Paxos

(deck based on slides from Lorenzo Alvisi and Tom Anderson)

The Part-Time Parliament

Leslie Lamport
ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. ***This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60].***
ACM SIGOPS Hall of Fame Award in 2012

- Parliament determines laws by passing sequence of numbered decrees
- Legislators can leave and enter the chamber at arbitrary times
- No centralized record of approved decrees– instead, each legislator carries a ledger

Safe Replication?

- Suppose using primary/hot standby replication
- How can we tell if primary has failed versus is slow? (if slow, might end up with two primaries!)
- FLP: impossible for a deterministic protocol to guarantee **consensus** in bounded time in an asynchronous distributed system (even if no failures actually occur and all messages are delivered)

2PC vs. Paxos?

- Two phase commit: blocks if coordinator fails after the prepare message is sent, until the coordinator recovers
- Paxos: non-blocking as long as a majority of participants are alive, provided there is a sufficiently long period without further failures
- By FLP cannot have both safety+liveness
  - Paxos guarantees safety, tries to be live

Operating model

- A set of processes that can propose values
- Processes can crash and recover
- Processes have access to stable storage
- Asynchronous communication via messages
- Messages can be lost and duplicated, but not corrupted

The Game: Consensus

SAFETY

- Only a value that has been proposed can be chosen
- Only a single value is chosen (consistency)
- A process never learns that a value has been chosen unless it has been (~atomicity)

LIVENESS

- Some proposed value is eventually chosen
- If a value is chosen, a process eventually learns it

The Game: Consensus

SAFETY

- Only a value that has been proposed can be chosen
- Only a single value is chosen (consistency)
- A process never learns that a value has been chosen unless it has been (~atomicity)

LIVENESS

- Some proposed value is eventually chosen
- If a value is chosen, a process eventually learns it

Consensus about one value can be generalized to consensus about a sequence of values: the sequence of operations to apply to a replicated state machine. Essentially, consensus about "what is the next operation to apply?" Note that in general, we don't care what the order of operations is, as long as there is an order, we all agree on it, and we can continue to make progress during failures.

The Players

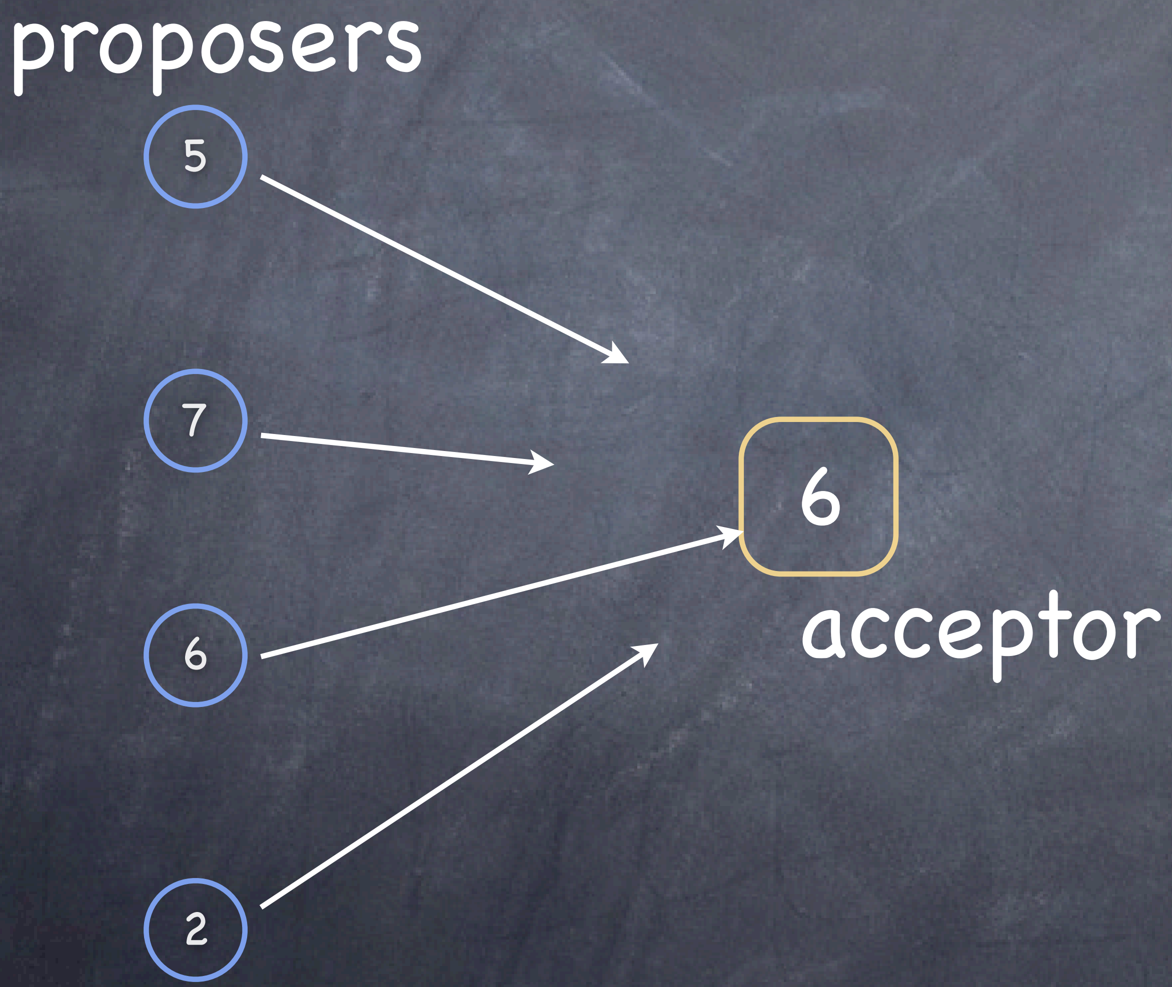
- Proposers
- Acceptors
- Learners

The Players

- Proposers
- Acceptors
- Learners

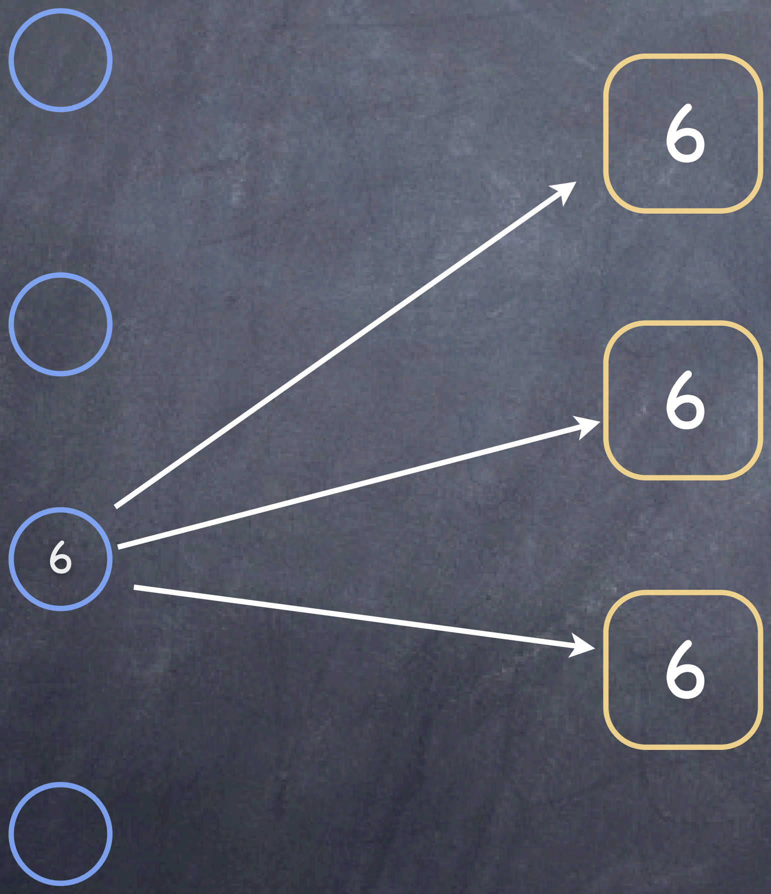
In a real implementation, these roles are implemented by a single node/process

Choosing a value



Use a single acceptor

What if the acceptor fails?



6 is chosen!

- Choose only when a "large enough" set of acceptors accepts
- Using a majority set guarantees that at most one value is chosen

Accepting a value

- Suppose only one value is proposed by a single proposer.
- That value should be chosen! (if not, then no liveness = cannot make progress)
- First requirement:

  P1: An acceptor must accept the first proposal that it receives

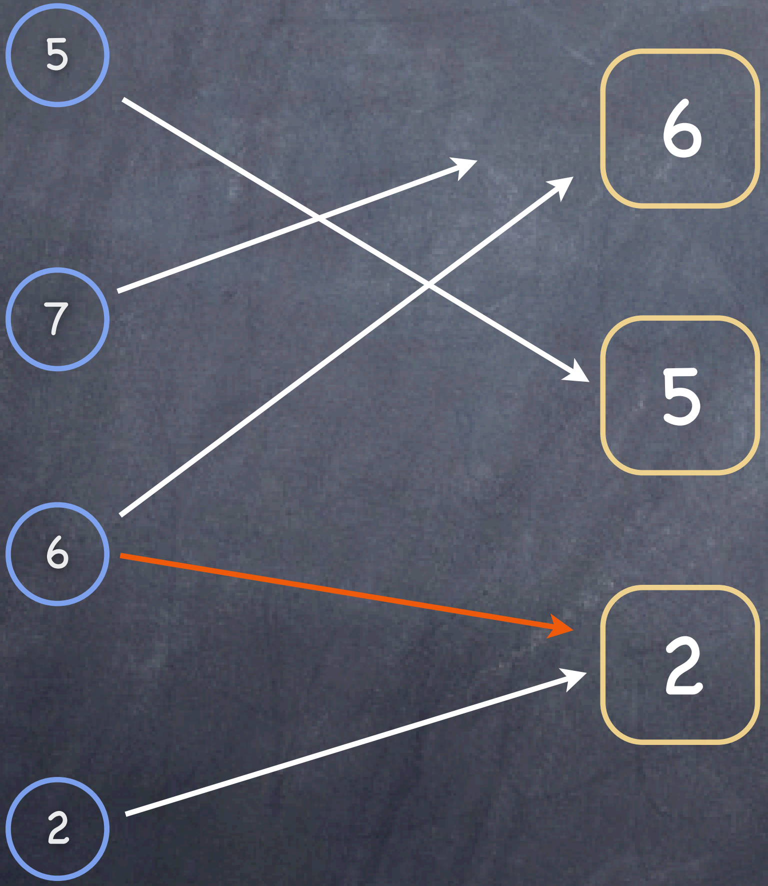
Accepting a value

- Suppose only one value is proposed by a single proposer.
- That value should be chosen!
- First requirement:

  P1: An acceptor must accept the first proposal that it receives

- ...but what if we have multiple proposers, each proposing a different value?

P1 + multiple proposers



No value is chosen!

