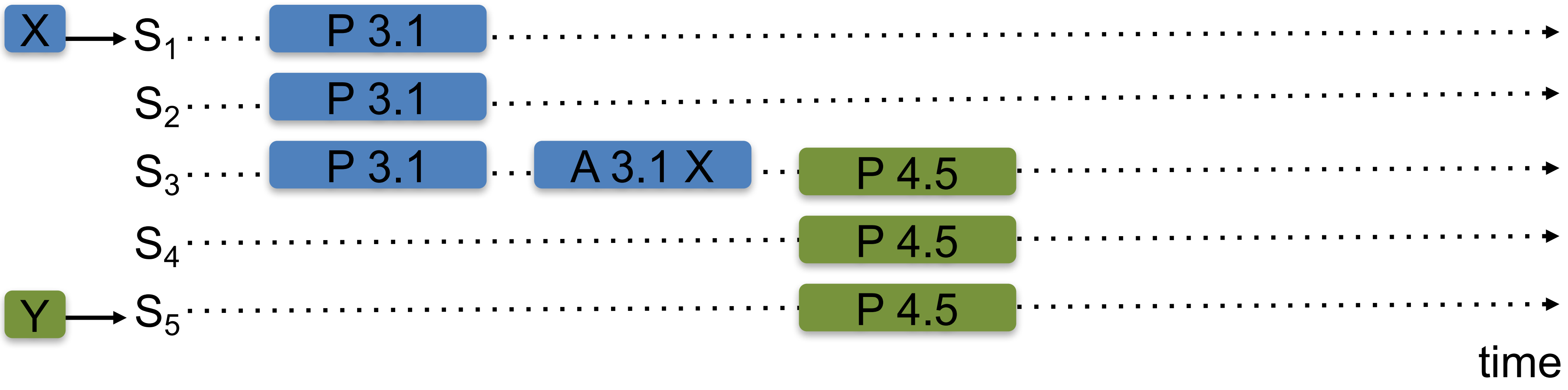
- New proposer will find it and use it

- Server 5 will see X as a response to its prepare request and will abandon the Y value. Then, it will accept the X value. Server 5 manages to get a value chosen, but this is the same X value that was previously chosen.

Paxos Examples

2. Previous value not chosen, but new proposer sees it:

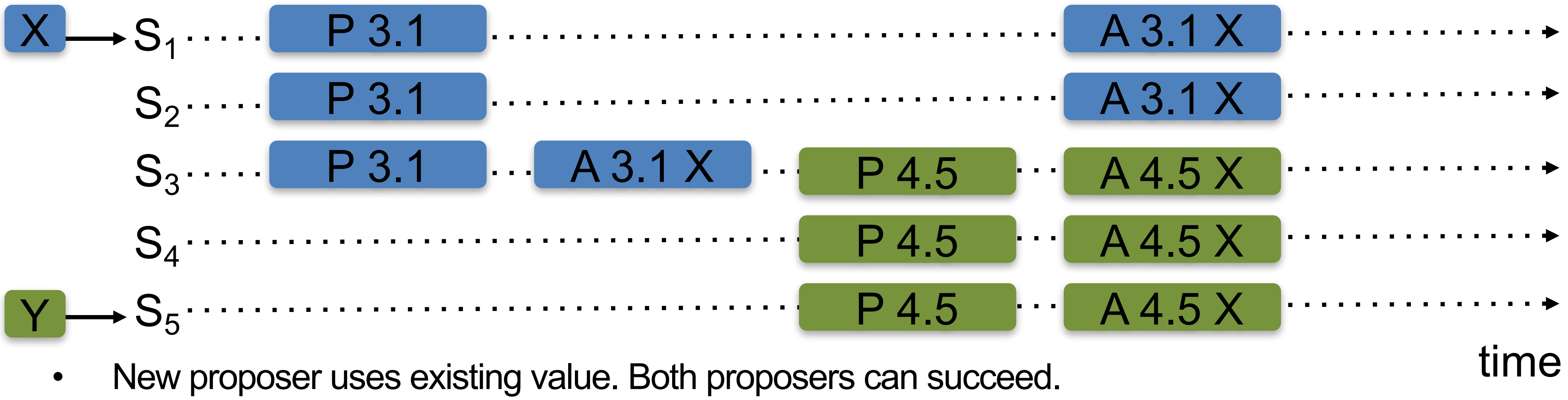
- Server 5 sends prepare requests (green) to the majority of the servers. The proposer will see the value that proposing Server 1 sent. What is happening at the accept stage?



Paxos Examples

2. Previous value not chosen, but new proposer sees it:

- Server 5 sends prepare requests (green) to the majority of the servers. The proposer will see the value that proposing Server 1 sent. What is happening at the accept stage?