Handling multiple proposals

- Realization: acceptors must (be able to) accept more than one proposal

- To track different proposals, assign a natural number to each proposal (*psn : proposal number)*
  - ☐ A proposal is then a pair (*psn*, value)
  - ☐ Different proposals have different *psn*
  - ☐ A proposal is chosen: when it has been accepted by a majority of acceptors
  - ☐ A value is chosen: when a single proposal with that value has been chosen

Choosing a unique value

- We need to guarantee that all chosen proposals result in choosing the same value
- We introduce a second requirement (by induction on the proposal number):

  P2. If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has value v

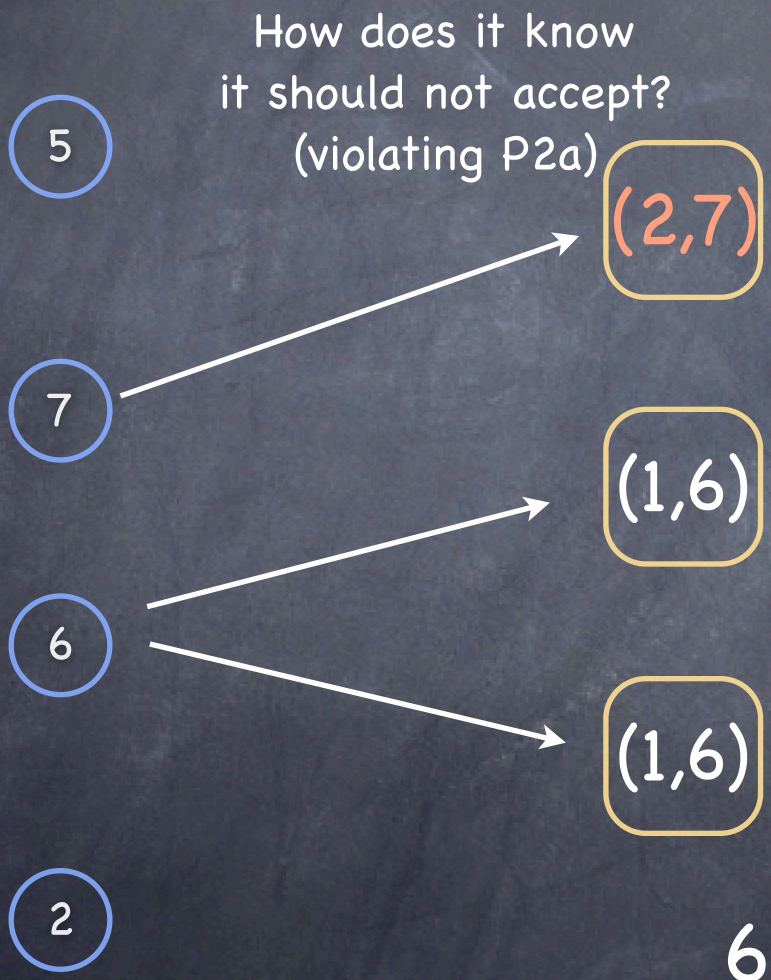
  which can be satisfied by:

  P2a. If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v

What about P1?

(P1: An acceptor must accept the first proposal that it receives)

(P2a: If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v)



- Do we still need P1?

  YES, to ensure that *some* proposal is accepted

- How well do P1 and P2a play together?

  Asynchrony is a problem...

6 is chosen!
(with psn 1) by P1

Another take on P2

- Recall P2a:

  If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v

We strengthen it to:

  P2b: If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v

Another take on P2

- Recall P2a:

  If a proposal with value v is chosen, then every higher-numbered proposal accepted by any acceptor has value v

We strengthen it to:

P2b is more restrictive than P2a: can't accept a proposal, if it isn't issued.

P2b: If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v

Implementing P2 (I)

P2b: If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v

Suppose a proposer p wants to issue a proposal numbered n. What value should p propose?

- If (n',v) with $n' < n$ is chosen, then in every majority set S of acceptors at least one acceptor has accepted (n',v)...
- ...so, if there is a majority set S where no acceptor has accepted (or will accept) a proposal with number less than n, then p can propose any value

Implementing P2 (II)

P2b: If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v

What if for all S (majority set) some acceptor ends up accepting a pair (n',v) with $n' < n$?

Claim (if met, P2b satisfied): p should propose the value of the highest numbered proposal among all accepted proposals numbered less than n

Proof: By induction on the number of proposals issued after a proposal is chosen (or by contradiction)

Implementing P2 (III)

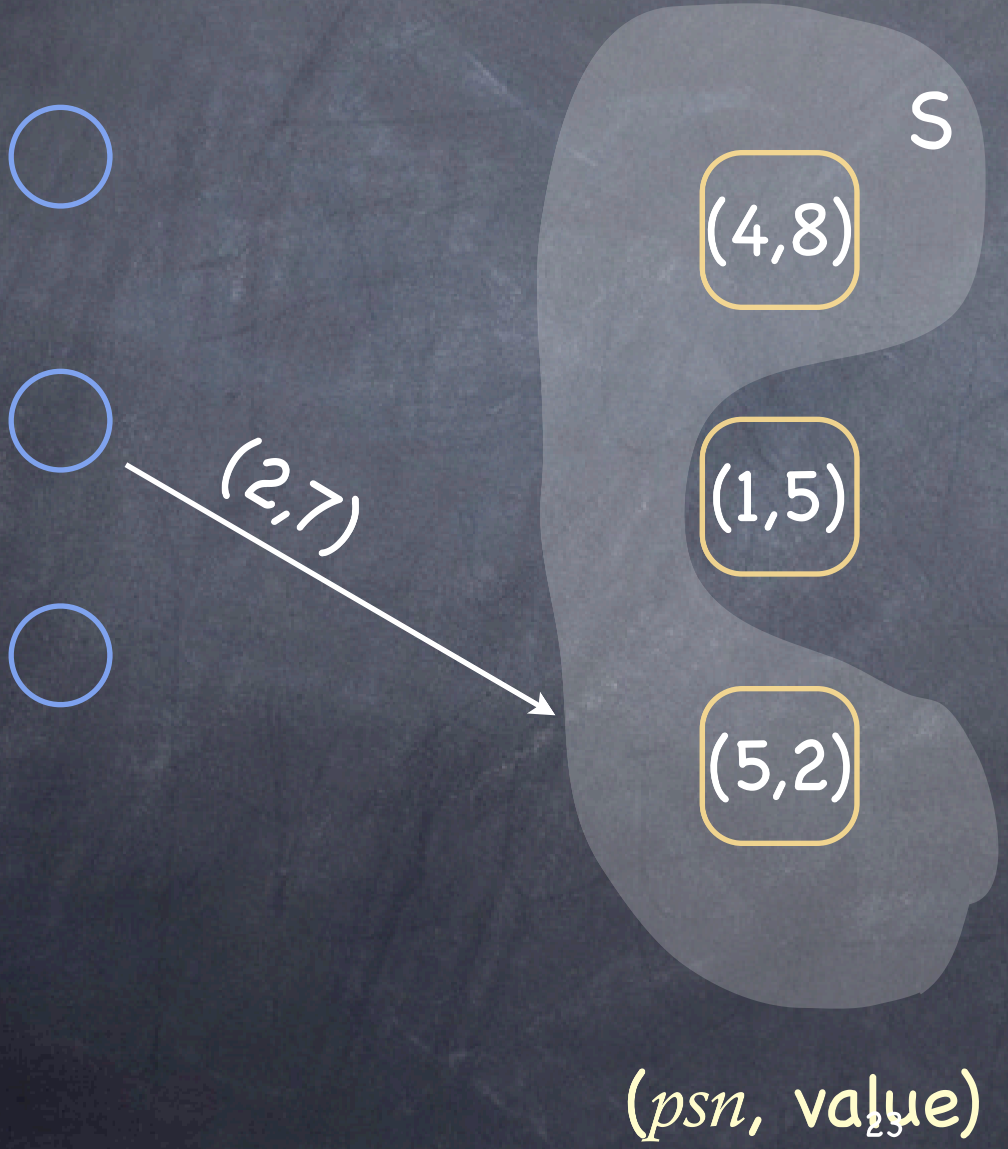
P2b: If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v

Achieved by enforcing the following invariant

P2c: For any v and n, if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors such that either:

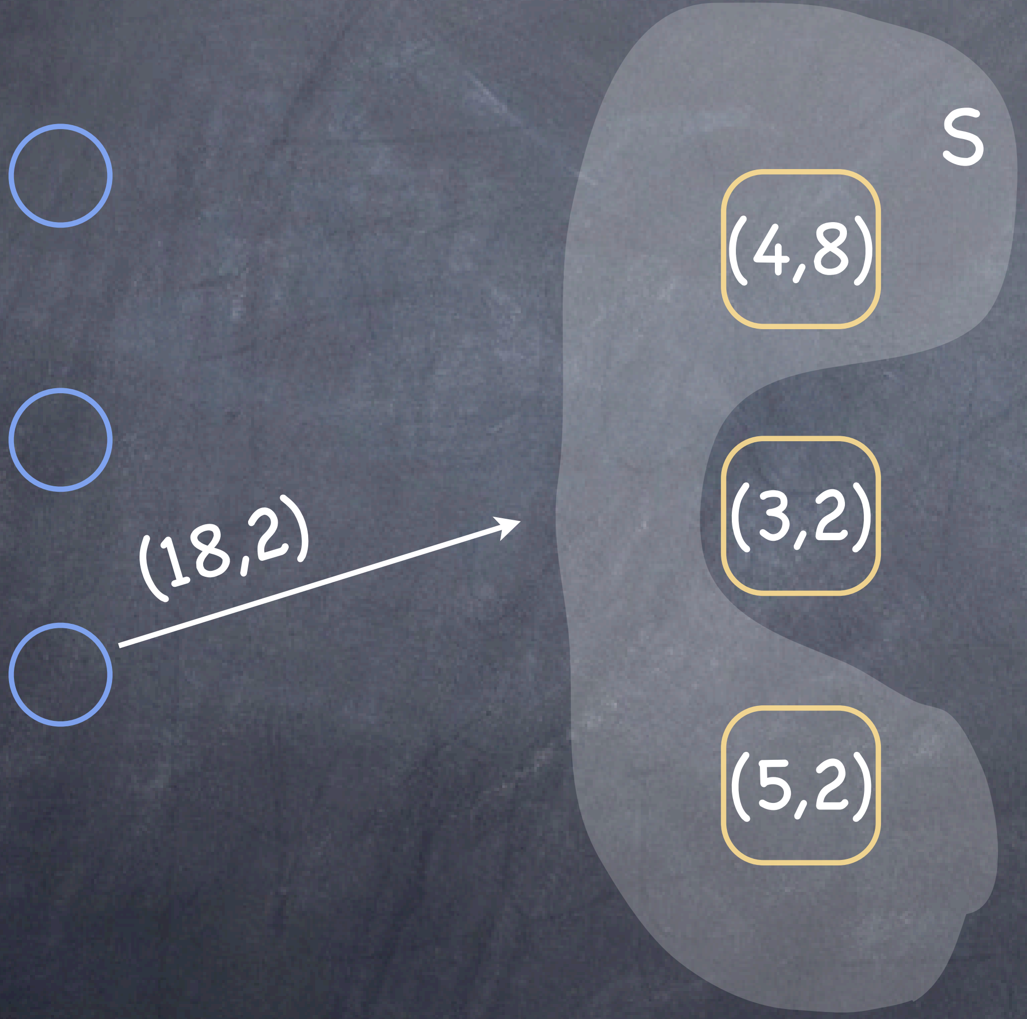
- no acceptor in S has accepted any proposal numbered less than n, or
- v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S

P2c in action



- No acceptor in S has accepted any proposal numbered less than psn n (=2)

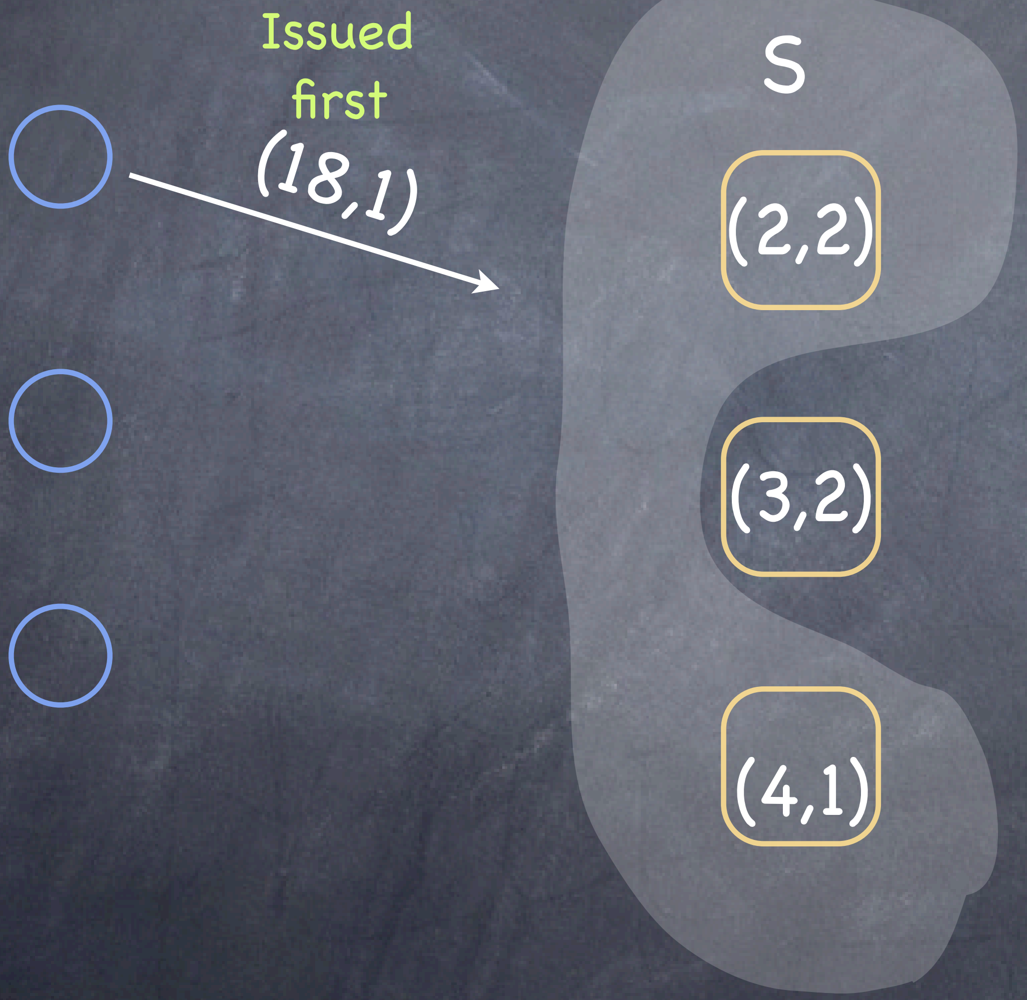
P2c in action



- *v (2)* is the value of the highest-numbered proposal (#5) among all proposals numbered less than *n (<18)* and accepted by the acceptors in S

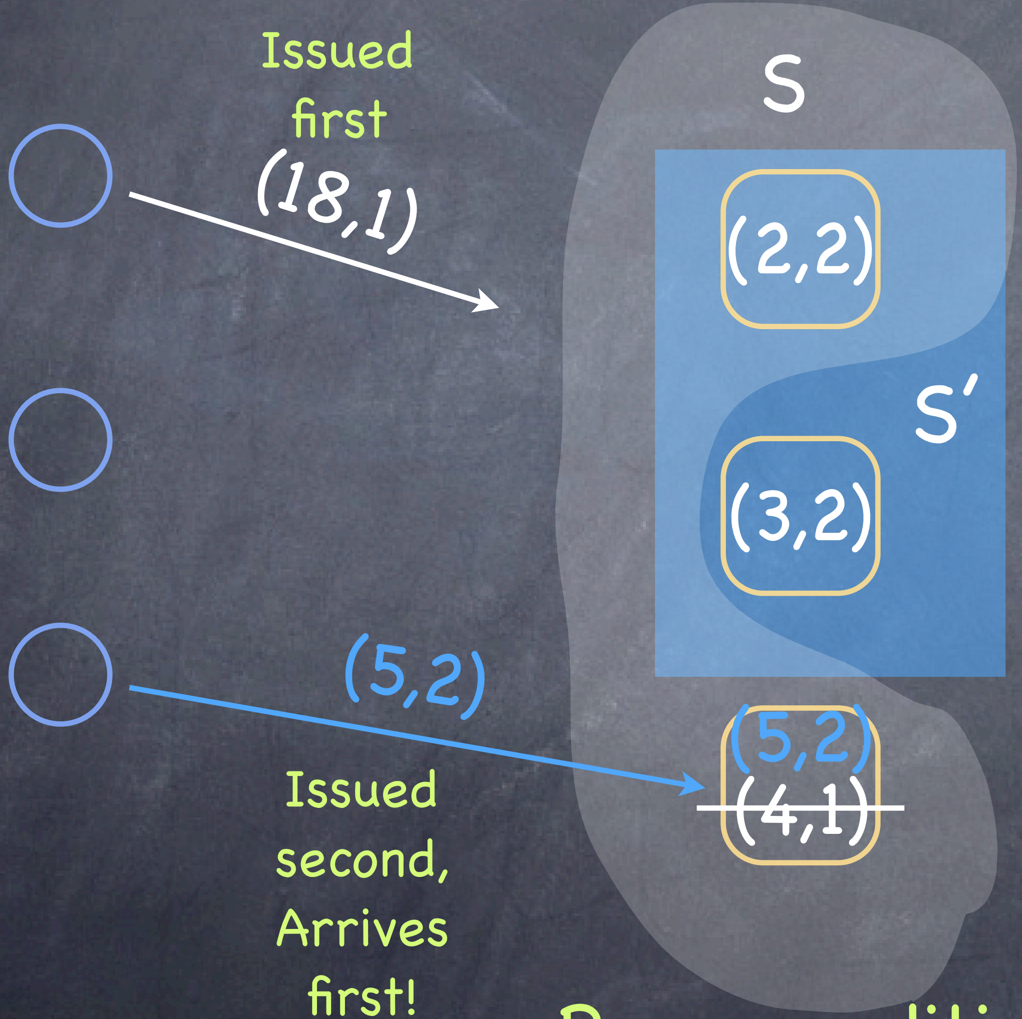
(*psn*, value)

P2c in action



- v is the value of the highest-numbered proposal among all proposals numbered less than n and accepted by the acceptors in S

P2c in action



- v is the value of the highest-numbered proposal among all proposals numbered less than n and accepted by the acceptors in S

Race condition between proposers:
The invariant may be violated

Future telling?

P2c: For any v and n, if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors such that either....

- To maintain P2c, a proposer that wishes to propose a proposal numbered n must learn the highest-numbered proposal with number less than n, if any, that has been or will be accepted by each acceptor in some majority of acceptors

Future telling?

- To maintain P2c, a proposer that wishes to propose a proposal numbered n must learn the highest-numbered proposal with number less than n, if any, that has been or will be accepted by each acceptor in some majority of acceptors

- Key strategy: avoid predicting the future by extracting a promise from a majority of acceptors not to subsequently accept any proposals numbered less than n

The proposer's protocol (I)

- A proposer chooses a new proposal number n and sends a request to each member of some (majority) set of acceptors, asking it to respond with:

  a. A promise never again to accept a proposal numbered less than n, and
  b. The accepted proposal with highest number less than n if any.

...call this a prepare request with number n

The proposer's protocol (II)

- If the proposer receives a response from a majority of acceptors, then it can issue a proposal with number n and value v, where v is

a. the value of the highest-numbered proposal among the responses, or
b. is any value selected by the proposer if responders returned no proposals

A proposer issues a proposal by sending, to some set of acceptors, a request that the proposal be accepted.

...call this an accept request.

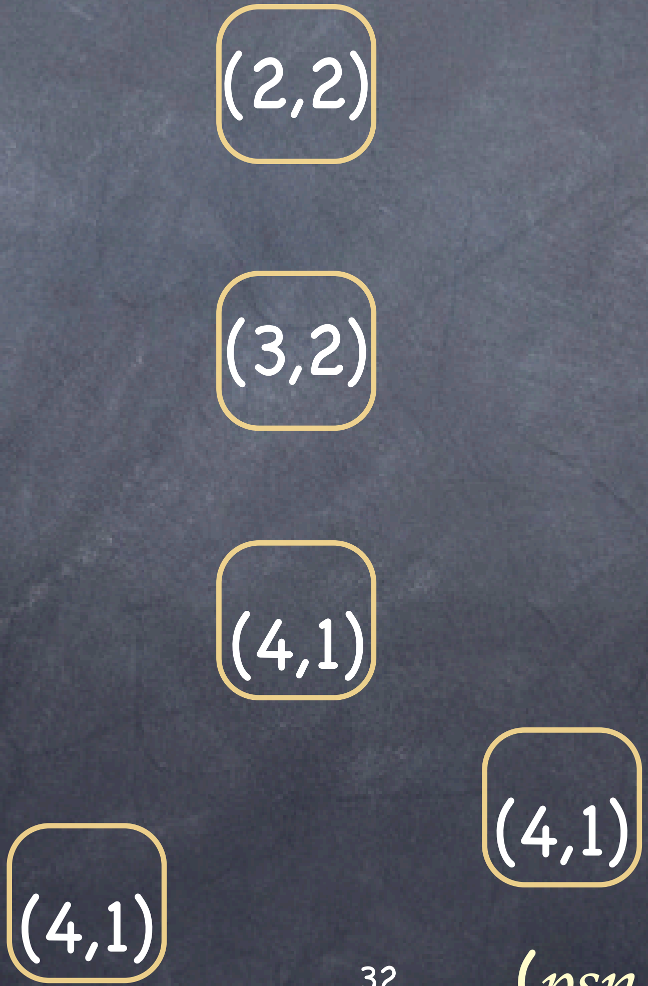
The acceptor's protocol

- An acceptor receives prepare and accept requests from proposers. It can ignore these without affecting safety.
  - ☐ It can always respond to a prepare request
  - ☐ It can respond to an accept request, accepting the proposal, iff it has not promised not to, e.g.

P1a: An acceptor can accept a proposal numbered n iff it has not responded to a prepare request having number greater than n

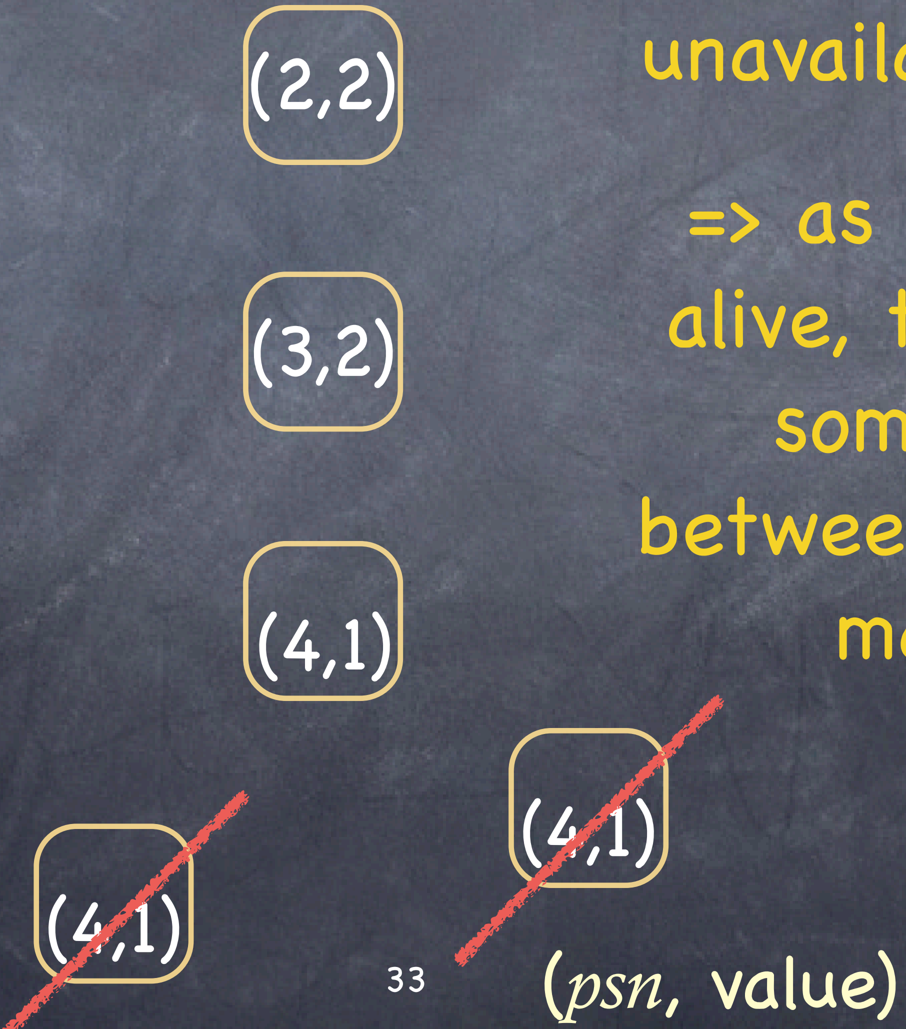
...which subsumes P1.

Putting it together
Initial sys config:



(psn, value)

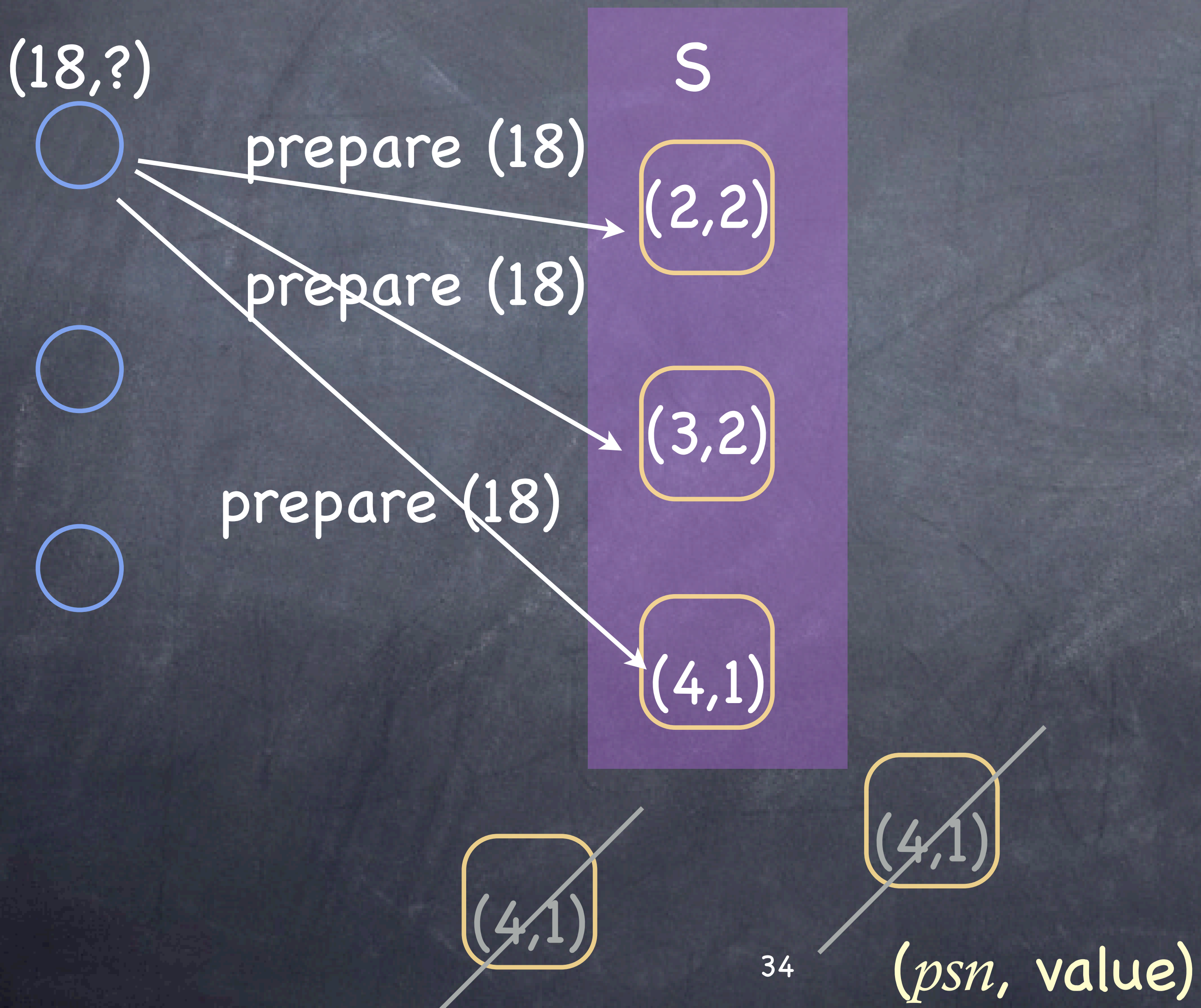
Minority fails



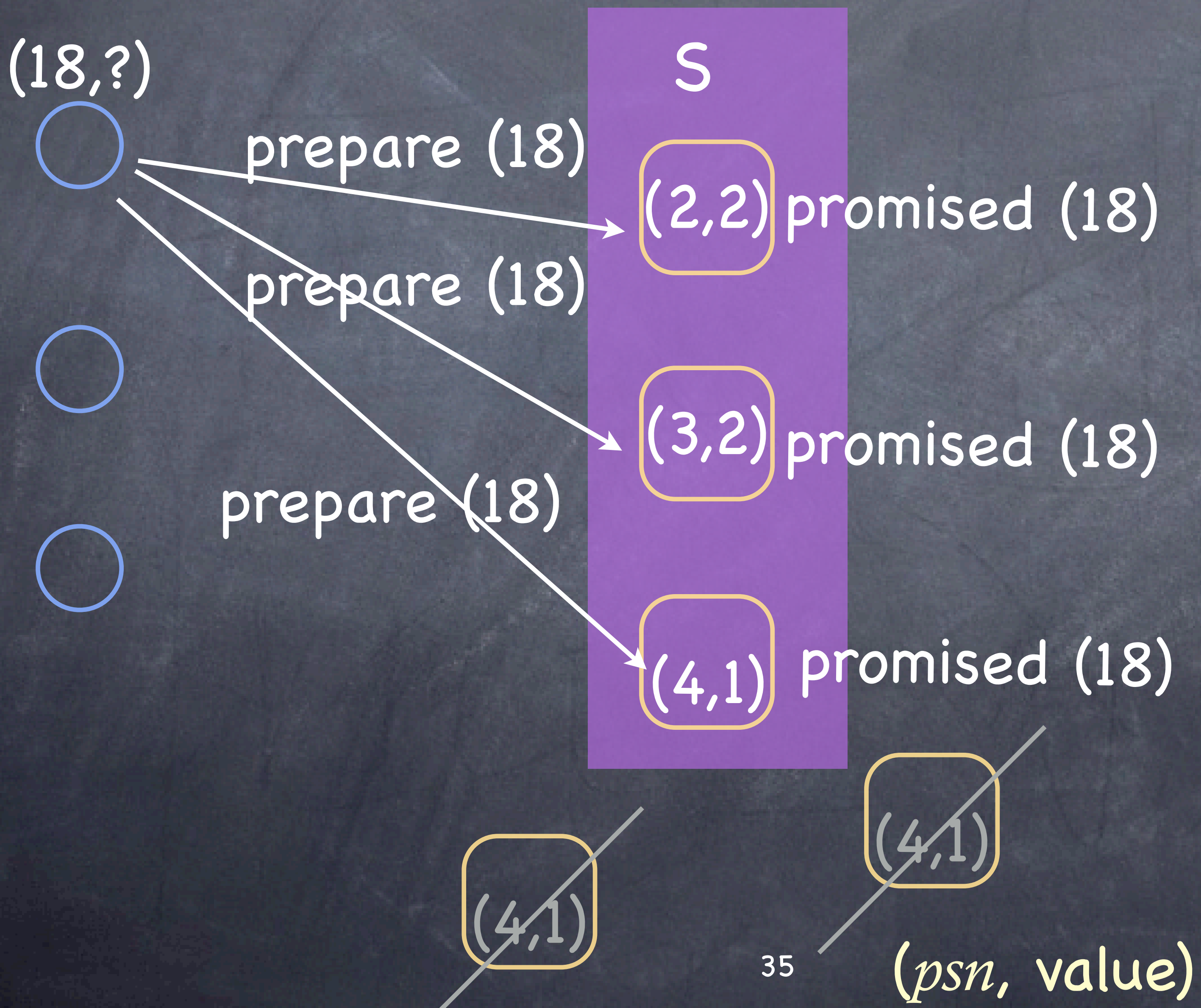
Note that if maj. fails, then Paxos is unavailable (not live)

=> as long as maj. alive, there will be some overlap between consecutive majorities

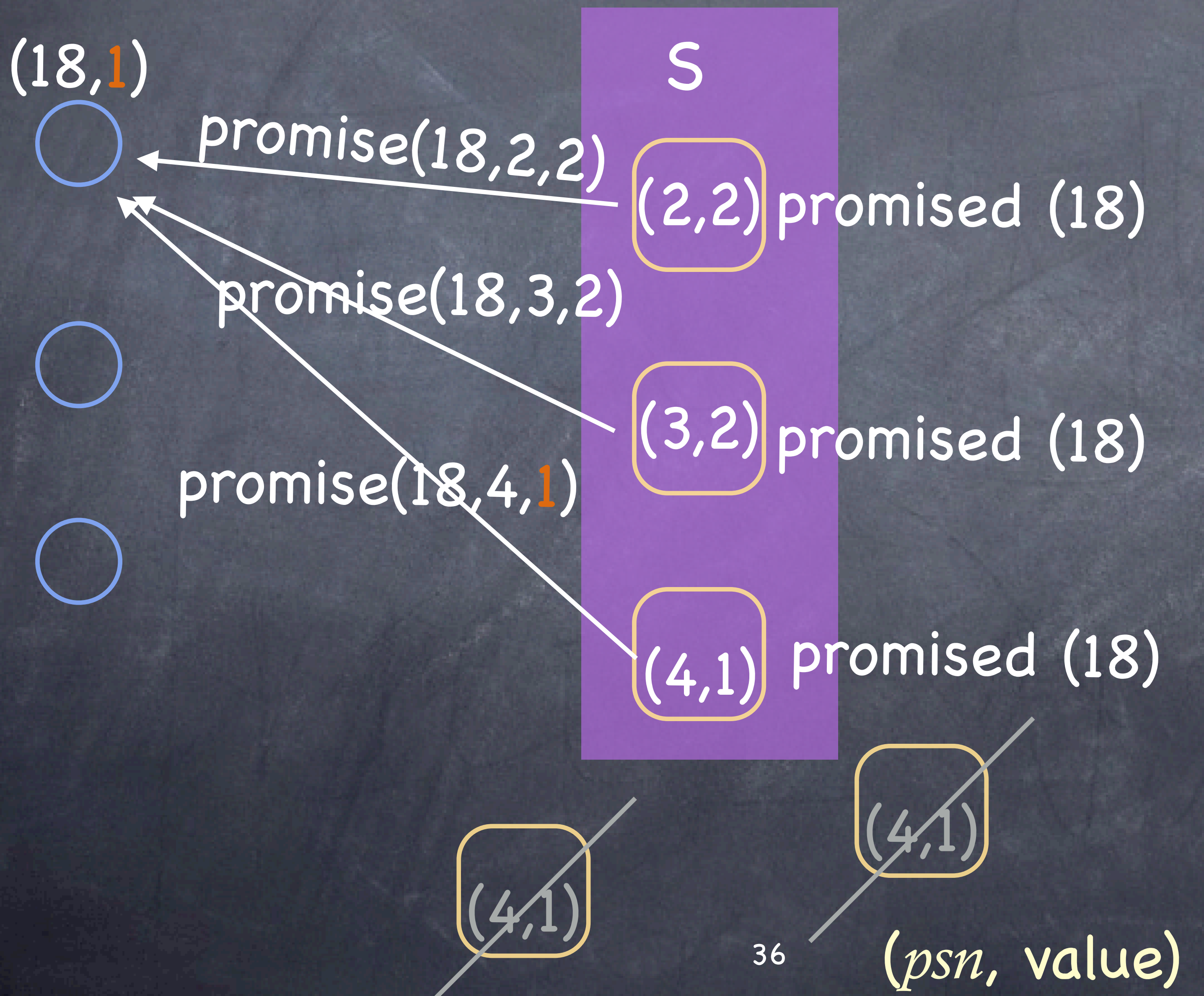
Working with remaining 3/5 majority



Working with remaining 3/5 majority



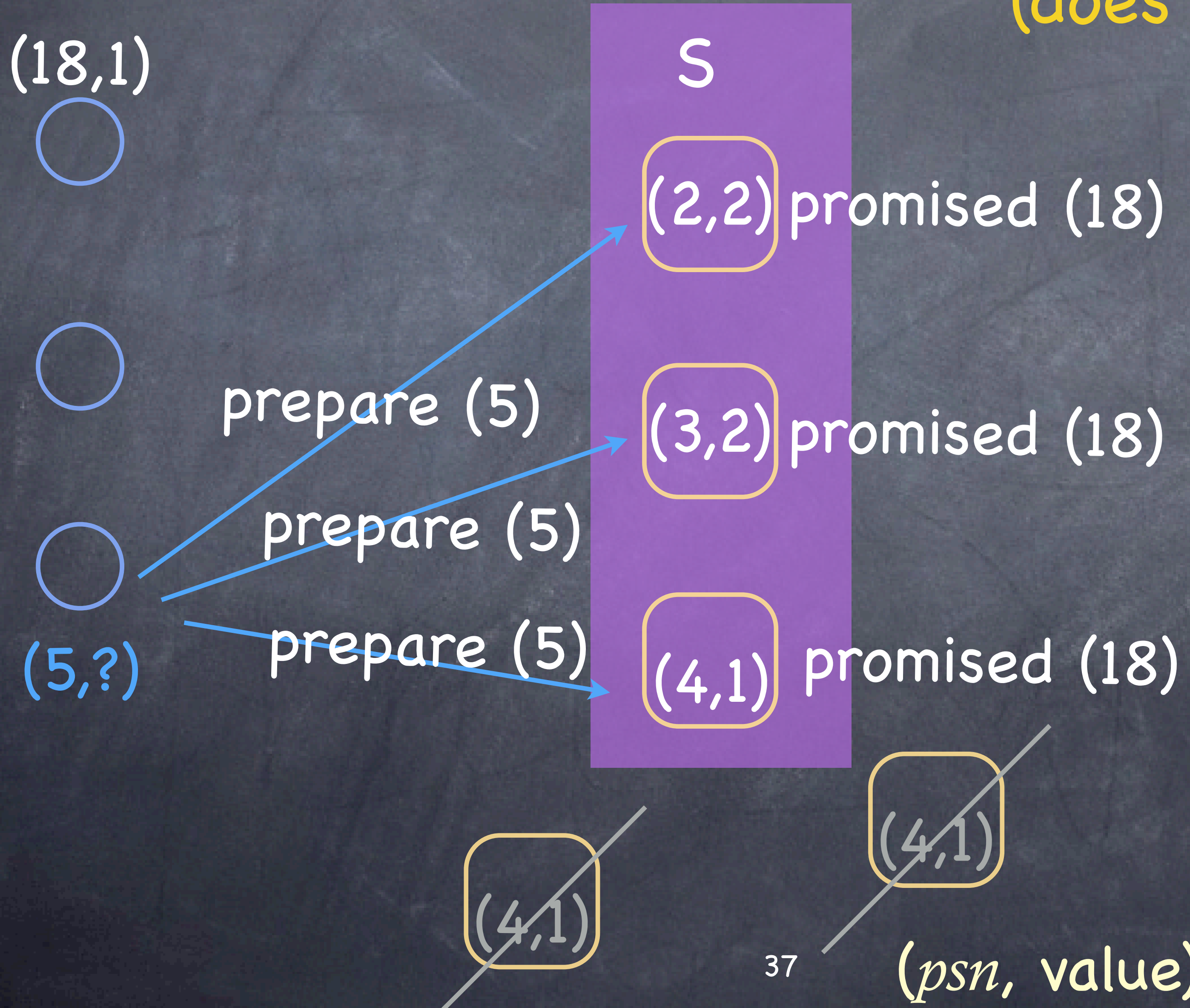
Working with remaining 3/5 majority



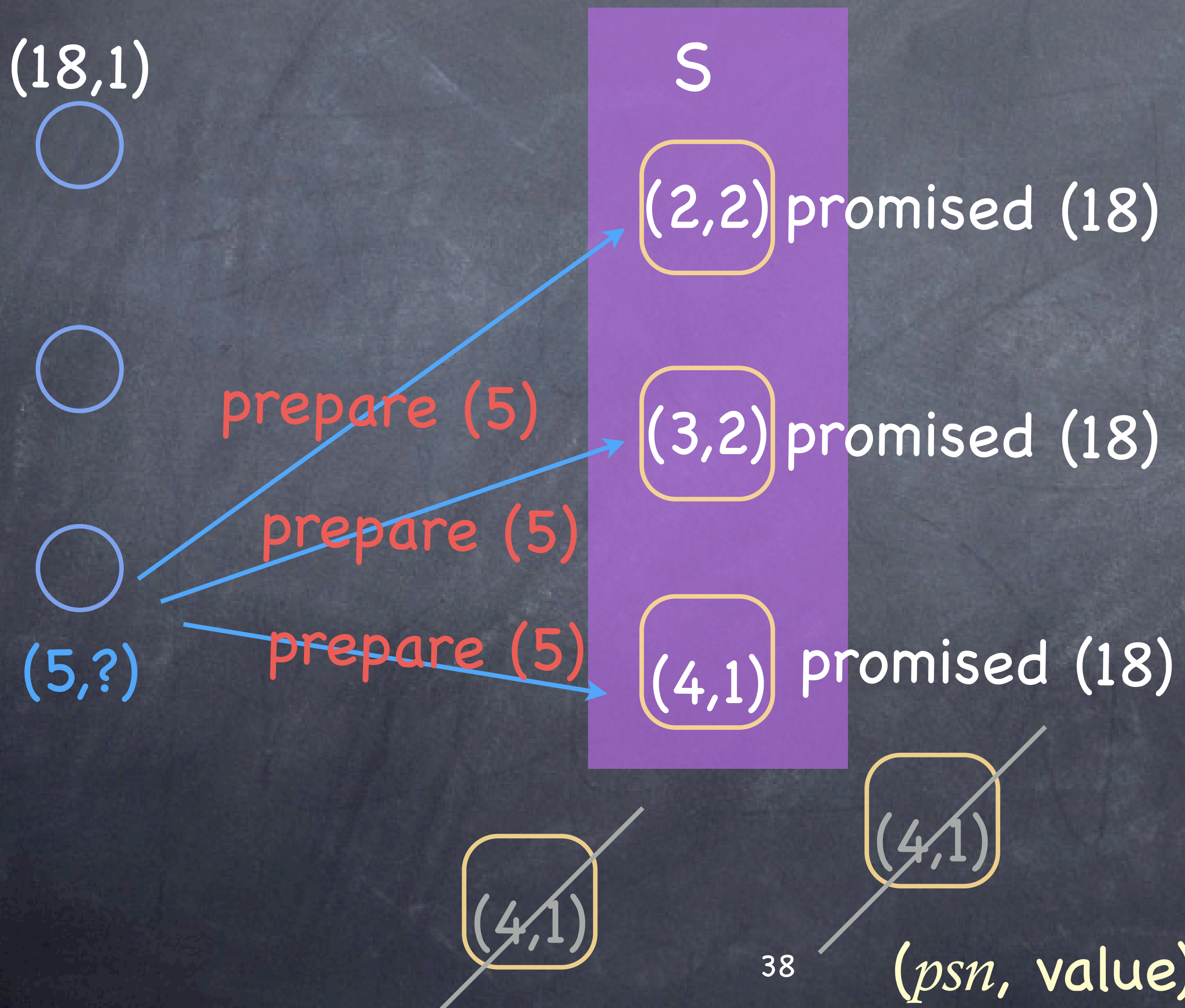
Promised to not accept any psn < 18

Majority overlap

Note: maj. overlap (does not need to be complete)

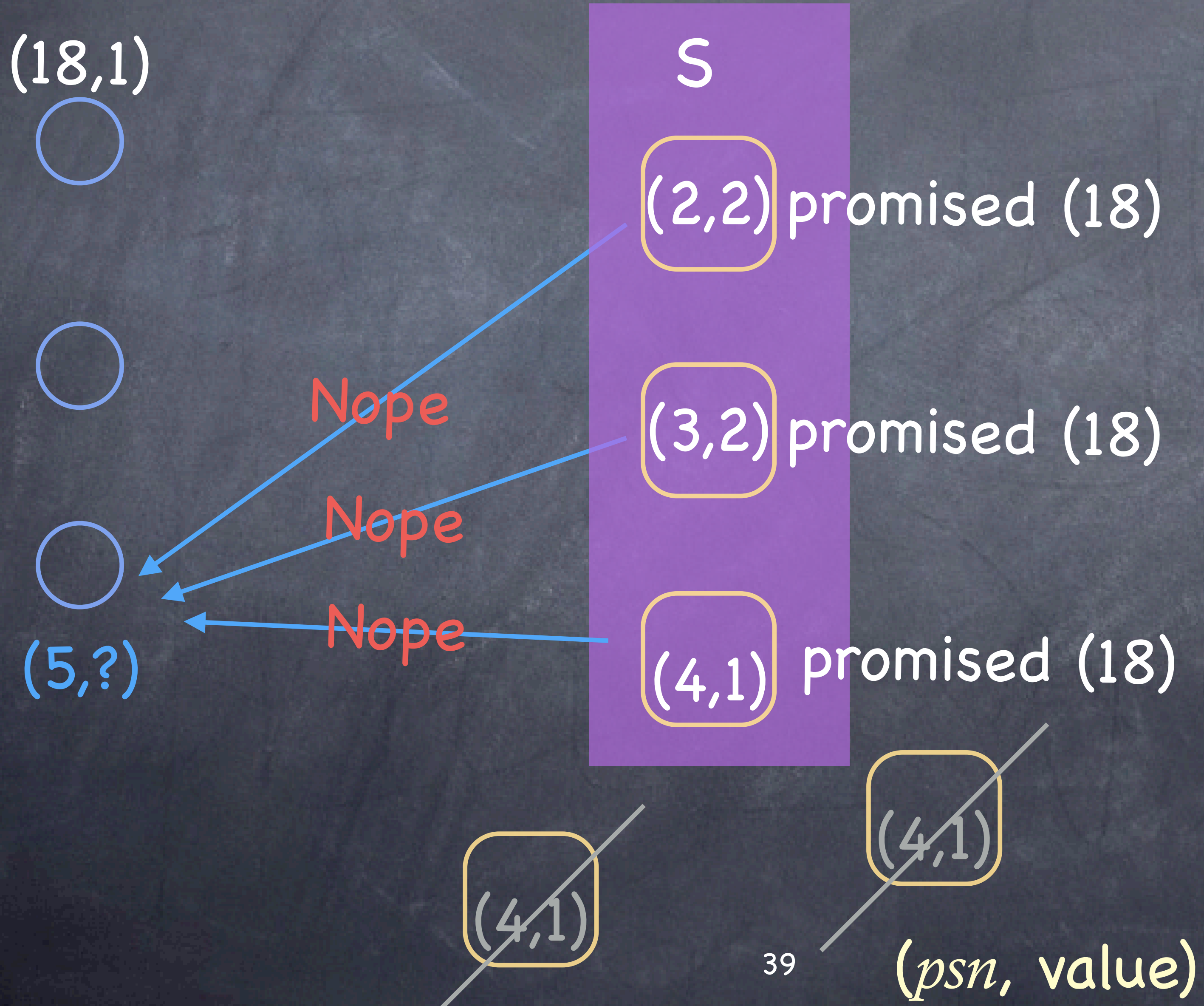


Prepare(5) conflicts with promised (18)



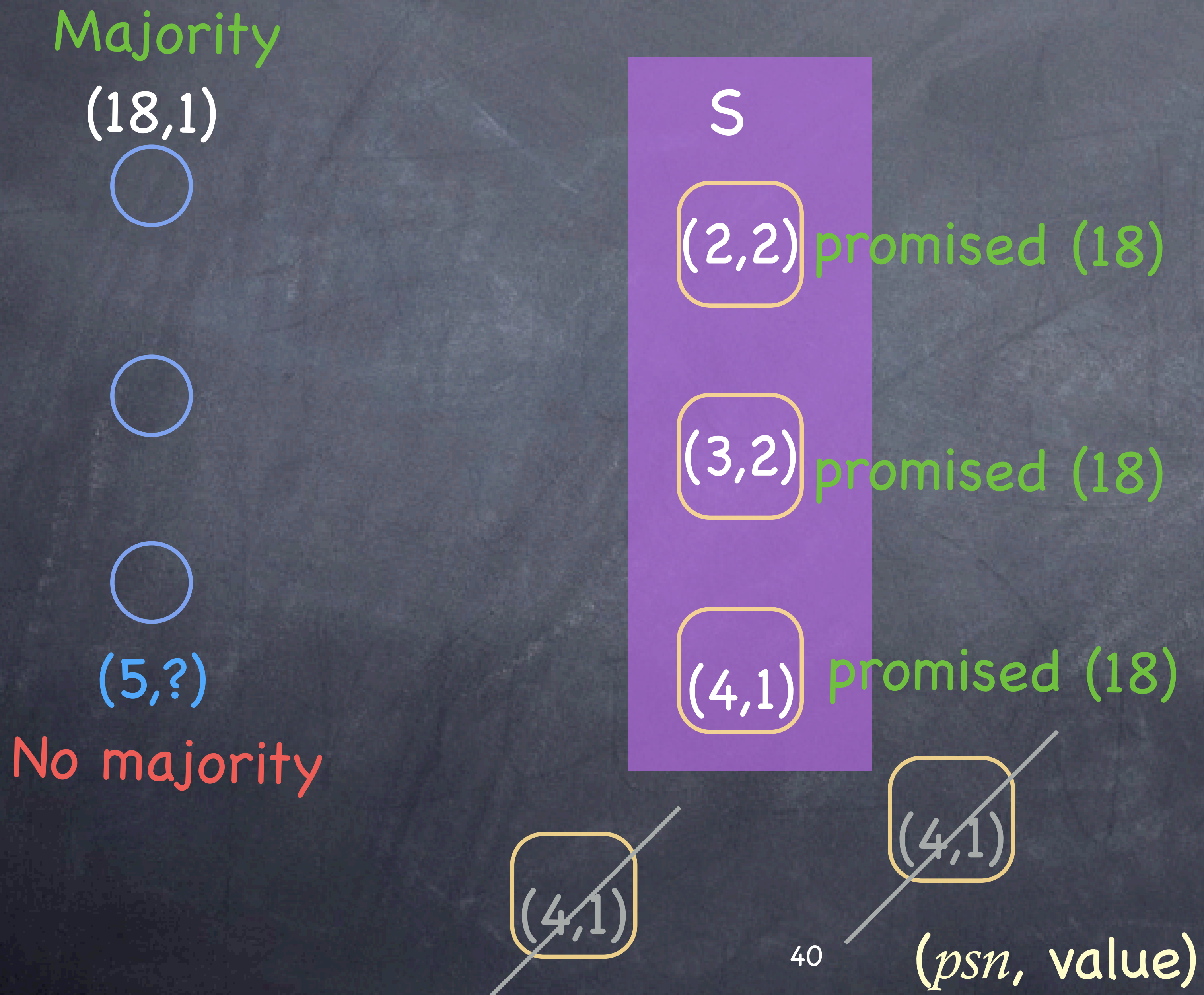
...Promised to not accept any psn < 18

Prepare(5) conflicts with promised (18)

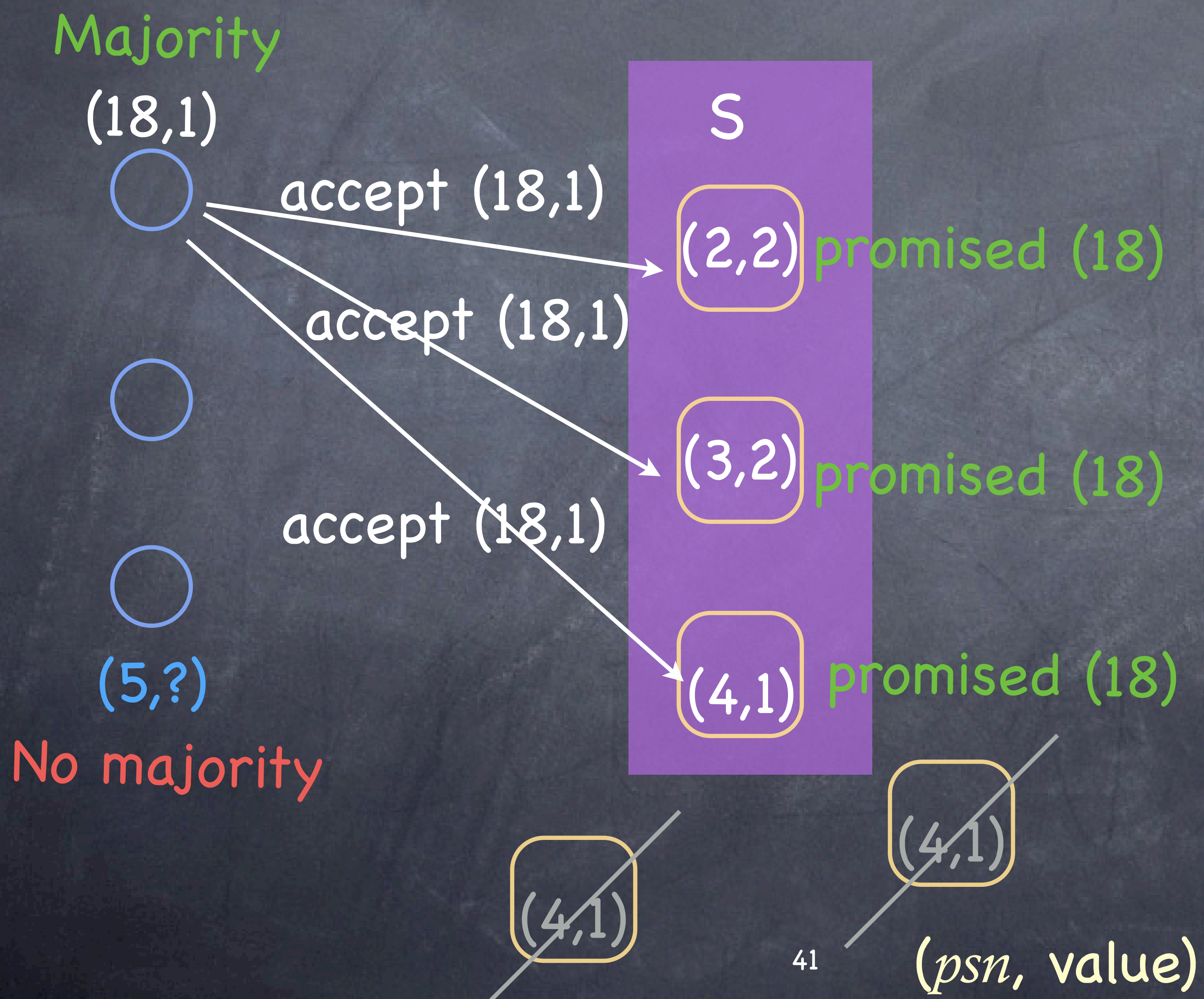


...Promised to not accept any psn < 18

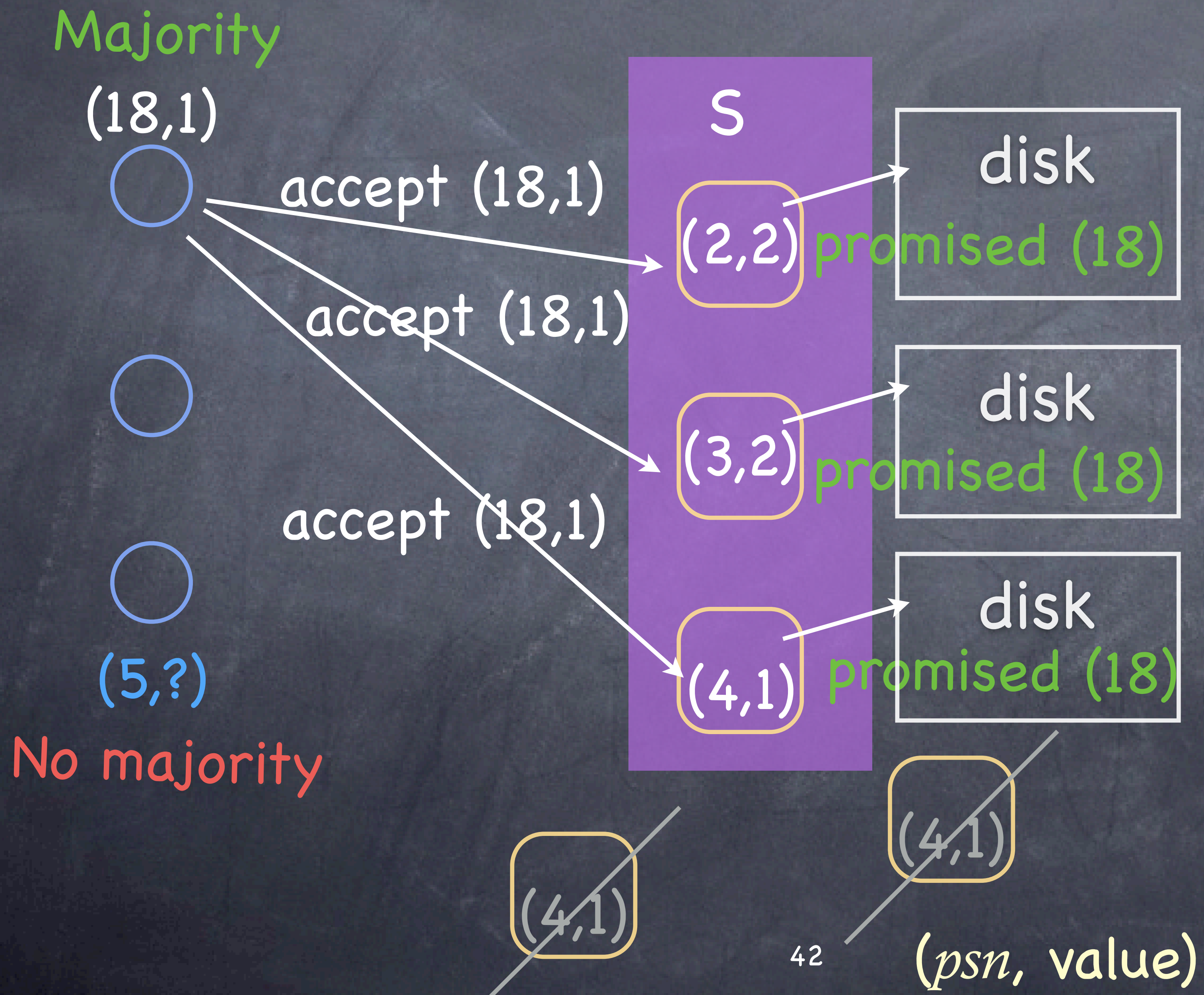
Outcome: just one proposer can (temporarily) prepare a majority



Outcome: just one proposer can (temporarily) prepare a majority



Outcome: just one proposer can (temporarily) prepare a majority



Small optimizations

- If an acceptor receives a prepare request numbered n when it has already responded to a prepare request for $n' > n$, then the acceptor can simply ignore this prepare.
- An acceptor can also ignore prepare requests for proposals it has already accepted

...so an acceptor needs only remember the highest numbered proposal it has accepted and the number of the highest-numbered prepare request to which it has responded.

This information needs to be stored on **stable storage** to allow restarts.

Summary: Choosing a value: Phase 1

- A proposer chooses a new *n* and sends <*prepare,n*> to a majority of acceptors
- If an acceptor receives <*prepare,n'*>, where *n' > n* of any <*prepare,n*> to which it has responded, then it responds to <*prepare, n'* > with
    - a promise not to accept any more proposals numbered less than *n'*
    - the highest numbered proposal (if any) that it has accepted

Summary: Choosing a value: Phase 2

- If the proposer receives a response to <*prepare,n*> from a majority of acceptors, then it sends to each <*accept,n,v*>, where *v* is either
  - the value of the highest numbered proposal among the responses
  - any value if the responses reported no proposals
- If an acceptor receives <*accept,n,v*>, it accepts the proposal unless it has in the meantime responded to <*prepare,n'*> , where $n' > n$

Learning chosen values (I)

Once a value is chosen, learners should find out about it. Many strategies are possible:

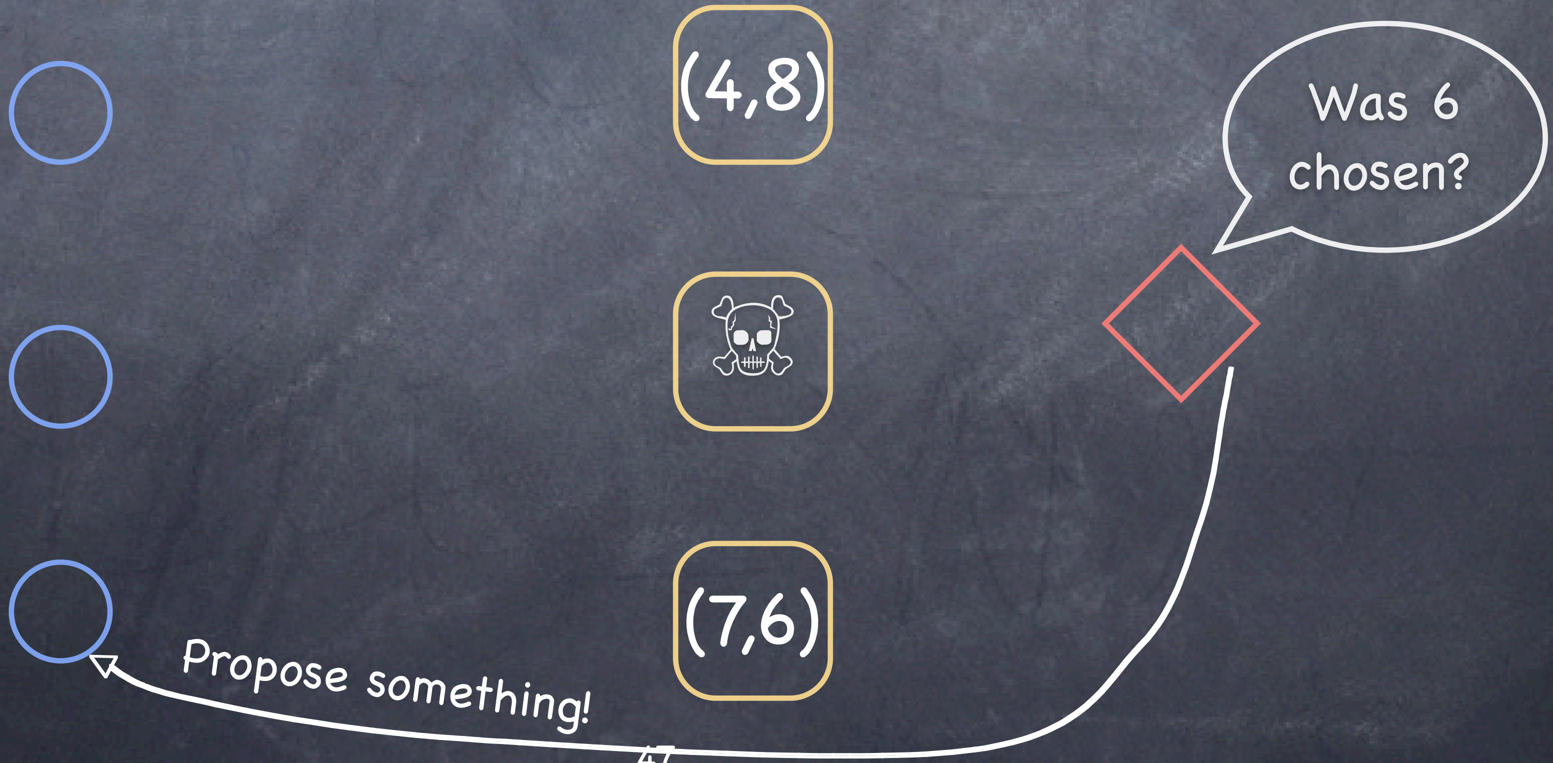
i. Each acceptor informs each learner whenever it accepts a proposal.

ii. Acceptors inform a distinguished learner, who informs the other learners

iii. Something in between (a set of not-quite-as-distinguished learners)

Learning chosen values (II)

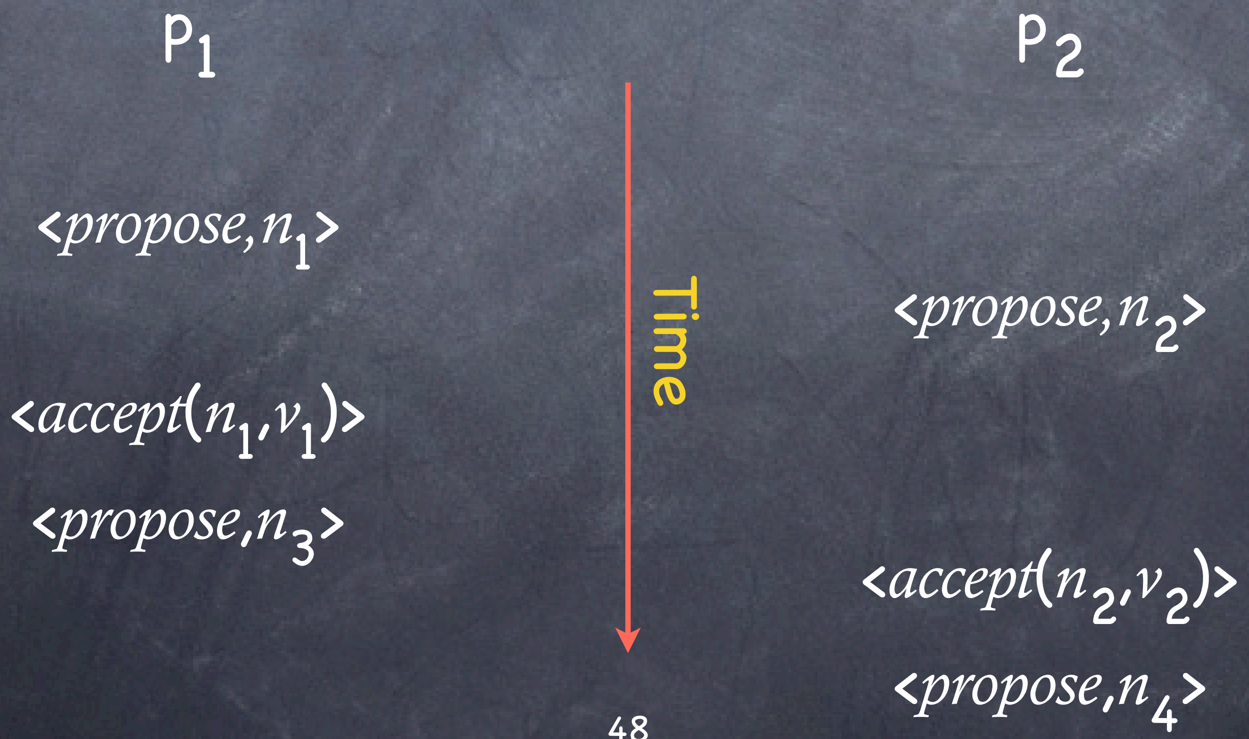
Because of failures (message loss and acceptor crashes) a learner may not learn that a value has been chosen



Liveness

Progress is not guaranteed:

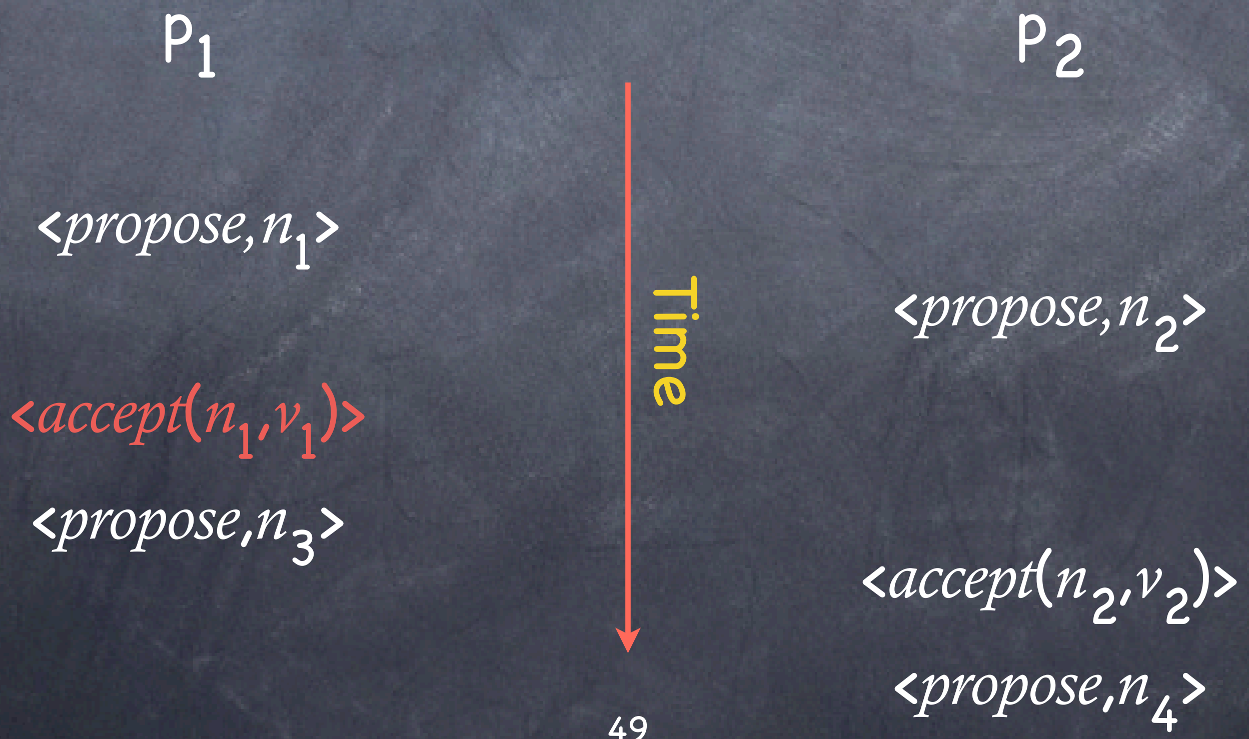
$n_1 < n_2 < n_3 < n_4 < \ldots$



Liveness

Progress is not guaranteed:

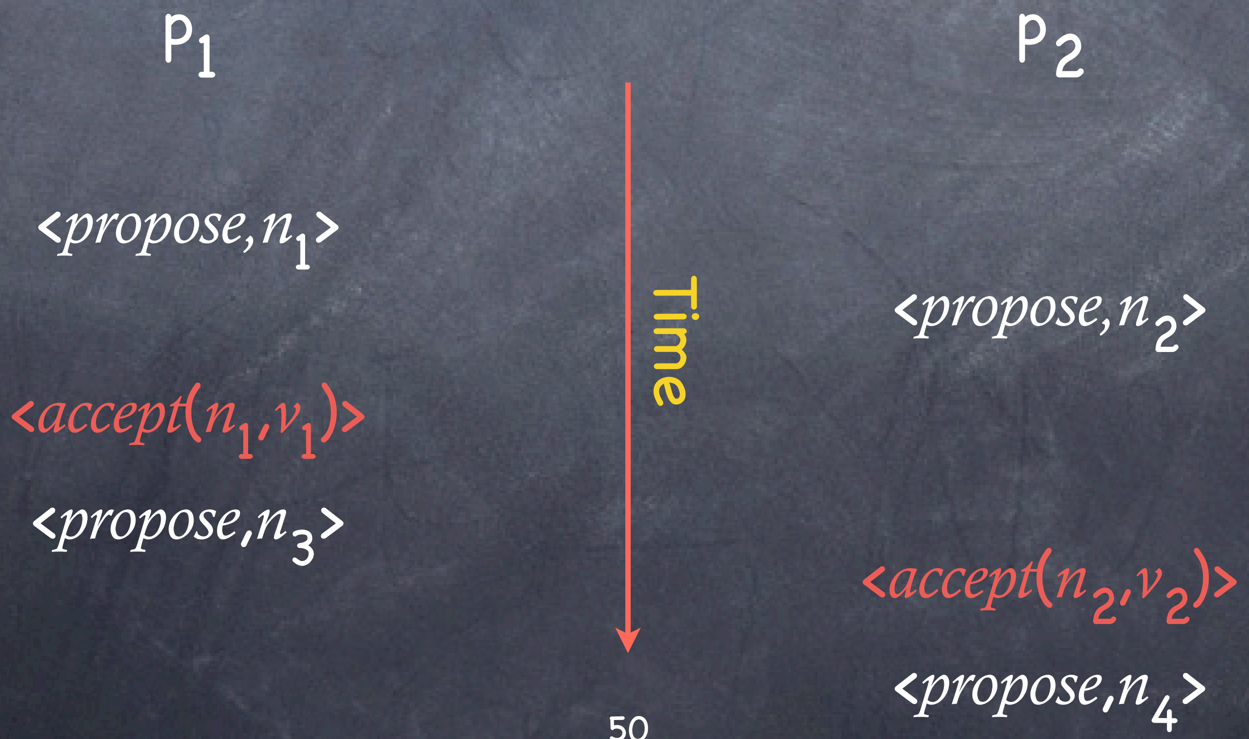
$n_1 < n_2 < n_3 < n_4 < \ldots$


p_1
p_2
Time
$\langle propose, n_1 \rangle$
$\langle propose, n_2 \rangle$
$\langle accept(n_1, v_1) \rangle$
$\langle propose, n_3 \rangle$
$\langle accept(n_2, v_2) \rangle$
$\langle propose, n_4 \rangle$

Liveness

Progress is not guaranteed:

$n_1 < n_2 < n_3 < n_4 < \ldots$



p_1
p_2
Time
$\langle propose, n_1 \rangle$
$\langle propose, n_2 \rangle$
$\langle accept(n_1, v_1) \rangle$
$\langle propose, n_3 \rangle$
$\langle accept(n_2, v_2) \rangle$
$\langle propose, n_4 \rangle$

Delegation

- Paxos is expensive compared to primary/backup; can we get the best of both worlds?
- Paxos group leases responsibility for order of operations to a primary, for a limited period
- If primary fails, wait for lease to expire, then can resume operation (after checking backups)
- If no failures, can refresh lease as needed

Paxos and FLP

- Paxos is always safe–despite asynchrony
- Once a leader is elected, Paxos is live.
- "Ciao ciao" FLP?
  - To be live, Paxos requires a single leader
  - "Leader election" is impossible in an asynchronous system (gotcha!)
- Given FLP, Paxos is the next best thing: always safe, and live during periods of synchrony

Implementing State Machine Replication (RSM)

- Implement a sequence of separate instances of consensus, where the value chosen by the i^{th} instance is the i^{th} message in the sequence.
- Each server assumes all three roles in each instance of the algorithm.
- Assume that the set of servers is fixed

RSM: The role of the leader

- In normal operation, elect a single server to be a leader. The leader acts as the distinguished proposer in all instances of the consensus algorithm.
    - Clients send commands to the leader, which decides where in the sequence each command should appear.
    - If the leader, for example, decides that a client command is the k^{th} command, it tries to have the command chosen as the value in the k^{th} instance of consensus.

RSM: A new leader λ is elected…

- Since λ is a learner in all instances of consensus, it should know most of the commands that have already been chosen. For example, it might know commands 1-10, 13, and 15.
  - It executes phase 1 of instances 11, 12, and 14 and of all instances 16 and larger.
  - This might leave, say, 14 and 16 constrained and 11, 12 and all commands after 16 unconstrained.
  - λ then executes phase 2 of 14 and 16, thereby choosing the commands numbered 14 and 16

RSM: Stop-gap measures

- All replicas can execute commands 1-10, but not 13-16 because 11 and 12 haven't yet been chosen.
- λ can either take the next two commands requested by clients to be commands 11 and 12, or can propose immediately that 11 and 12 be no-op commands.
- λ runs phase 2 of consensus for instance numbers 11 and 12.
- Once consensus is achieved, all replicas can execute all commands through 16.

RSM: To infinity, and beyond

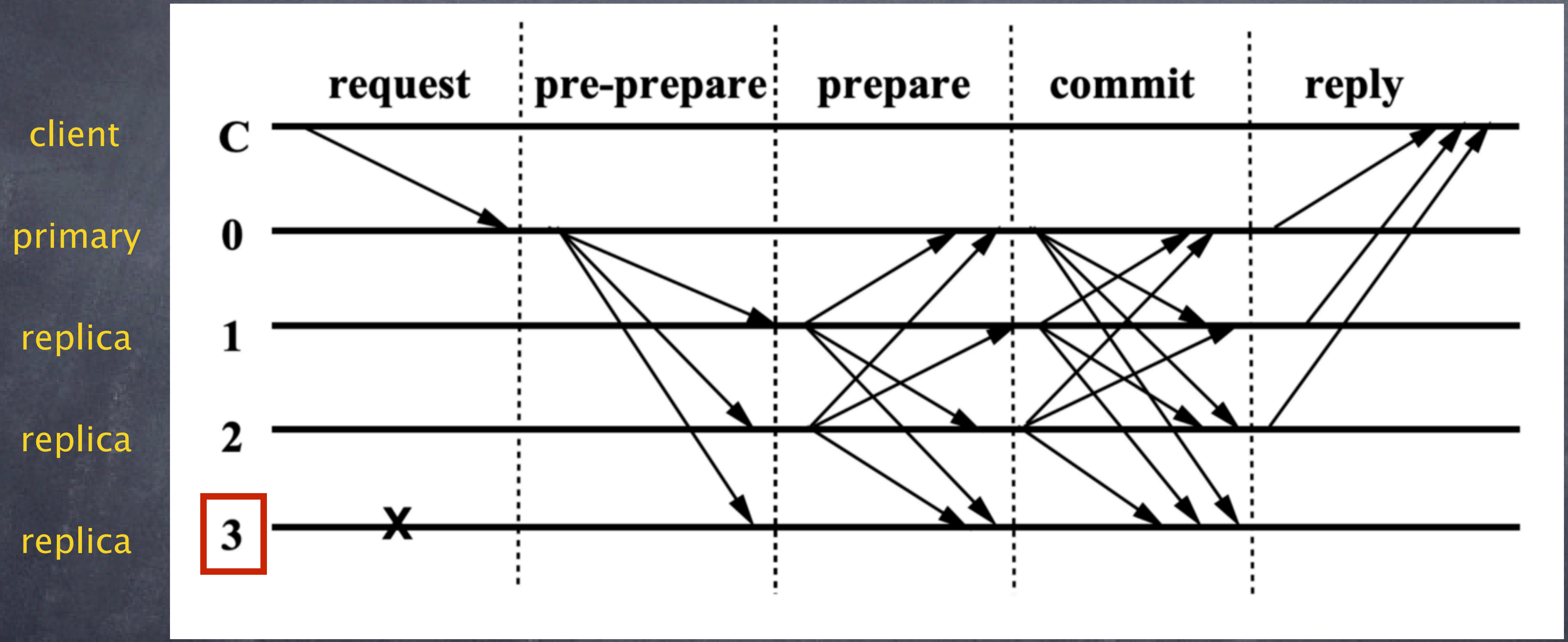
- λ can efficiently execute phase 1 for infinitely many instances of consensus! (e.g. command 16 and higher)
  - λ just sends a message with a sufficiently high proposal number for all instances
  - An acceptor replies non trivially only for instances for which it has already accepted a value

Byzantine Paxos

Clearly, Paxos is easy to corrupt -- if a proposer proposes a different value than what the acceptors returned; or if the acceptor says that a value was accepted when it wasn't, or vice versa.
How do we fix this?

- What if a Paxos node goes rogue? (or two?)
- Solution sketch: instead of just one node in the overlap between majority sets, need more: 2f + 1, to handle f byzantine nodes
  - The extra f+1 outvote the f byzantine nodes, allowing you to make progress.
- Practical Byzantine Fault Tolerance (PBFT) protocol implements this idea

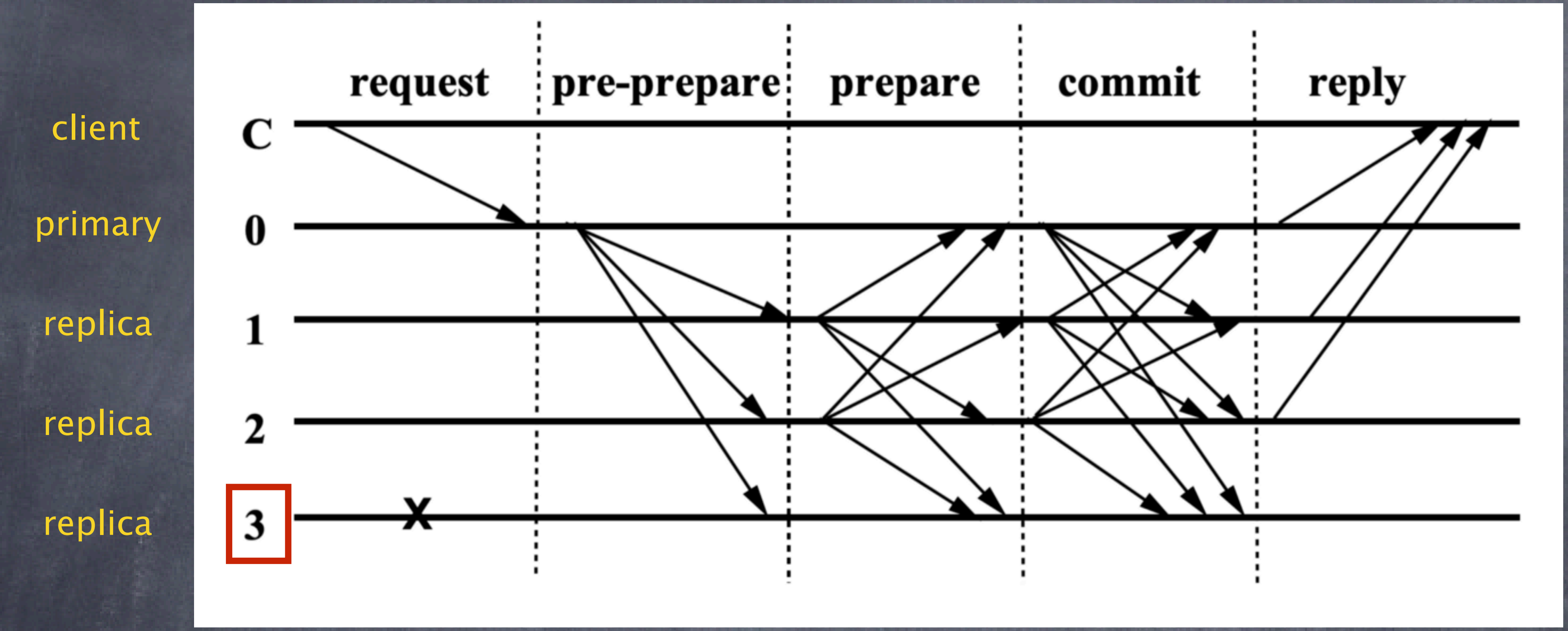
PBFT in one slide



f=1 (byzantine failures)

3f+1 = 4 (minimum nodes in the system to survive f = 1)

PBFT (slightly) explained



- **Request**: the user sends transactions to the primary.
- **Pre-prepare**: the primary produces a proposal containing transactions and forwards to all replicas.
- **Prepare**: Upon receiving a proposal, backups will verify it, and if it succeeds, they will broadcast prepare message to all other replicas. Backups do nothing if verification fails. This is the first round of voting.
- **Commit**: Upon receiving prepare messages from ⅔ of all backups, replicas will now broadcast commit messages. This is the second round of voting.
- **Reply**: the client sees the result of consensus.