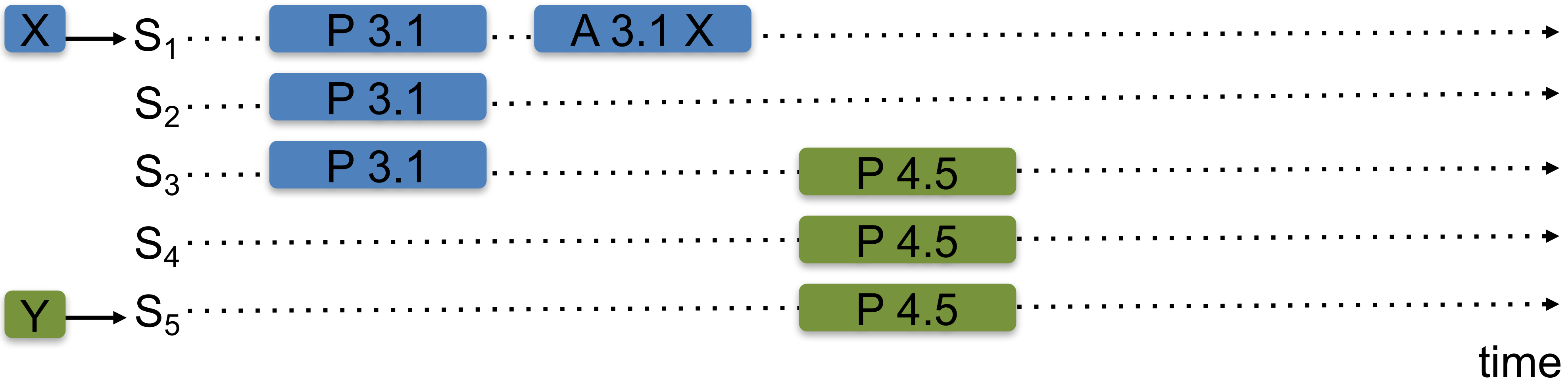
- New proposer uses existing value. Both proposers can succeed.

- Server 5 gets value X from server 3. It doesn't know if X value has been chosen, because it only talks to the majority of the servers. It assumes that possibly that the value has been chosen. It will go with the other value instead of its own value.

Paxos Examples

3. Previous value not chosen, new proposer doesn't see it

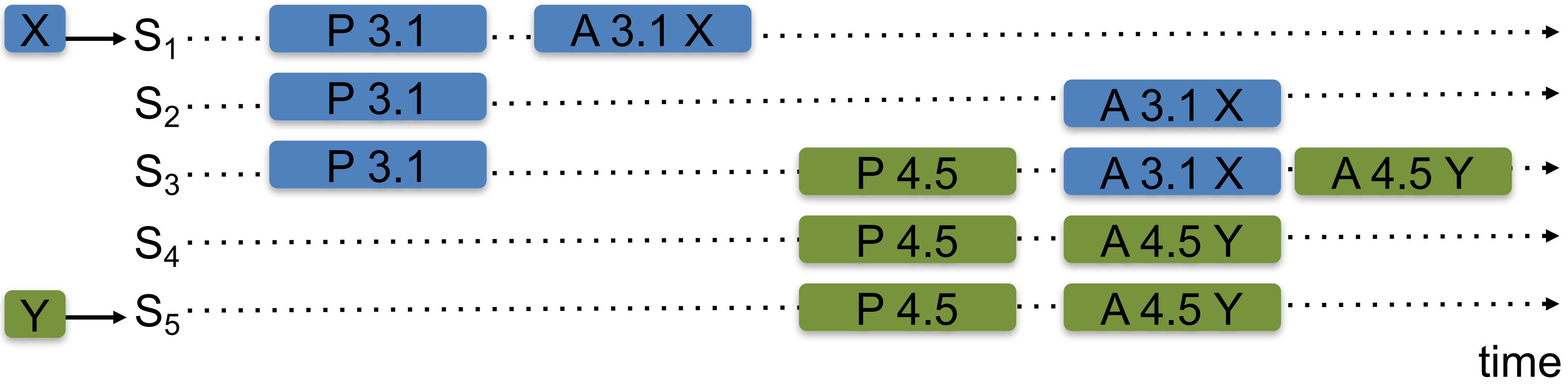
- Server 5 sends prepare requests (green) to the majority of the servers. The proposer will not see the value that proposing Server 1 sent. What is happening at the accept stage?



Paxos Examples

3. Previous value not chosen, new proposer doesn't see it

- Server 5 sends prepare requests (green) to the majority of the servers. The proposer will not see the value that proposing Server 1 sent. What is happening at the accept stage?

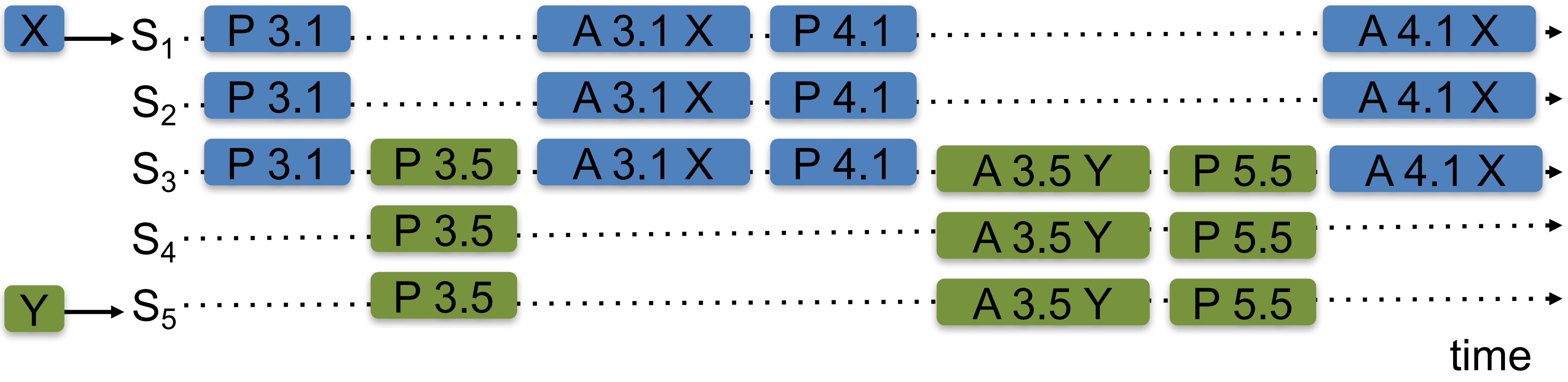


- New Proposer chooses its own value. Older proposal blocked. What happens for Server 1?

- On the servers that Server 5 checks, there is no other value returned. Eventually, Server 1 will get prepare requests to the majority of the servers. These will be discarded as Server 5 has a higher proposal number.The server will go ahead and accept its own value. Y will then be chosen.

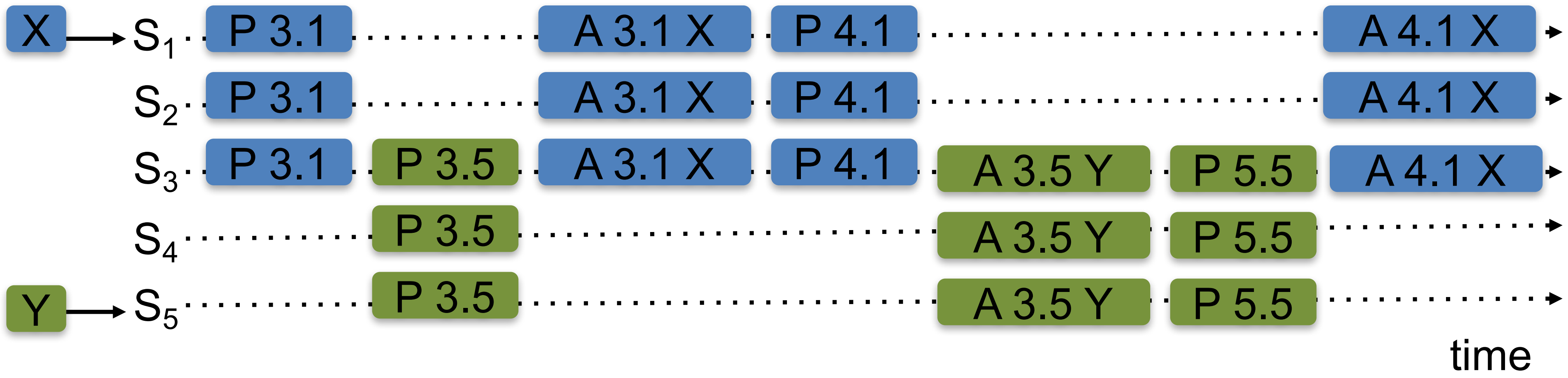
Paxos Examples

- Do you see a problem here?



Paxos Examples

- Do you see a problem here?



- Competing proposers can livelock. Proposer 1 (S1) completes a round of prepares, but before it completes the accepting phase, another server does its round of prepares. That cuts off the accept phase for server 3, and therefore Proposer 1 starts again with proposal #4. The same goes on (and on and on) for Proposer 2 (S5).

- One solution: randomized delay before restarting. That gives other proposers a chance to finish choosing.

Fill out this poll:

https://goo.gl/R9lydk

Assignment 2

Due October 19

Wednesday topic

Consensus II: Viewstamped Replication and Raft

Additional resources for Paxos

- Example from class: https://www.youtube.com/watch?v=JEpsBg0AO6o
- https://www.quora.com/Distributed-Systems-What-is-a-simple-explanation-of-the-Paxos-algorithm
- https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2003-96.pdf
- http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf