Algorithms and Distributed Systems 2019/2020 (Lecture Six)

MIEI - Integrated Master in Computer Science and Informatics
Specialization block

João Leitão (jc.leitao@fct.unl.pt)

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Lecture structure:

- FLP
- Paxos

Last Lecture…

- We have started to study the Consensus Problem
  - C1 Termination: Every correct process eventually decides a value.
  - C2 Validity: If a process decides v, then v was proposed by some process.
  - C3 Integrity: No process decides twice.
  - C4 Uniform Agreement: No two processes decide differently.
- We studied two algorithms for the synchronous system (For **regular** and **uniform** consensus)

Solving Consensus on an Asynchronous System

- What is your best proposal for solving consensus in an asynchronous system where processes may fail (Crash fault model)?

Solving Consensus on an Asynchronous System

- What is your best proposal for solving consensus in an asynchronous system where processes may fail (Crash fault model)?

FLP

M. J. Fisher, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. JACM, Vol. 32, no. 2, April 1985, pp. 374-382

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

FLP

M. J. Fisher, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. JACM, Vol. 32, no. 2, April 1985, pp. 374-382

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL . FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

FLP: Explained

- There is no deterministic protocol that solves consensus in an asynchronous system in which a single process may fail by crashing.

FLP: Explained

- There is no deterministic protocol that solves consensus in an asynchronous system in which a single process may fail by crashing.

- How do we demonstrate this?

FLP: Explained

- There is no deterministic protocol that solves consensus in an asynchronous system in which a single process may fail by crashing.

- How do we demonstrate this?

- By contradiction and through an indistinguishability argument.

System trace

- A system trace is a way to model the execution of a distributed system considering only its *externally observable behaviour* where:
  - Only inputs and outputs are considered.
  - We fully abstract the internal state of each process.

- Notation is usually: Process Identifier: Action
- E.g.:
  - P1: Proposes(v), P2:Proposes(v’), P1:Decides(v), P2: Decides(v)

FLP: Explained

- Let’s consider two sets of processes of arbitrary size (with at least one process): A and B
- Now let’s assume that there exists a deterministic algorithm that solves consensus.
- Let’s build a few traces of the execution of such system (in an asynchronous system under the crash fault model).

FLP: Explained

The Proof itself involves an initial step that is to prove that the output of consensus must depend on the ordering of messages exchanged among processes.
This can be done by leveraging the property:

C2 Validity: If a process decides v, then v was proposed by some process.

FLP: Explained

Regular Consensus Specification:

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
C4 Agreement: No two correct processes decide differently.

- Run One:
  - All processes in B crash at time $t_{0.}$
  - All processes in A propose *v* at time t_1 ($t_1 > t_0$).

FLP: Explained

Regular Consensus Specification:

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
C4 Agreement: No two correct processes decide differently.

- Run One:
  - All processes in B crash at time $t_{0.}$
  - All processes in A propose *v* at time t_1 ($t_1 > t_0$).
  - All processes in A decide v at some time t2 ($t_2 > t_1$)

FLP: Explained

Regular Consensus Specification:

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
C4 Agreement: No two correct processes decide differently.

- Run One:
  - All processes in B crash at time $t_{0.}$
  - All processes in A propose *v* at time t_1 ($t_1 > t_0$).
  - All processes in A decide v at some time t2 ($t_2 > t_1$)

- Trace: B:Crash(), A:Propose(v), A:Decide(v)

FLP: Explained

Regular Consensus Specification:

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
C4 Agreement: No two correct processes decide differently.

- Run Two:
  - All processes in A crash at time $t_{0.}$
  - All processes in B propose *v'* at time t_1 ($t_1 > t_0$).

FLP: Explained

Regular Consensus Specification:

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
C4 Agreement: No two correct processes decide differently.

- Run Two:
  - All processes in A crash at time $t_{0.}$
  - All processes in B propose *v'* at time t_1 ($t_1 > t_0$).
  - All processes in B decide v' at some time t_3 ($t_3 > t_1$)

FLP: Explained

Regular Consensus Specification:

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
C4 Agreement: No two correct processes decide differently.

- Run Two:
  - All processes in A crash at time $t_{0.}$
  - All processes in B propose *v'* at time t_1 ($t_1 > t_0$).
  - All processes in B decide v' at some time t_3 ($t_3 > t_1$)

- Trace: A:Crash(), B:Propose(v'), B:Decide(v')

FLP: Explained

Regular Consensus Specification:

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
C4 Agreement: No two correct processes decide differently.

- Run Three:
  - Messages between processes in A and B are delayed up to some time t_5 ($t_5 > t_4$ and t5 > t_3).
  - All processes in A propose v at some time t_1.
  - All processes in B propose v’ at some time t_1 (v != v’).

FLP: Explained

Regular Consensus Specification:

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
C4 Agreement: No two correct processes decide differently.

- Run Three:
  - Messages between processes in A and B are delayed up to some time t_5 ($t_5 > t_4$ and t5 > t_3).
  - All processes in A propose v at some time t_1.
  - All processes in B propose v’ at some time t_1 (v != v’).
  - By indistinguishability with Run One processes in A decide v at some time t_2 ($t_2 > t_1$).
  - By indistinguishability with Run Two processes in B decide v’ at some time t_3 ($t_3 > t_1$).

FLP: Explained

Regular Consensus Specification:

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
C4 Agreement: No two correct processes decide differently.

- Run Three:
  - Messages between processes in A and B are delayed up

Trace:

A:Propose(v), B:Propose(v'), A:Decide(v), B:Decide(v')

  - By indistinguishability with Run One processes in A decide v at some time t_2 ($t_2 > t_1$).
  - By indistinguishability with Run Two processes in B decide v' at some time t_3 ($t_3 > t_1$).

FLP: Explained

Regular Consensus Specification:

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
~~C4 Agreement: No two correct processes decide differently.~~

- Run Three:
  - Messages between processes in A and B are delayed up

Trace:

A:Propose(v), B:Propose(v'), A:Decide(v), B:Decide(v')

  - By indistinguishability with Run One processes in A decide v at some time t_2 ($t_2 > t_1$).
  - By indistinguishability with Run Two processes in B decide v' at some time t_3 ($t_3 > t_1$).

FLP: Explained

Regular Consensus Specification:

C1 Termination: Every correct process eventually decides a value.
C2 Validity: If a process decides v, then v was proposed by some process.
C3 Integrity: No process decides twice.
~~C4 Agreement: No two correct processes decide differently.~~

Trace:

A:Propose(v), B:Propose(v'), A:Decide(**v**), B:Decide(**v'**)

Contradition: Hence our base assumption that there is a deterministic algorithm that solves consensus (in asynchronous systems where a process can crash) must be false.

decide v' at some time t_3 ($t_3 > t_1$).

FLP: Secondary Result (Positive Result)

- There are deterministic protocols that solve consensus in an asynchronous system when no process crashes during the execution of the algorithm.

Consequences of FLP

- Consensus is not solvable (by a deterministic algorithm) in asynchronous systems under the crash fault model.
- What about equivalent problems to consensus?

Consequences of FLP

- A problem that has been demonstrated to be equivalent to consensus: Total Order Broadcast

- Total Order Broadcast Specification:
  - TO (Total Order): Let m_1 and m_2 be any two messages. Let p_i and p_j be any two correct processes that deliver m_1 and m_2. If p_i delivers m_1 before m_2, then p_j delivers m_1 before m_2.
  - RB1 (Validity): If a correct process i broadcasts message m, then i eventually delivers the message.
  - RB2 (No Duplications): No message is delivered more than once.
  - RB3 (No Creation): If a correct process j delivers a message m, then m was broadcast to j by some process i.
  - RB4 (Aggrement): If a message m is delivered by some correct process i, them m is eventually delivered by every correct process j.

Consequences of FLP

- A problem that has been demonstrated to be equivalent to consensus: Total Order Broadcast

- Total Order Broadcast Specification:
  - **TO (Total Order): Let m_1 and m_2 be any two messages. Let p_i and p_j be any two correct processes that deliver m_1 and m_2. If p_i delivers m_1 before m_2, then p_j delivers m_1 before m_2.**
  - RB1 (Validity): If a correct process i broadcasts message m, then i eventually delivers the message.
  - RB2 (No Duplications): No message is delivered more than once.
  - RB3 (No Creation): If a correct process j delivers a message m, then m was broadcast to j by some process i.
  - RB4 (Aggrement): If a message m is delivered by some correct process i, them m is eventually delivered by every correct process j.

Consequences of FLP

- A problem that has been demonstrated to be equivalent to consensus: Total Order Broadcast

- Equivalent implies that:
  - If you have consensus, then you can solve the total order broadcast problem.
  - If you have total order broadcast, then you can solve the consensus problem.

Consequences of FLP

- A problem that has been demonstrated to be equivalent to consensus: Total Order Broadcast

- Equivalent implies that:
  - If you have consensus, then you can solve the total order broadcast problem.
  - If you have total order broadcast, then you can solve the consensus problem.
- Bad News: Since the problems are equivalent the FLP result also applies to Total Order Broadcast.

Consequences of FLP

- State machine replication requires either Consensus or Total Order Broadcast (trivial to demonstrate the second, since they are equivalent ☺).

The World is asynchronous…

...so there goes state machine replication down the drain!

Consequences of FLP

And so are distributed systems in practice?

Consequences of FLP

- So this is done right?

- We cannot do anything… maybe lets cancel this course... Go home... And think about the vacum of Human and distributed systems existence?

Consequences of FLP

- So this is done right?

- We cannot do anything… maybe lets cancel this course... Go home... And think about the vacum of Human and distributed systems existence?

- Not so fast: We can always **circumvent (i.e, go around)** the impossibility result.
- But how?

Circumventing FLP:

- By relaxing the specification of Consensus obviously…

Circumventing FLP:

- By relaxing the specification of Consensus obviously…
- C1 Termination: Every correct process eventually decides a value.
- C2 Validity: If a process decides v, then v was proposed by some process.
- C3 Integrity: No process decides twice.
- C4 Agreement: No two correct processes decide differently.

Circumventing FLP:

- By relaxing the specification of Consensus obviously…
- ~~C1 Termination: Every correct process eventually decides a value.~~
- C2 Validity: If a process decides v, then v was proposed by some process.
- C3 Integrity: No process decides twice.
- C4 Agreement: No two correct processes decide differently.

- Technique 1: Let’s use a probabilistic algorithm that ensures termination with high probability (but not for sure).

Circumventing FLP:

- By relaxing the specification of Consensus obviously…
- C1 Termination: Every correct process eventually decides a value.
- ~~C2 Validity: If a process decides v, then v was proposed by some process.~~
- C3 Integrity: No process decides twice.
- ~~C4 Agreement: No two correct processes decide differently.~~

- Technique 2: Let's relax the agreement and validity properties such that there must exist proposed values by at least k (k<n) processes, and we only decide values with this property.

Circumventing FLP:

- By relaxing the specification of Consensus obviously…
- C1 Termination: Every correct process eventually decides a value.
- C2 Validity: If a process decides v, then v was proposed by some process.
- C3 Integrity: No process decides twice.
- C4 Agreement: No two correct processes decide differently.

- Technique 3: Let’s modify our system model to say that there is this “magic abstraction” that allows us to detect process failures (**failure detectors**).
  (Interesting question: what are the minimum guarantees that a fault detector has to provide for consensus to be solvable?)

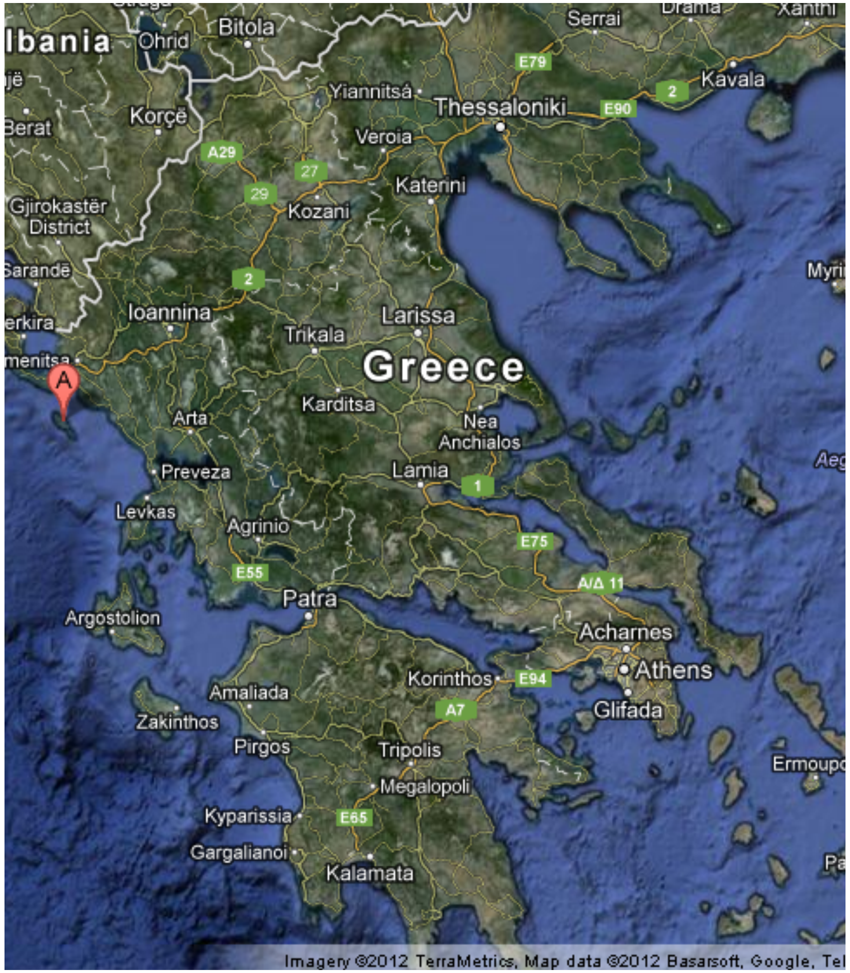
Circumventing FLP:

- By relaxing the specification of Consensus obviously…
- ~~C1 Termination: Every correct process eventually decides a value.~~
- C2 Validity: If a process decides v, then v was proposed by some process.
- C3 Integrity: No process decides twice.
- C4 Agreement: No two correct processes decide differently.

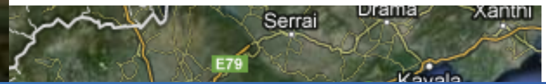
- Technique 4: Let's relax the termination property such that we "only ensure" termination if the system behaves in a synchronous way (so no termination at all).

Exploring alternative 4: Paxos

- Solves (a weaker variant of) Consensus in asynchronous systems under crash fault model.
- Termination can only be achieved in periods where the system behaves in a synchronous way.
- Used in practice: Google, Yahoo!, Microsoft, Amazon, etc.



Exploring alternative 4: Paxos



- Used in practic
  Google, Yahoo
  Microsoft, Am

Exploring alternative 4: Paxos

Paxos: a brief history

- Leslie Lamport. **1998**. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133-169.

Paxos: a brief history

- Leslie Lamport. **1998**. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133-169.
- Leslie Lamport. Paxos made simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December **2001**)

Paxos: a brief history

- Leslie Lamport. **1998**. The part-time

At the PODC 2001 conference, I got tired of everyone saying how difficult it was to understand the Paxos algorithm, published in [122]. Although people got so hung up in the pseudo-Greek names that they found the paper hard to understand, the algorithm itself is very simple. So, I cornered a couple of people at the conference and explained the algorithm to them orally, with no paper. When I got home, I wrote down the explanation as a short note, which I later revised based on comments from Fred Schneider and Butler Lampson. The current version is 13 pages long, and contains no formula more complicated than n1 > n2.

-- From Leslie Lamport page.

Paxos: a brief history

- Leslie Lamport. **1998**. The part-time parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133-169.
- Leslie Lamport. Paxos made simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December **2001**)
- Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *ACM Comput. Surv.* 47, 3, Article 42 (February **2015**), 36 pages.

Paxos: Consensus Specification

- C2 Validity: If a process decides v, then v was proposed by some process.
- C3 Integrity: No process decides twice.
- C4 Agreement: No two correct processes decide differently.

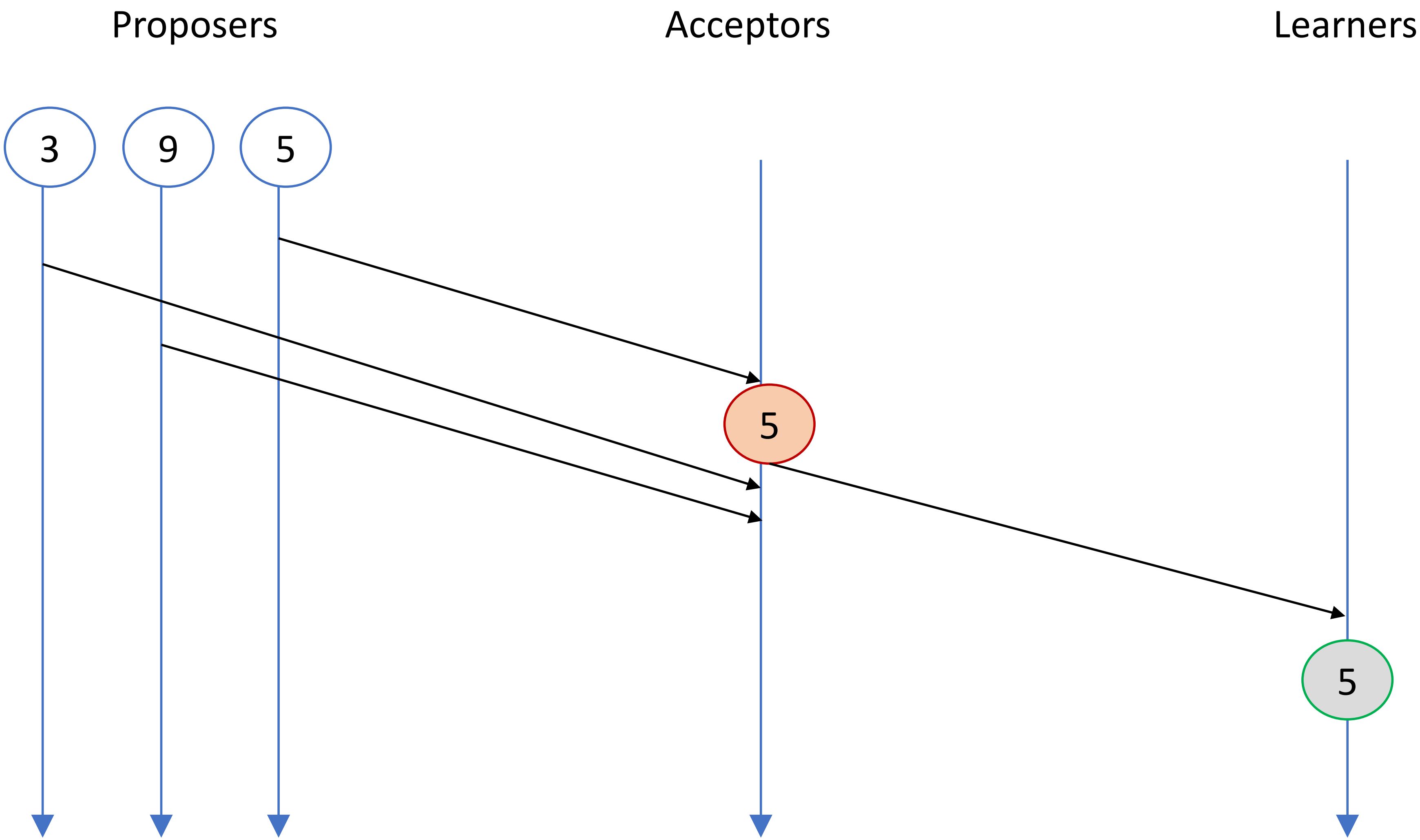
Paxos: Assumptions

- Asynchronous System
- Messages exchanged among processes can be lost, duplicated, but never corrupted.
- Processes can fail by crash and recover at some point in the future (crash-recovery fault model).
  - Each process has access to persistent storage (e.g., hard disk) that ’survives’ to a crash.

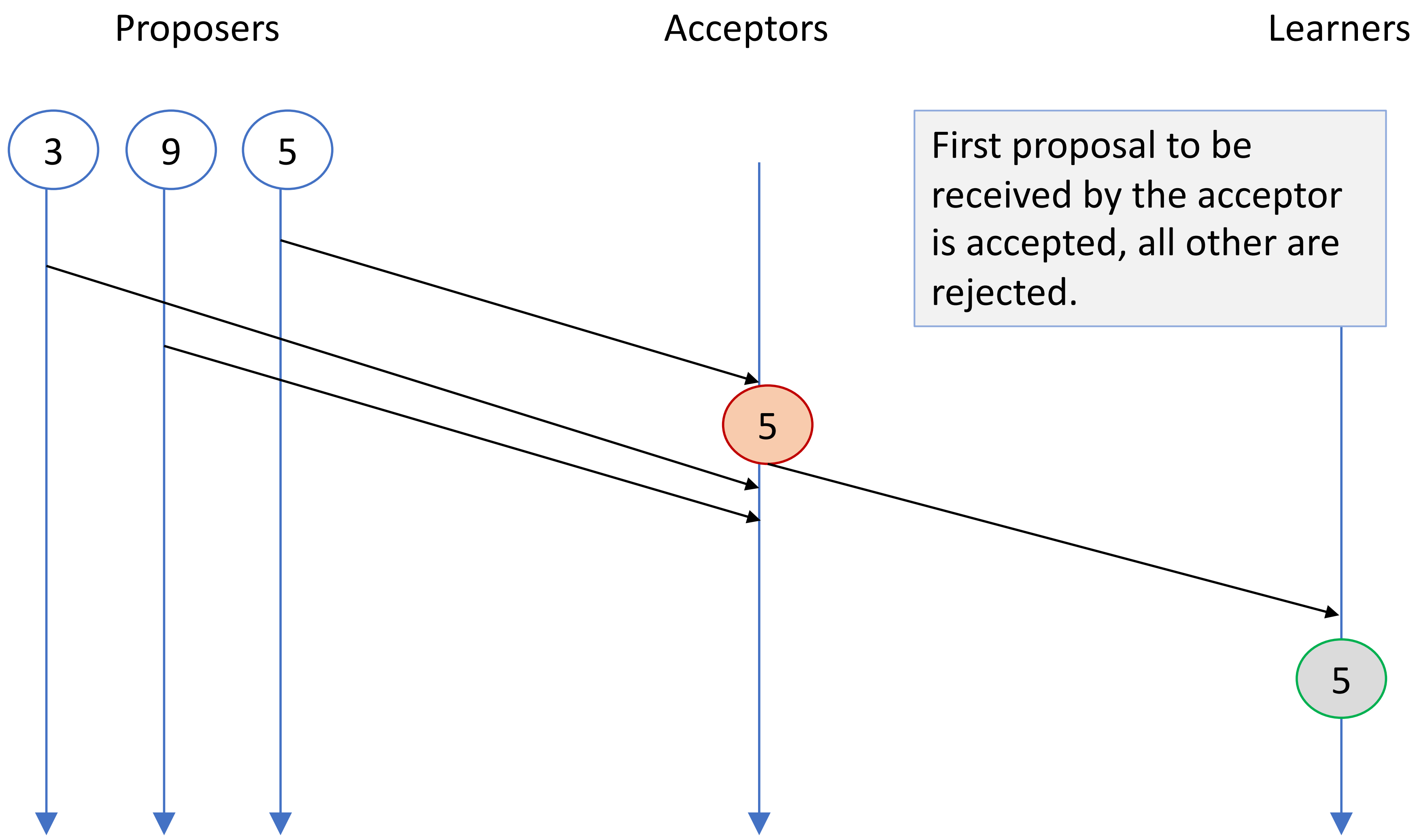
Paxos: Separation of Roles

- 3 different types of processes:
  - Proposers: Propose values
  - Acceptors: Accept proposed values.
  - Learners: Learn decided values.
- In practice these three roles can be executed together in a single machine
  - A process can have and execute all three roles simultaneously.

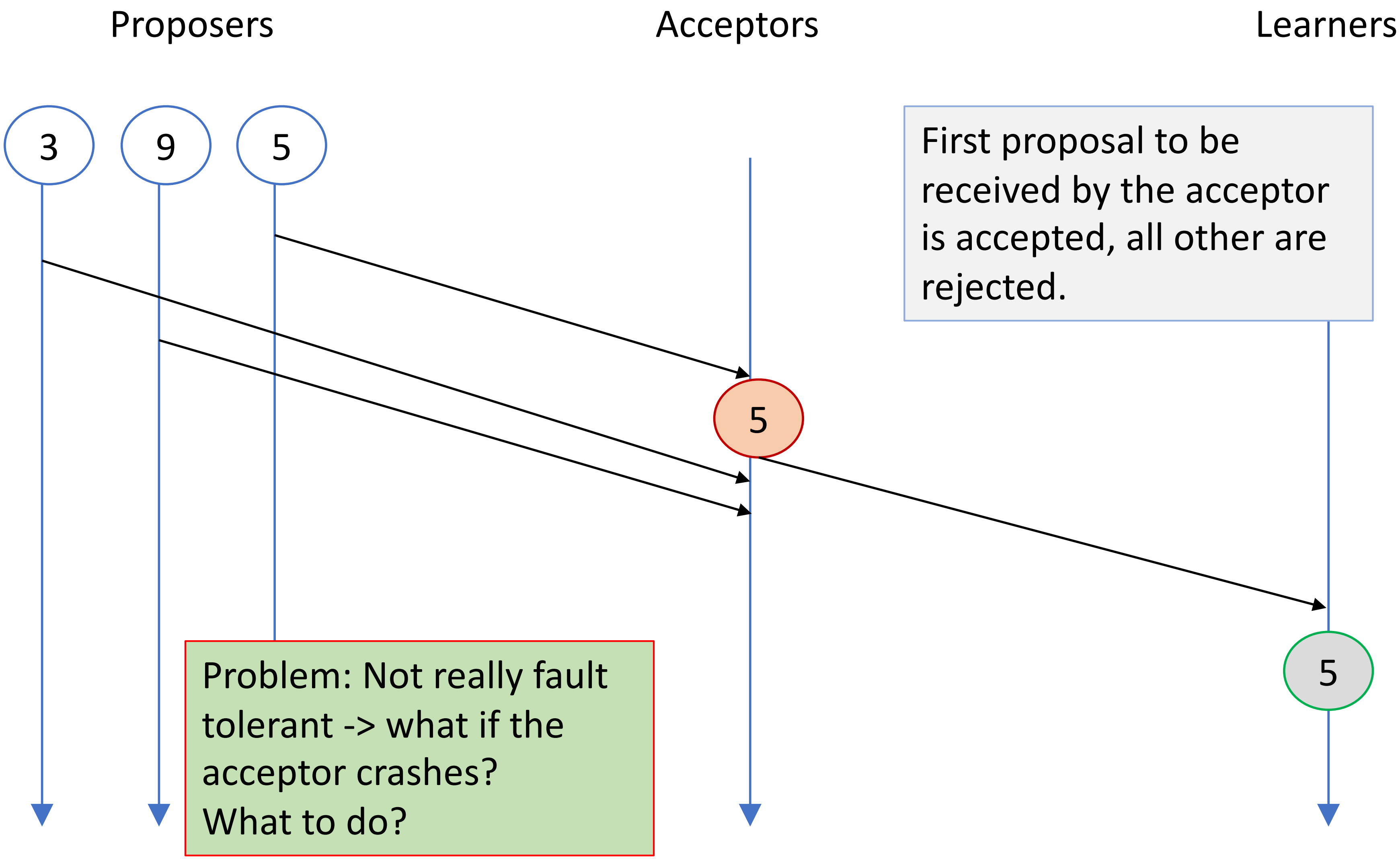
Trivial Solution: Single Acceptor



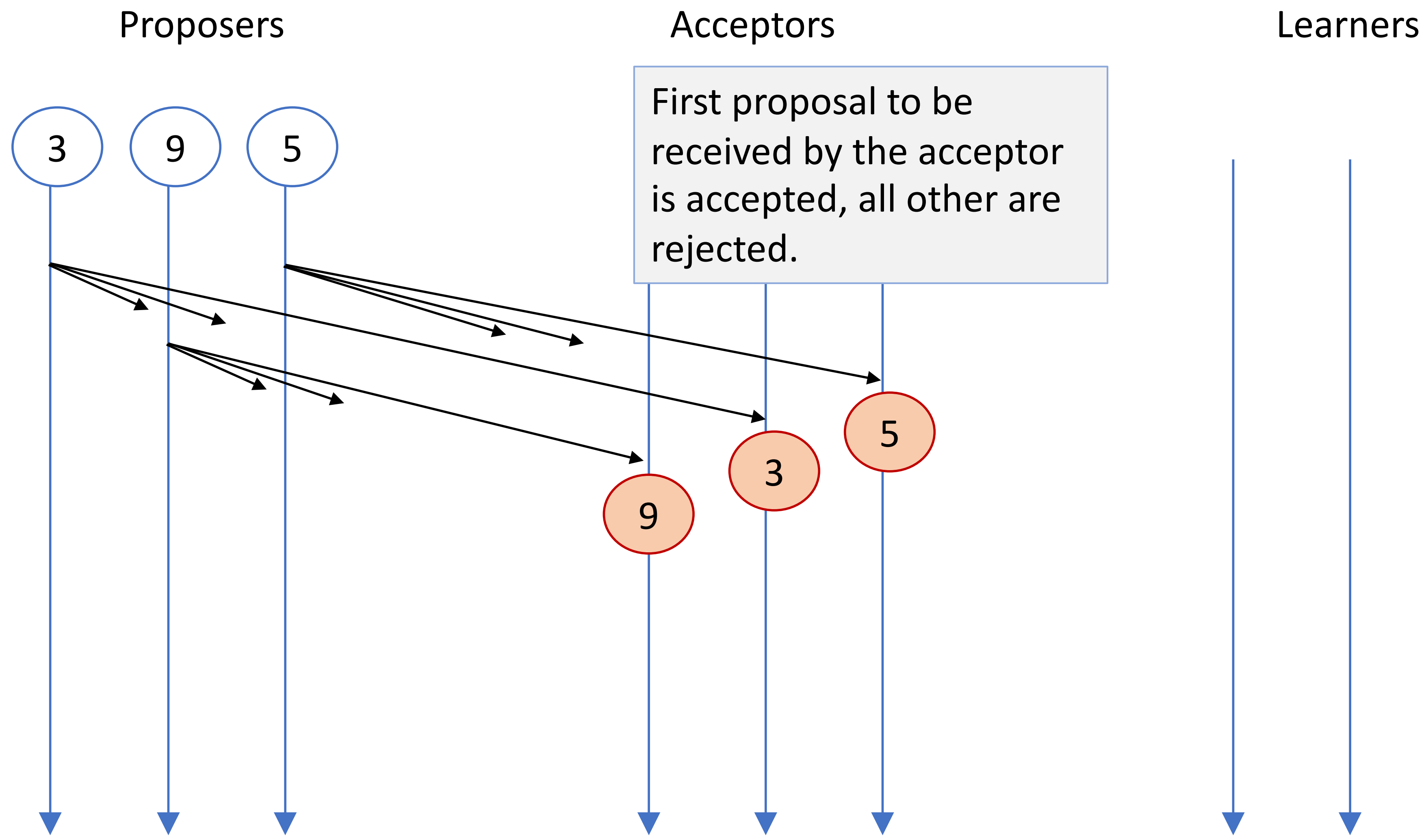
Trivial Solution: Single Acceptor



Trivial Solution: Single Acceptor



Paxos: Tolerating failure of the Acceptor (Through Replication)



When can processes decide?

- Problem: different acceptors might receive proposals in a different order, so when can they make a “final” decision?

When can processes decide?

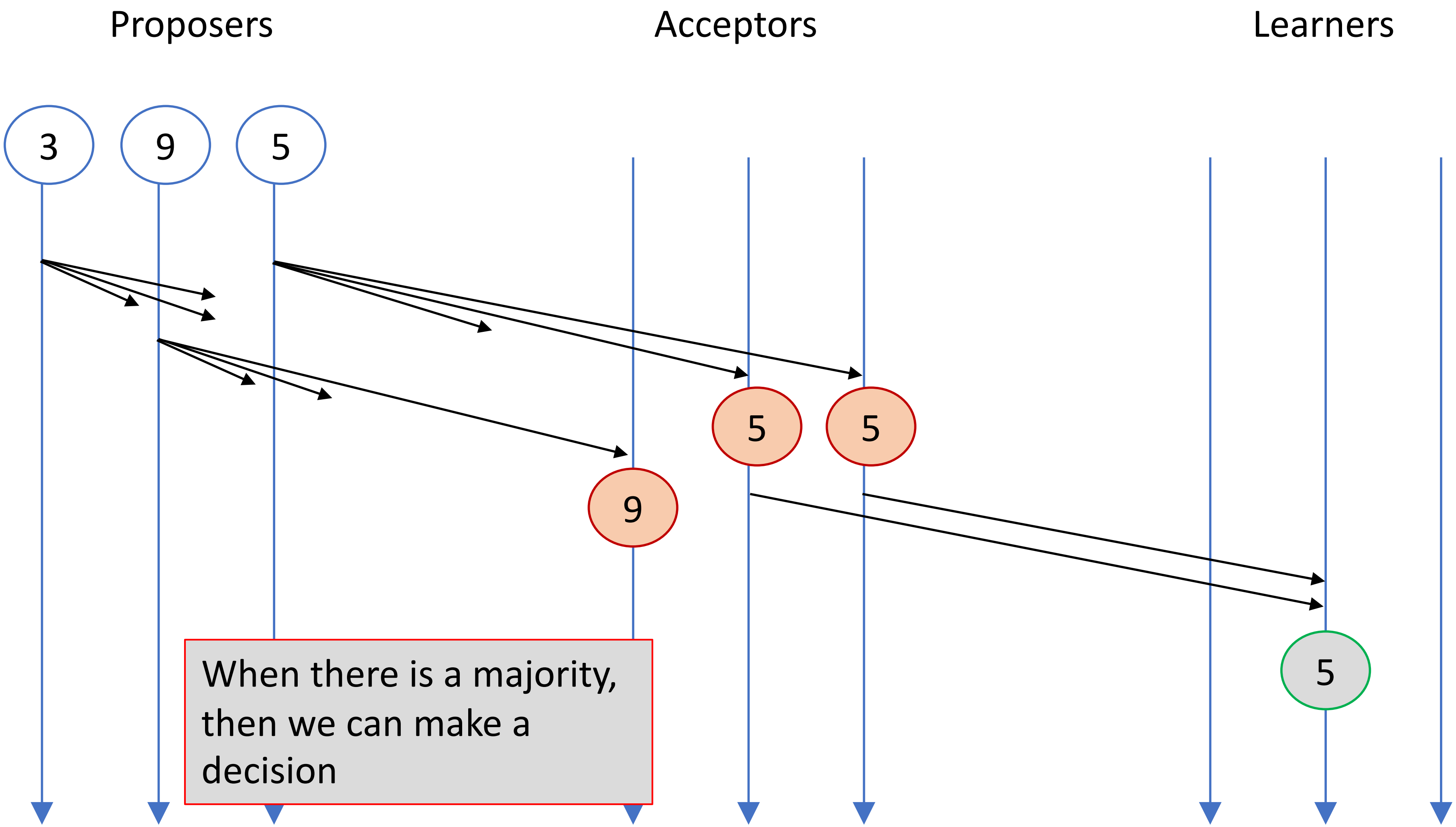
- Problem: different acceptors might receive proposals in a different order, so when can they make a “final” decision?
- Suggestion: Lets think in this in terms of quorums…

When can processes decide?

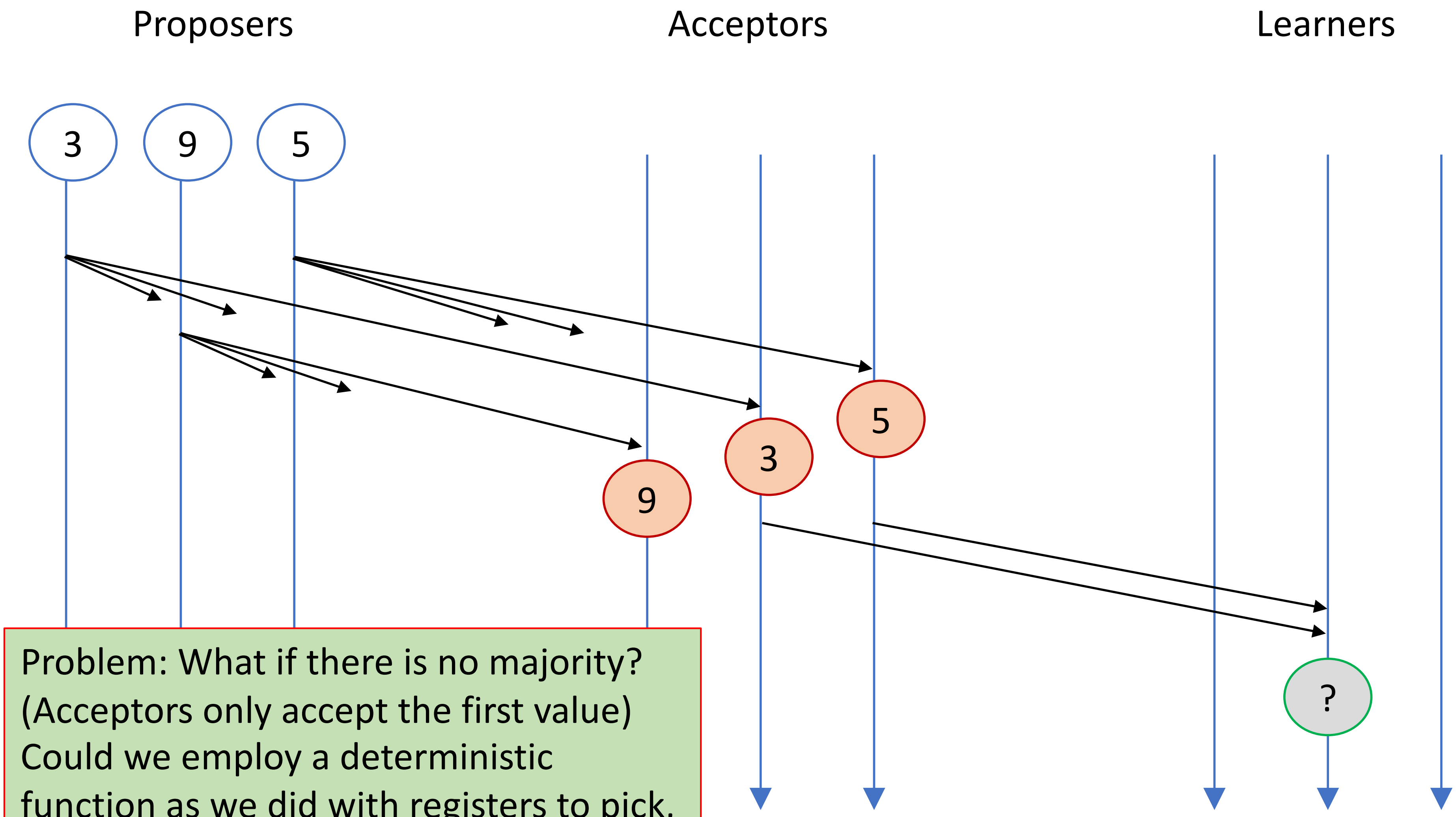
- Problem: different acceptors might receive proposals in a different order, so when can they make a “final” decision?
- Suggestion: Lets think in this in terms of quorums…

- Decision is made when a majority accepts the same value, this will ensure an intersection as in a majority-based quorum.

Paxos: Tolerating failure of the Acceptor (Quorum)



Paxos: Tolerating failure of the Acceptor (Quorum)



Problem: What if there is no majority?
(Acceptors only accept the first value)
Could we employ a deterministic function as we did with registers to pick, for instance the largest value?

How to deal with multiple (concurrent) proposals?

How to deal with multiple (concurrent) proposals?

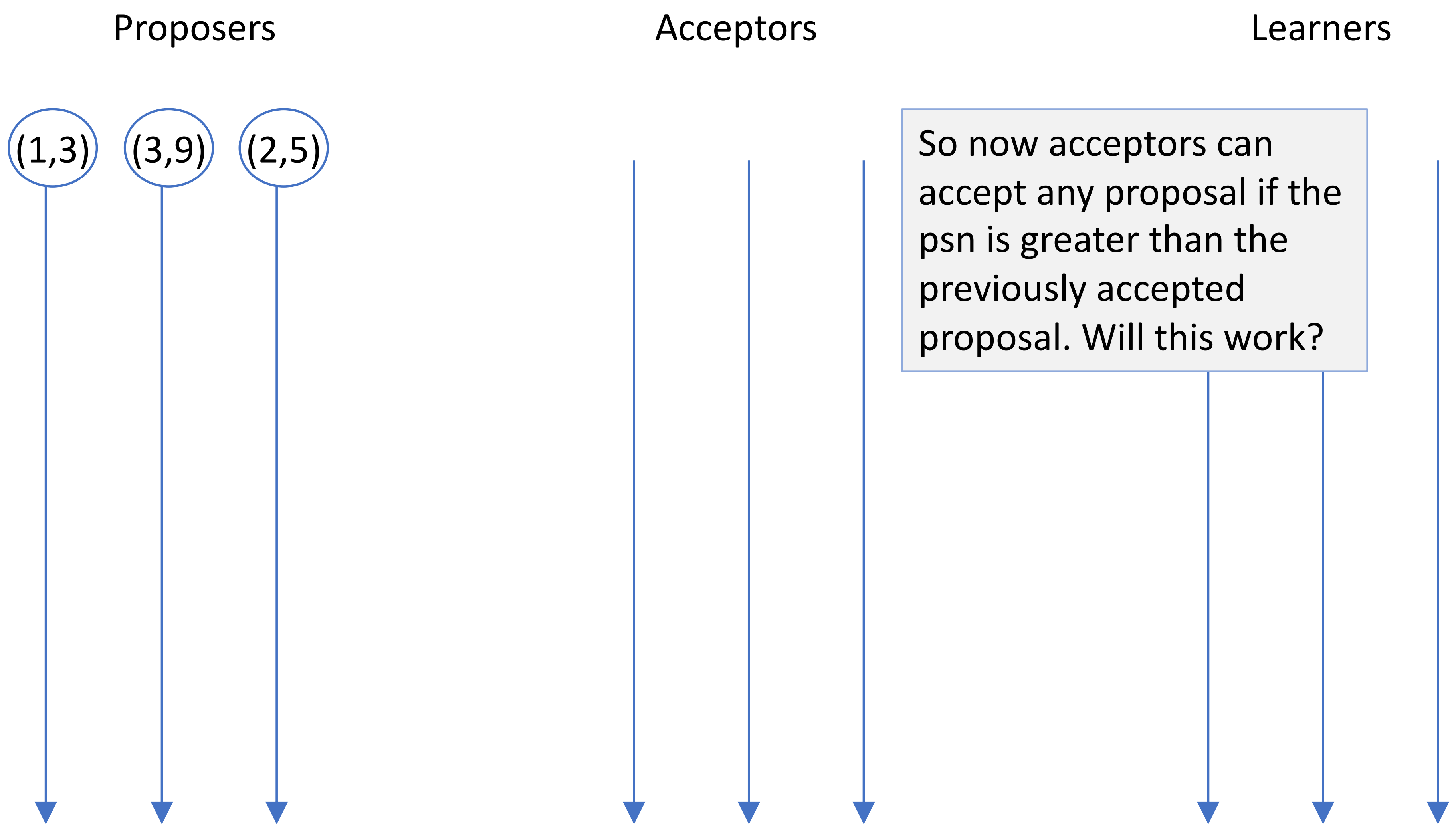
- Acceptors must be able to accept more than one proposal (i.e., more than one value, meaning that they might change their opinion regarding the value that is going to be decided).

How to deal with multiple (concurrent) proposals?

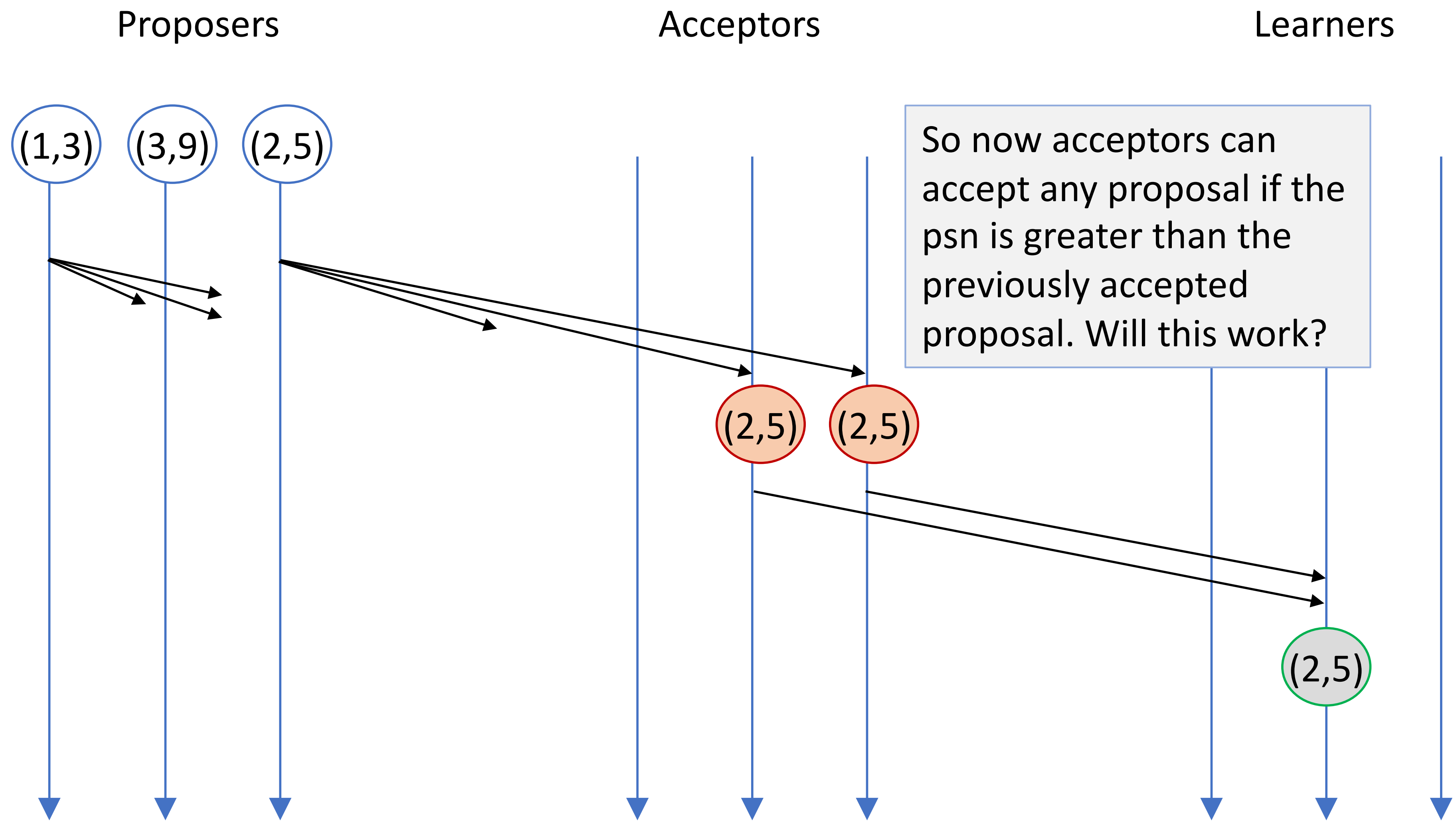
- Acceptors must be able to accept more than one proposal (i.e., more than one value, meaning that they might change their opinion regarding the value that is going to be decided).

- Each proposal will be enriched with a sequence number that allows to distinguish different proposal and order them.
  - Proposal = (psn, value)
  - All proposals have a different proposal sequence number (psn)
  - Definition: a proposal (meaning a pair (psn, value)) is considered **selected** when it is accepted by a majority of acceptors (f < N/2).
    - At this point learns can declare the decided value.
- Sequence numbers can be generated by each proposal by using its identifiers (1..N, where N is the number of proposers) and adding N whenever it needs a new sequence number.

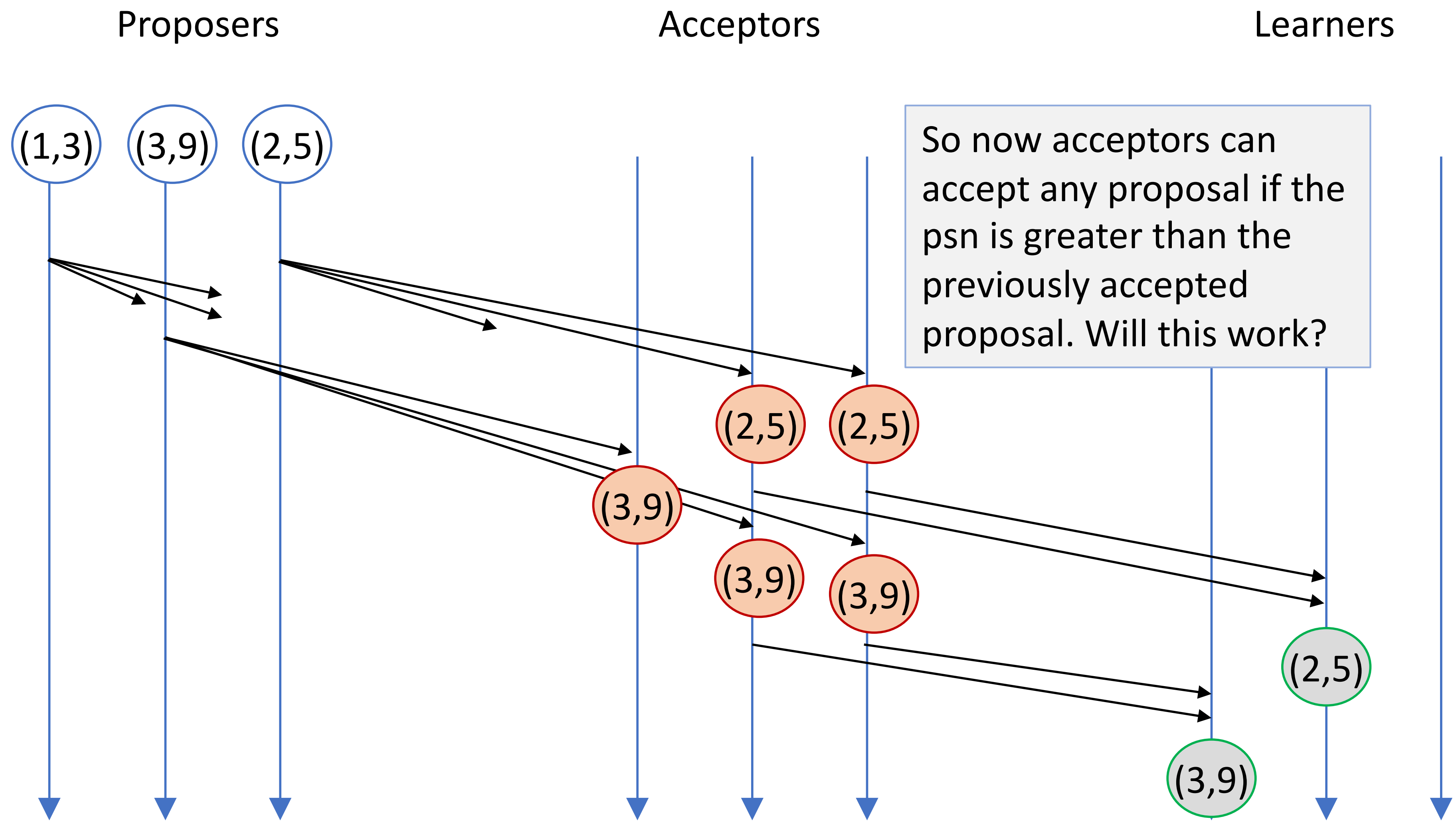
Paxos: Tolerating failure of the Acceptor (Quorum)



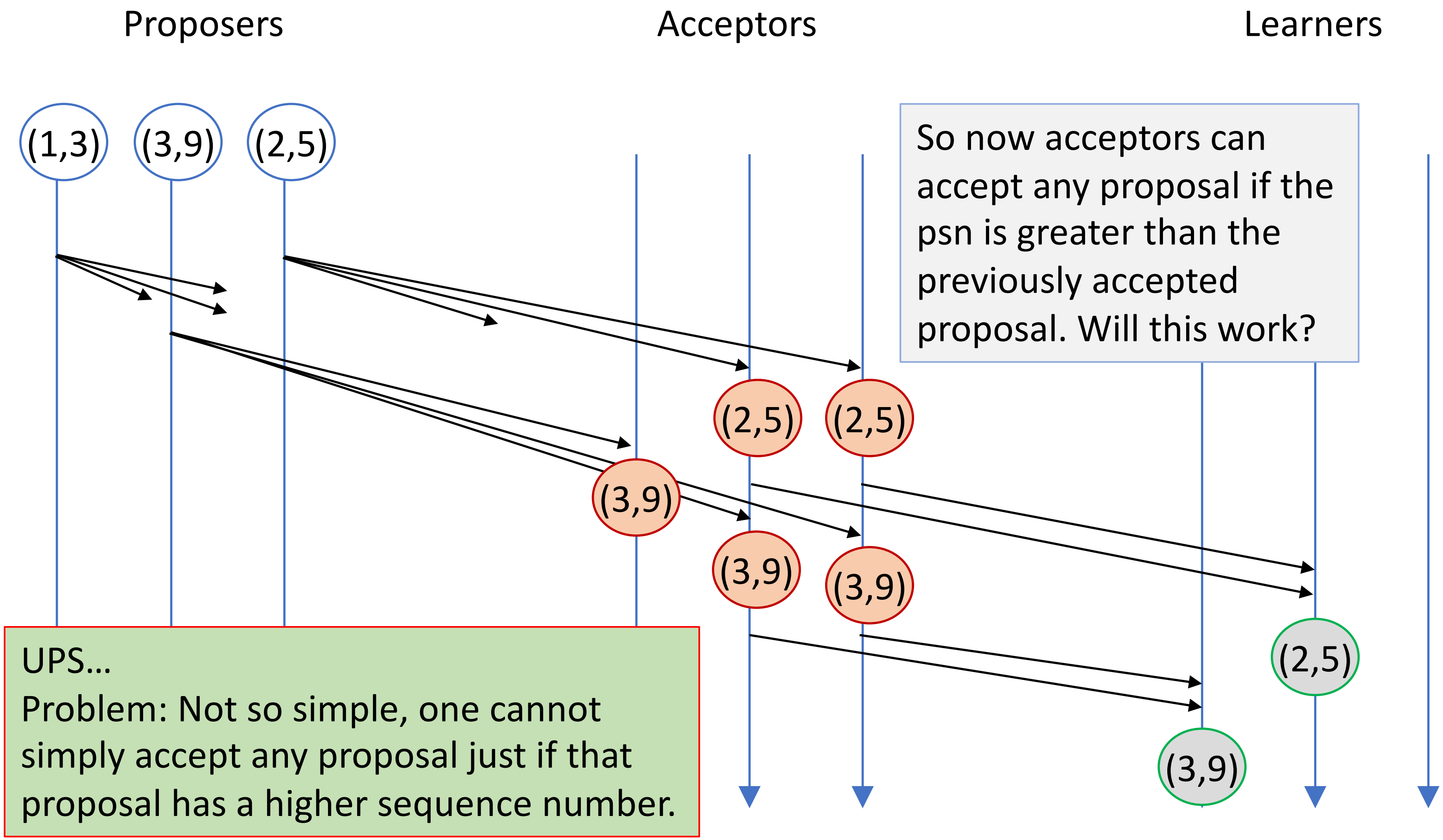
Paxos: Tolerating failure of the Acceptor (Quorum)



Paxos: Tolerating failure of the Acceptor (Quorum)



Paxos: Tolerating failure of the Acceptor (Quorum)



When should an acceptor change its accepted proposal?

When should an acceptor change its accepted proposal?

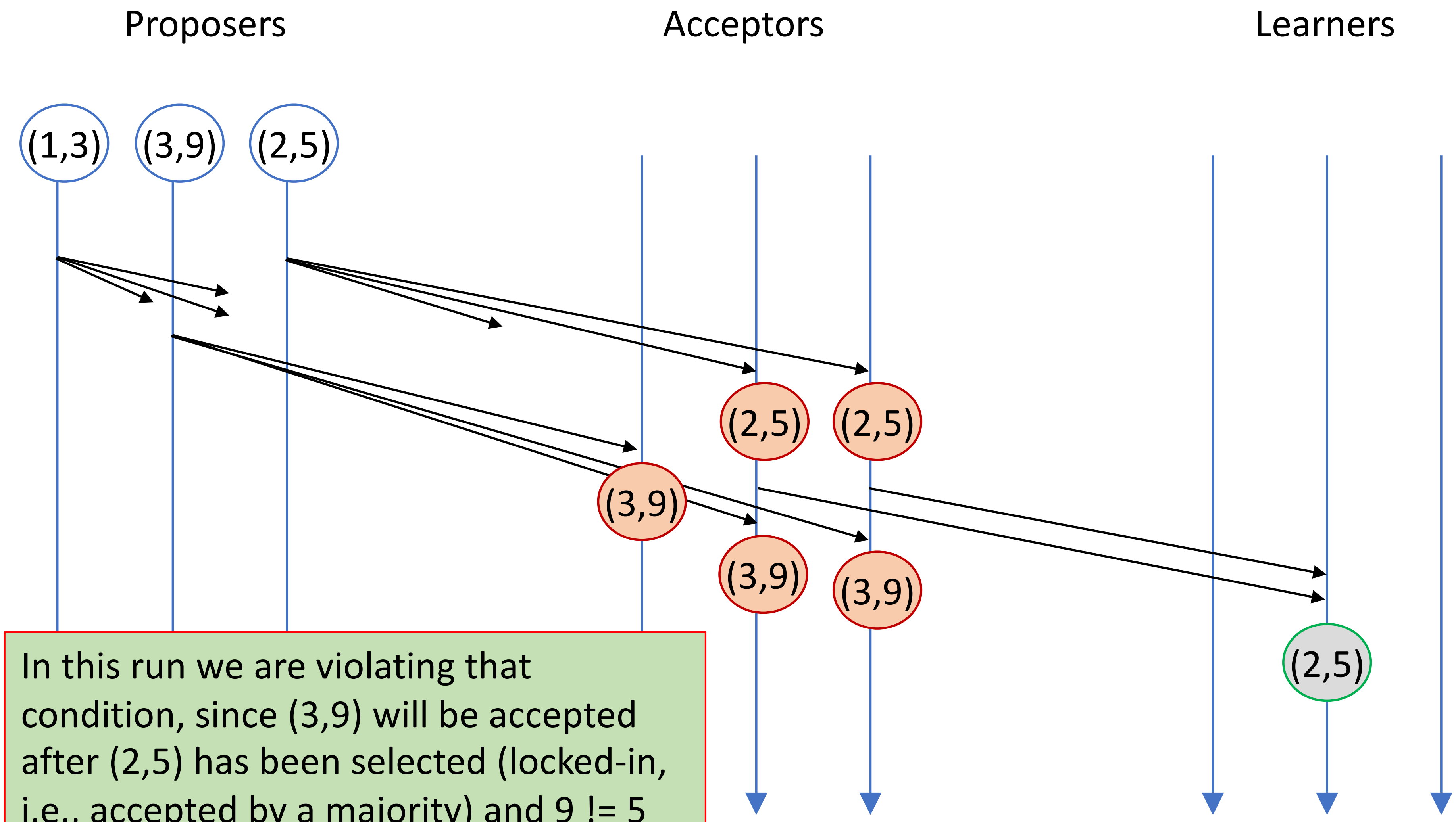
- If an acceptor changes its currently accepted proposal just because it receives a proposal with a higher sequence number, more than one value can be decided (violating the definition of consensus).

When should an acceptor change its accepted proposal?

- If an acceptor changes its currently accepted proposal just because it receives a proposal with a higher sequence number, more than one value can be decided (violating the definition of consensus).

- The trick is to avoid an acceptor to change its accepted proposal if there is already a proposal that was accepted by a majority (and hence decided).
- A value in a proposal that was accepted by a majority of acceptors is said to be **locked-in**.

Problelm of acceptors changing previously accepted proposal.



In this run we are violating that condition, since (3,9) will be accepted after (2,5) has been selected (locked-in, i.e., accepted by a majority) and 9 != 5

What is the solution for this?

- Intuitively, an acceptor could safely accept another proposal if the sequence number of that proposal is higher than the previously accepted proposal and:
  1. If there is no previously locked-in value, then that proposal can propose any value to be decided by consensus.
  2. If there is already a locked-in value *v*, then the new proposal also proposes *v* (this will ensure that two different values cannot be decided).

What is the solution for this?

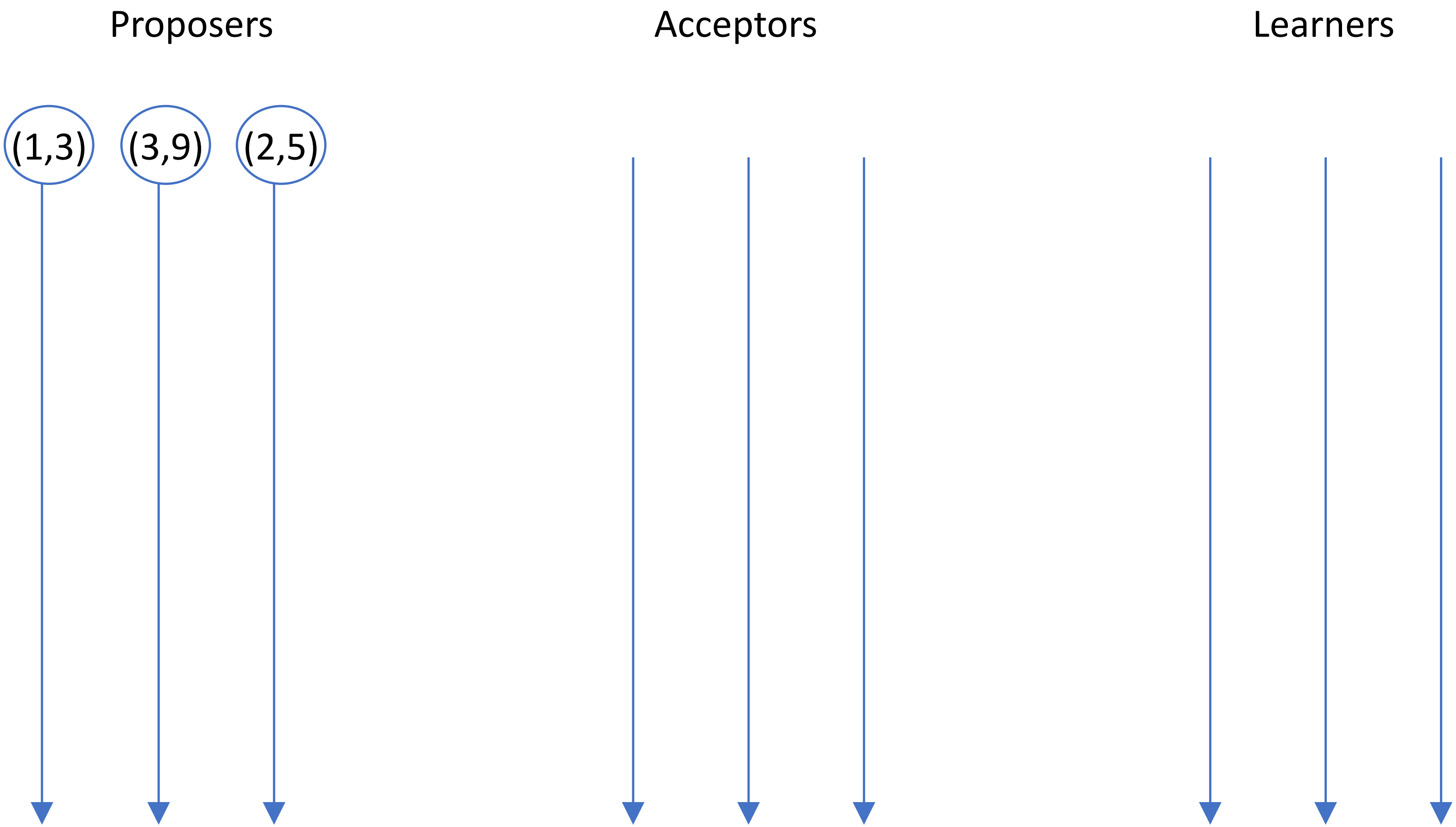
- We must delegate in the proposers the responsibility to check if a value has already been locked-in before they make their propose.
- *If no value has been locked-in, then the proposer can propose its initial value to be decided.*
- *If some value v has been already locked-in, then the proposer **must** propose that value.*

What is the solution for this?

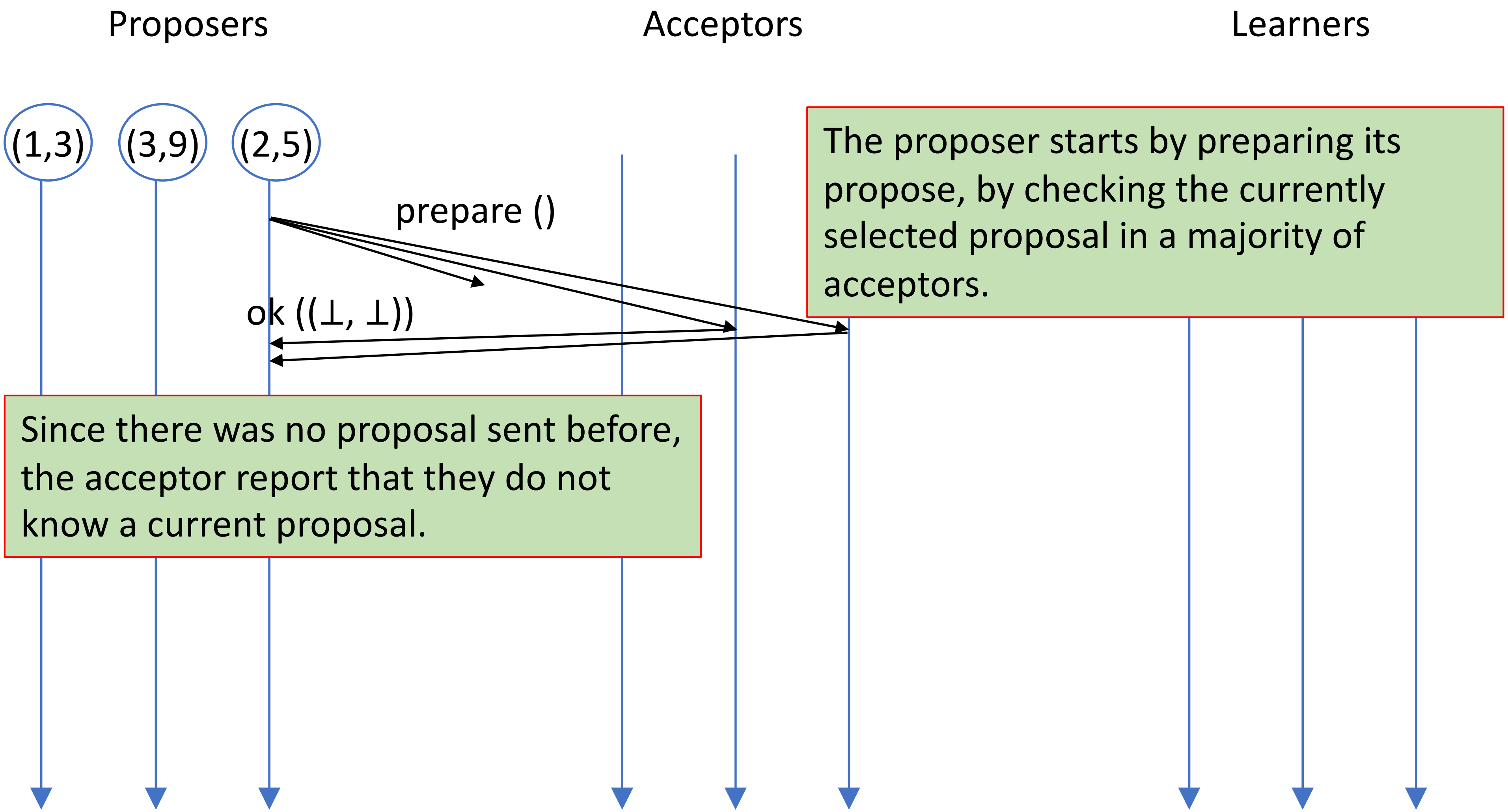
- We must delegate in the proposers the responsibility to check if a value has already been locked-in before they make their propose.
- *If no value has been locked-in, then the proposer can propose its initial value to be decided.*
- *If some value v has been already locked-in, then the proposer* ***must*** *propose that value.*

- This can be achieved by having the proposer read accepted values from a majority of acceptors.

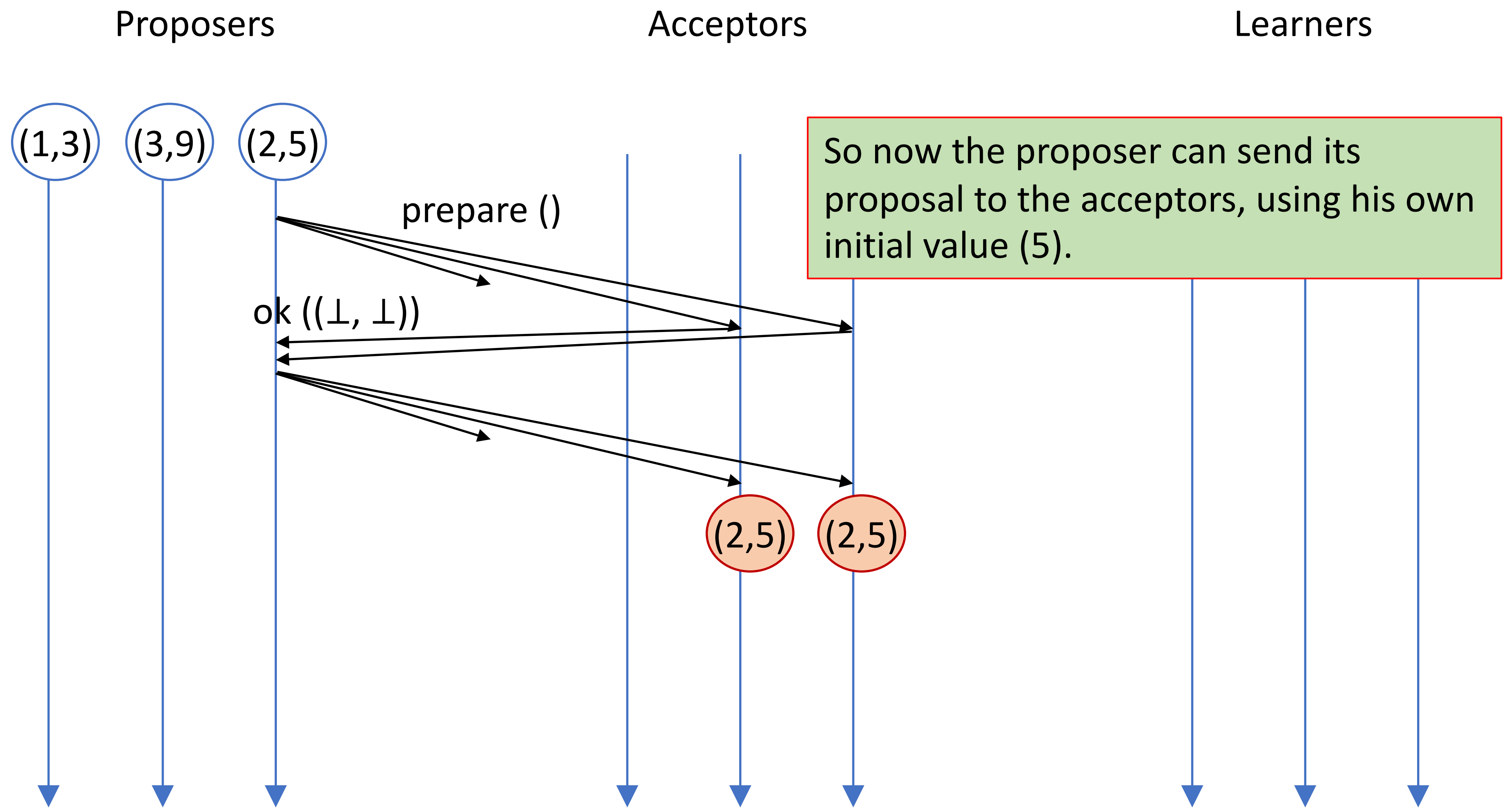
How to determine if there is a previous proposal that is locked-in?



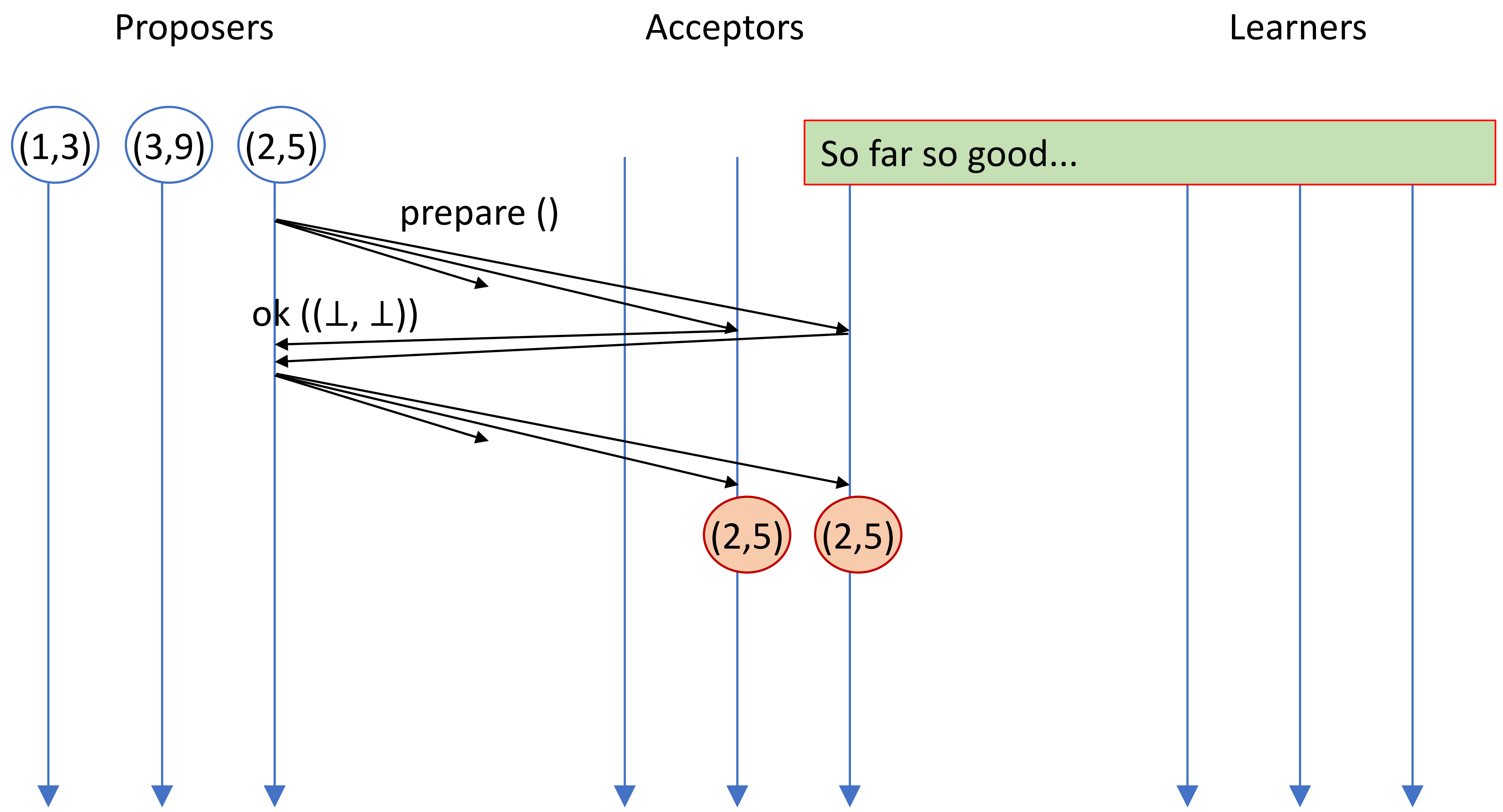
How to determine if there is a previous proposal that is locked-in?



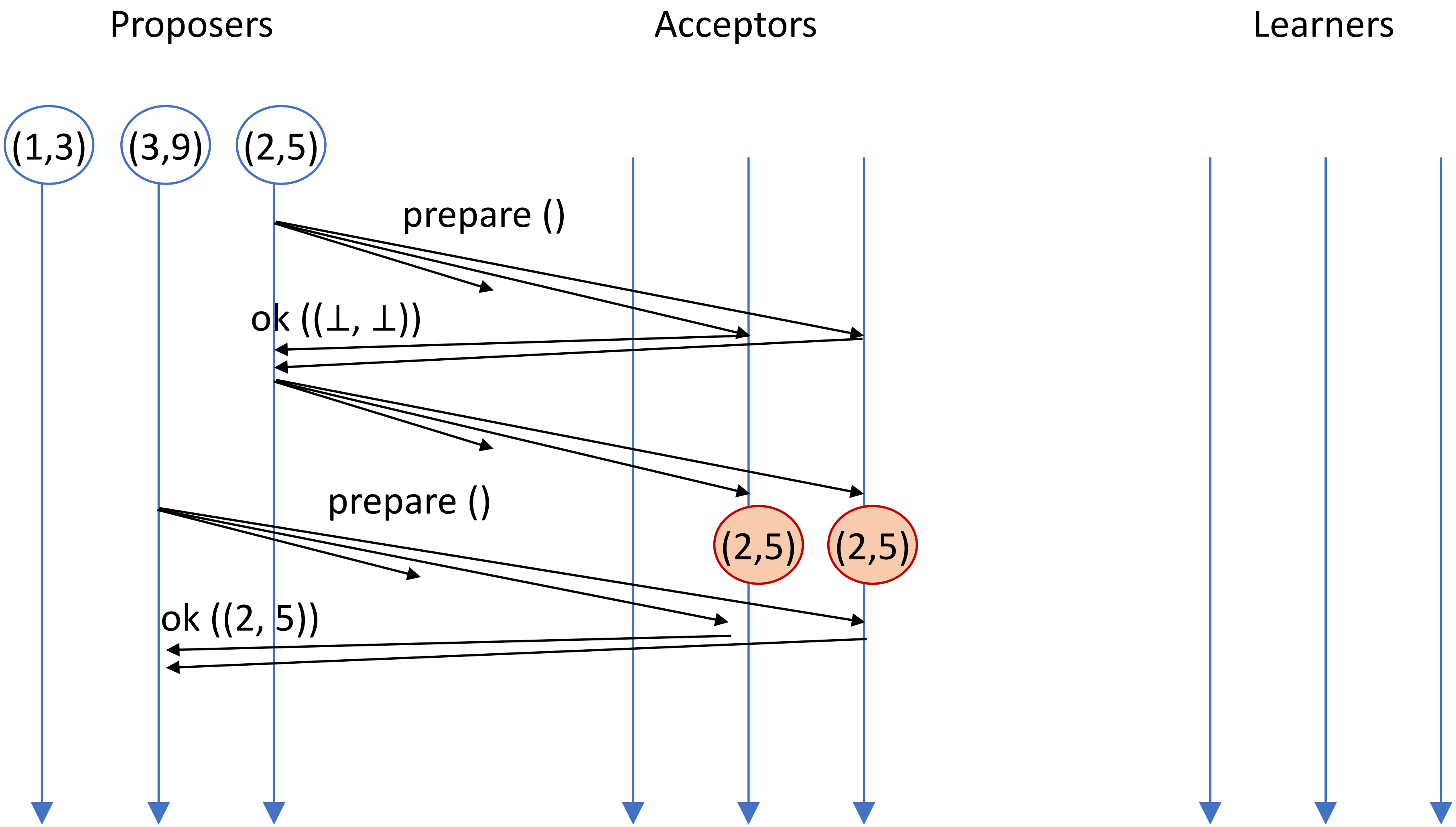
How to determine if there is a previous proposal that is locked-in?



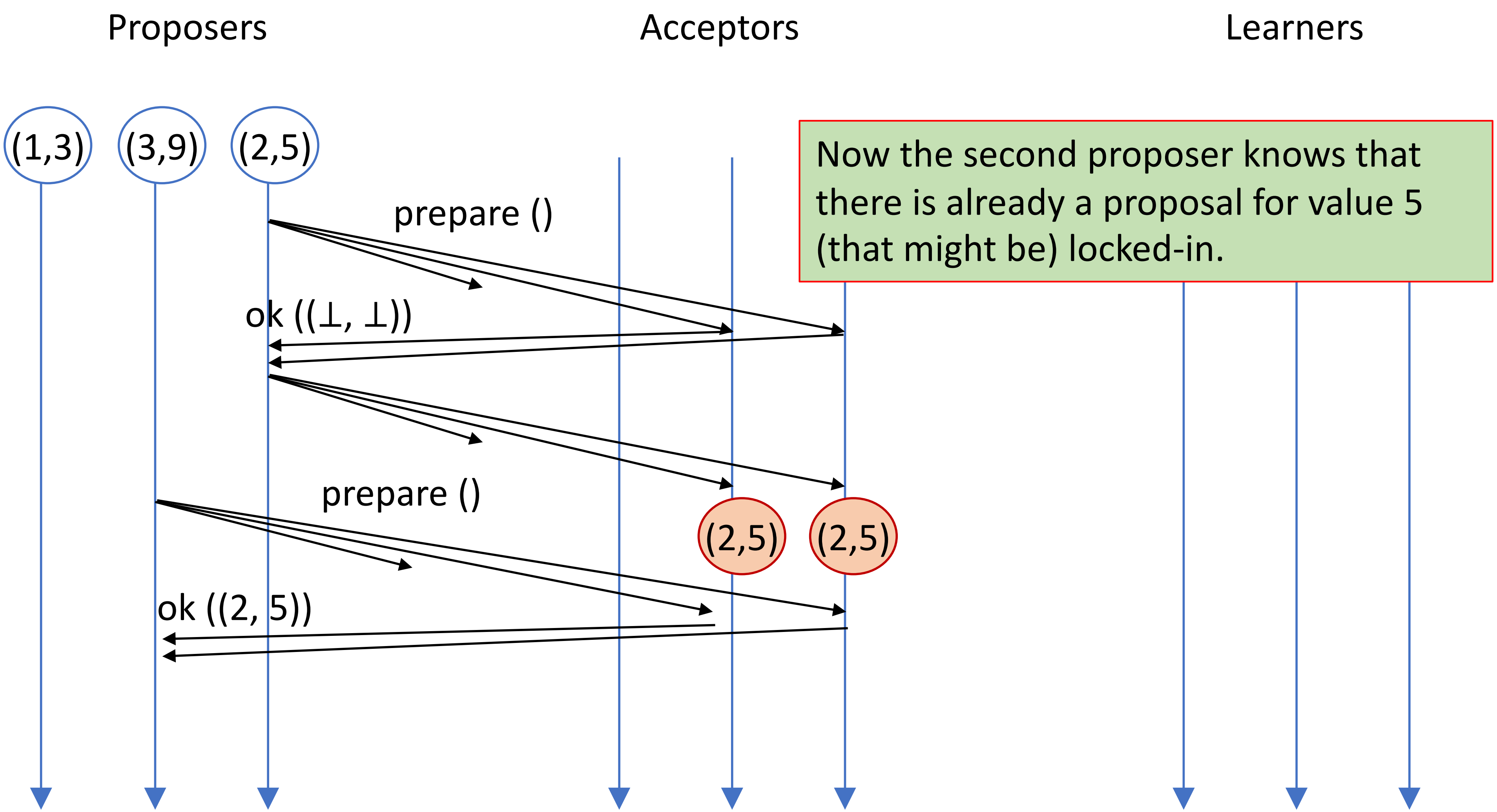
How to determine if there is a previous proposal that is locked-in?



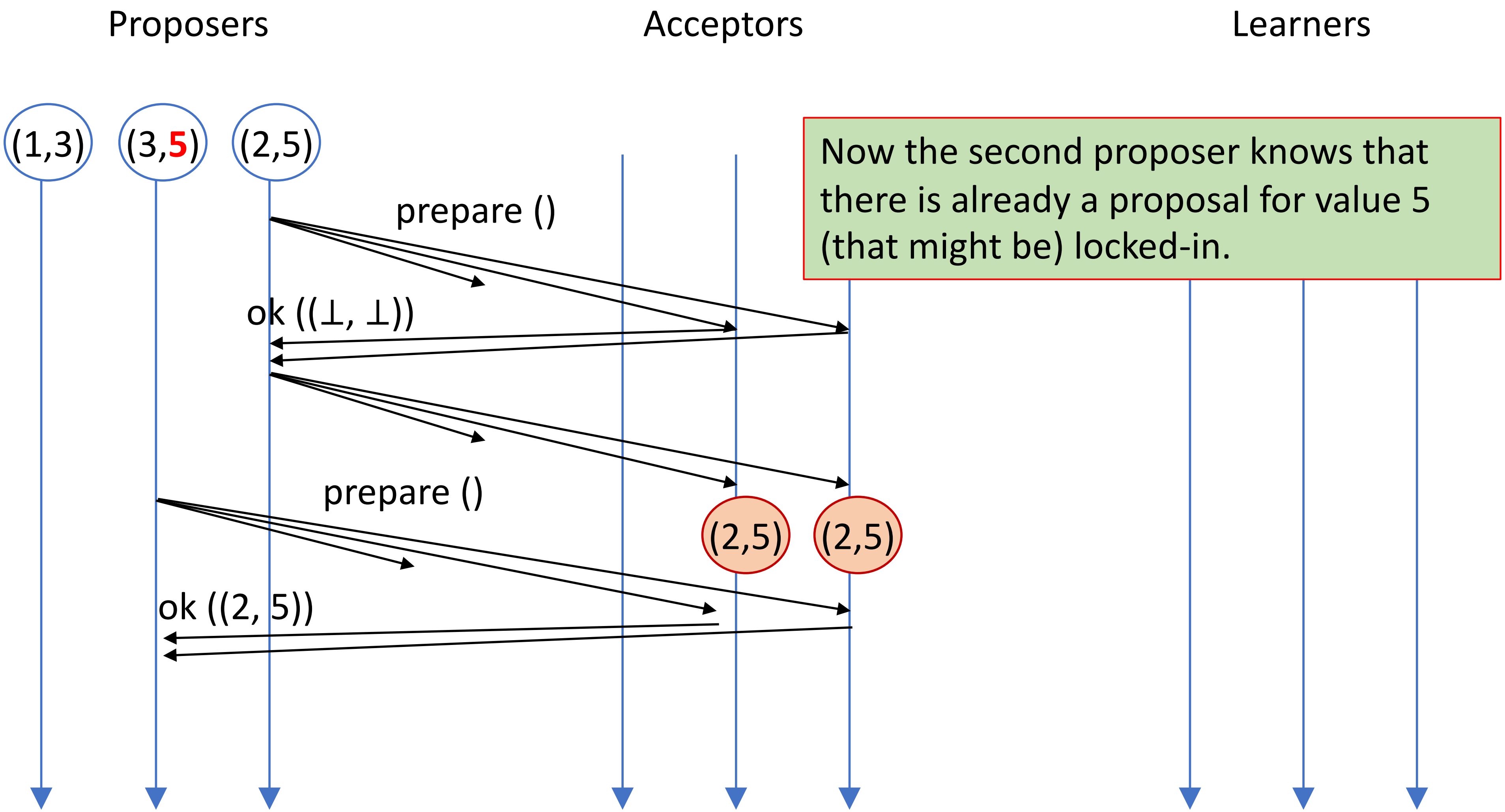
How to determine if there is a previous proposal that is locked-in?



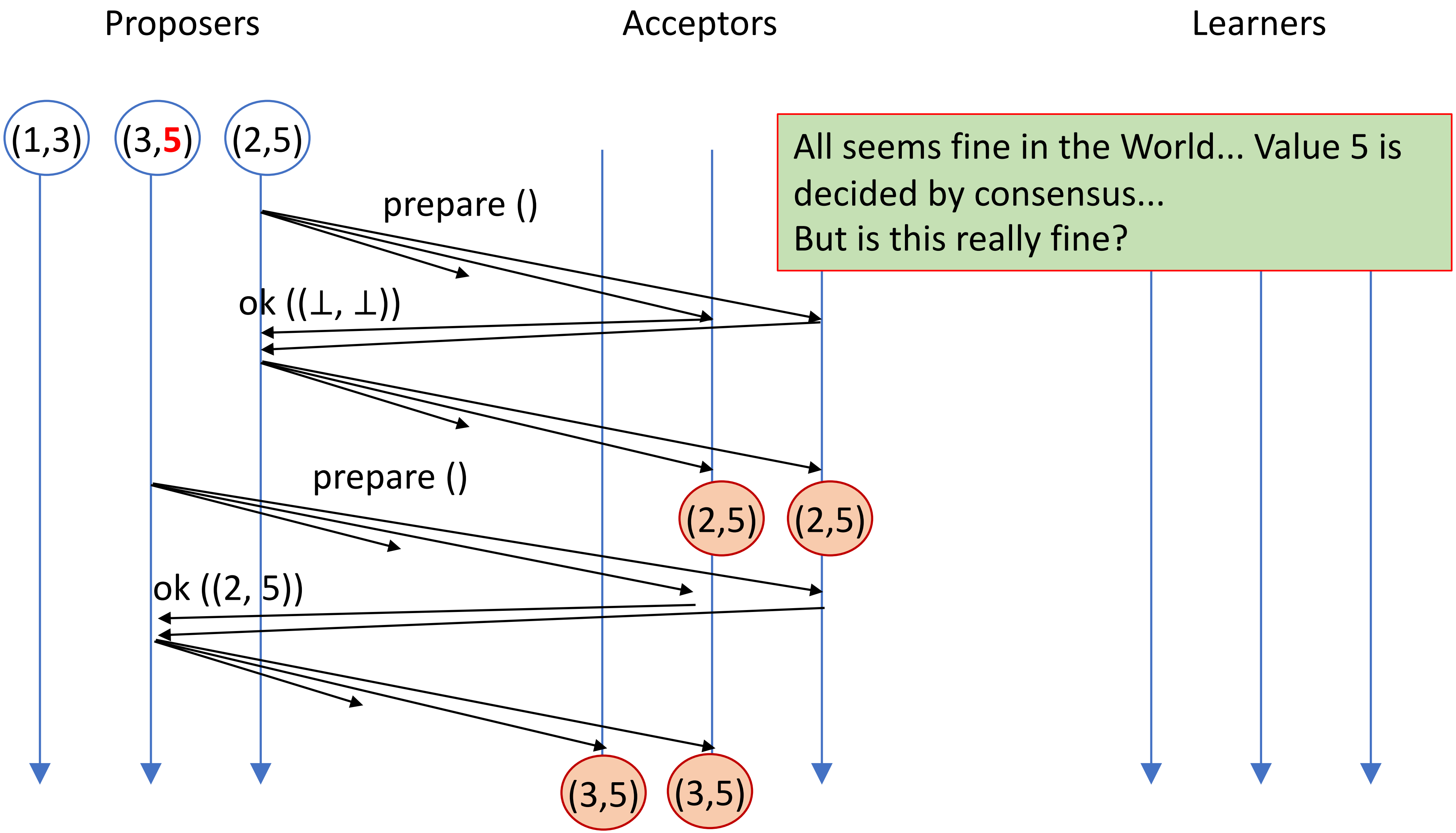
How to determine if there is a previous proposal that is locked-in?



How to determine if there is a previous proposal that is locked-in?



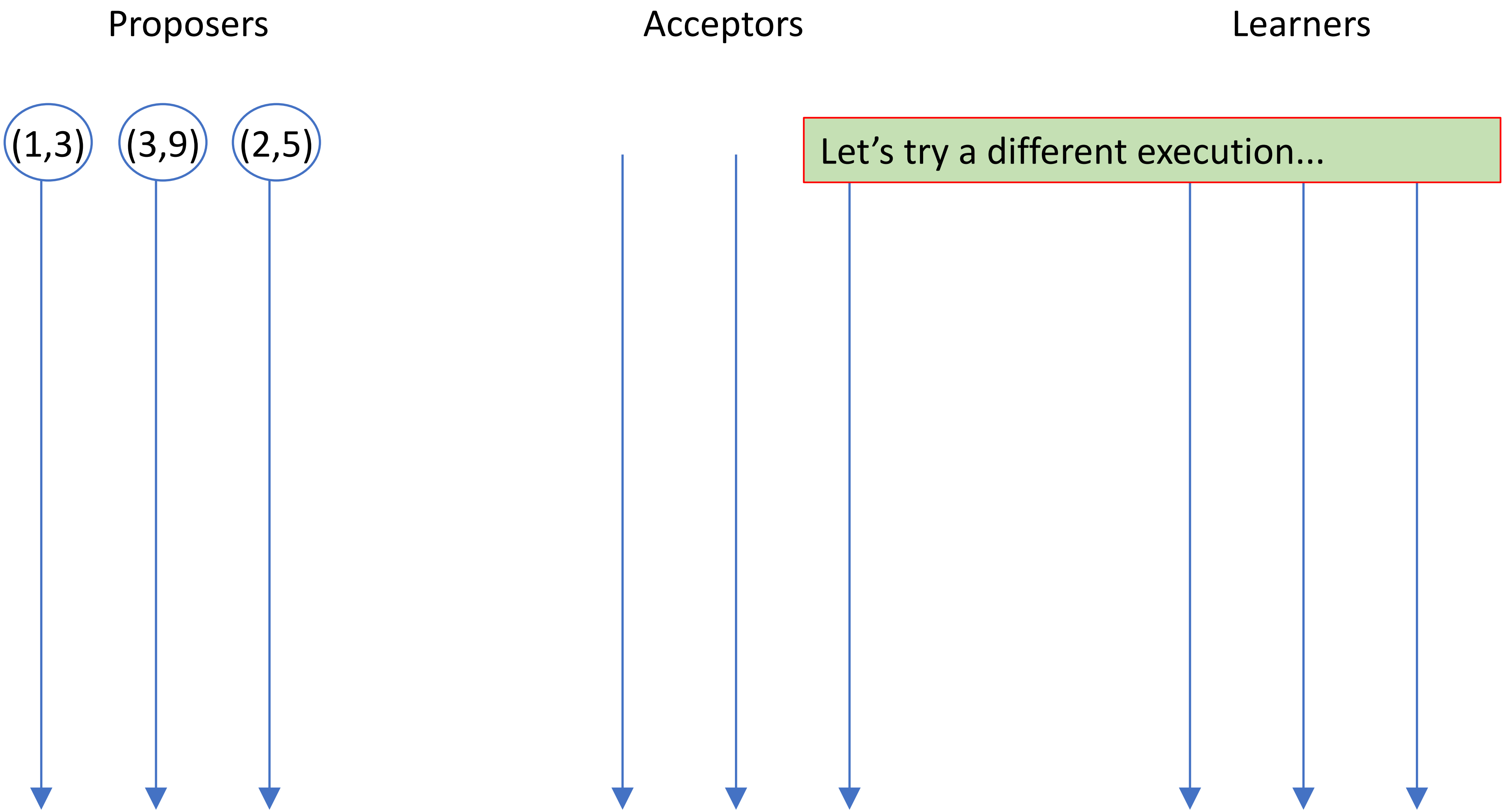
How to determine if there is a previous proposal that is locked-in?



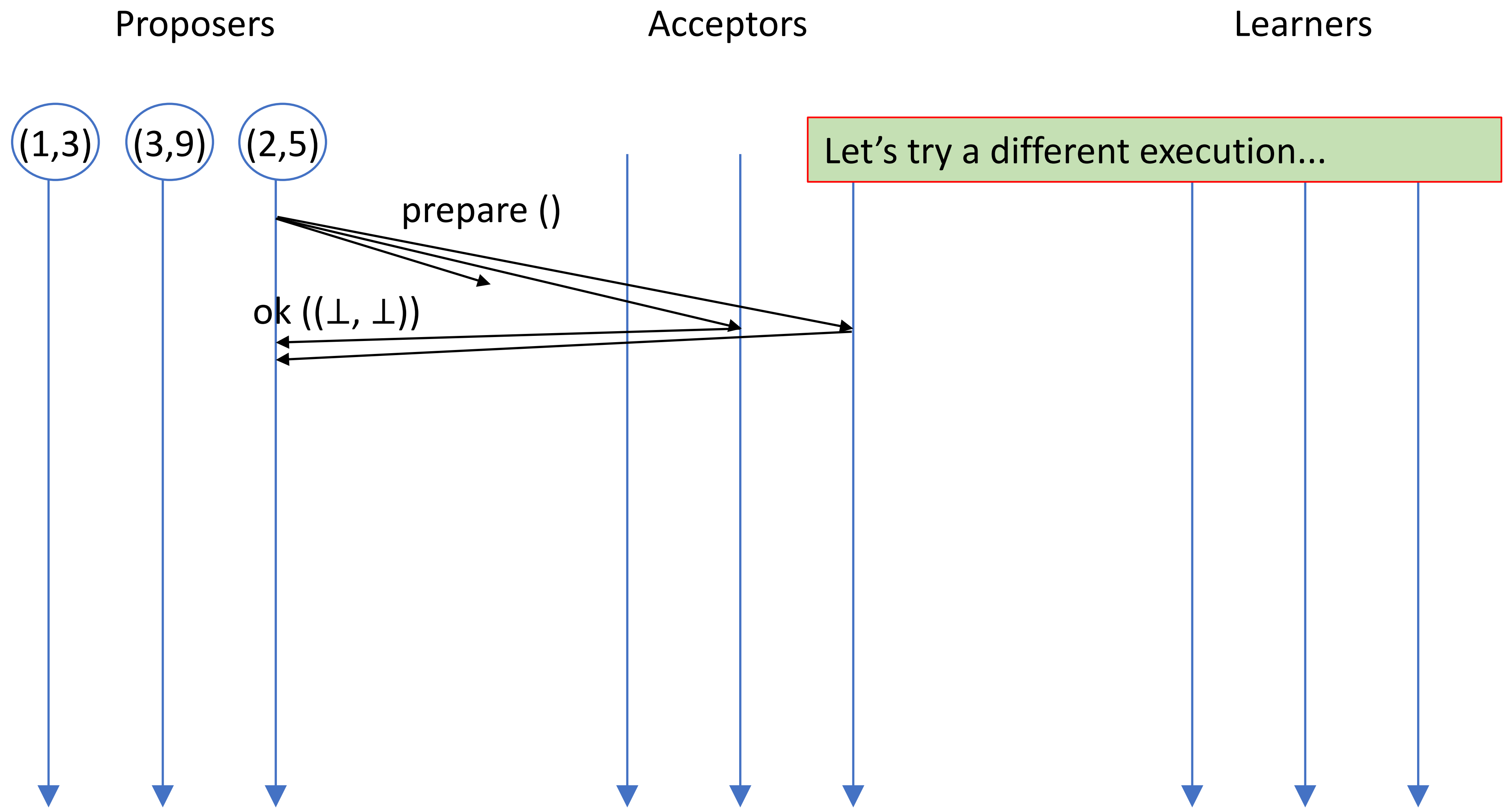
What did we do there?

- Assume that proposer p wants to emit a proposal with psn_i.
- If p can be sure that there is no other proposal with psn_j, such that $psn_j < psn_i$, and value v' that was not decided then he can propose his initial value v.
- Otherwise, p must propose value v' that was already locked in.
- To do so, the proposer checks selected proposals from a majority of acceptors, and if there is a proposal already there, he changes his proposed value to the value in the proposal with largest psn_j (might not be locked-in yet, but is safe).

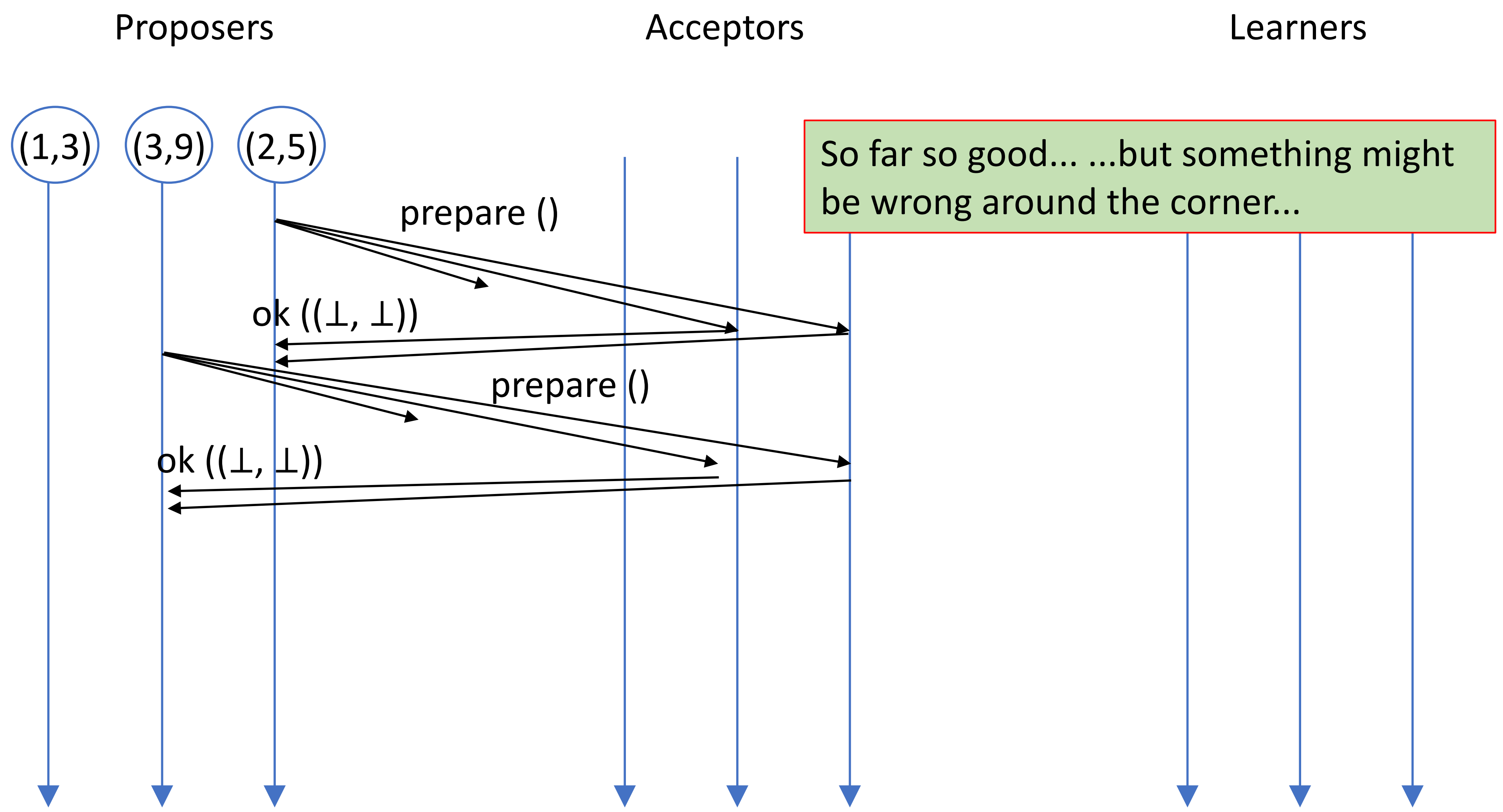
Is this enough to ensure that we don't decide two different values?



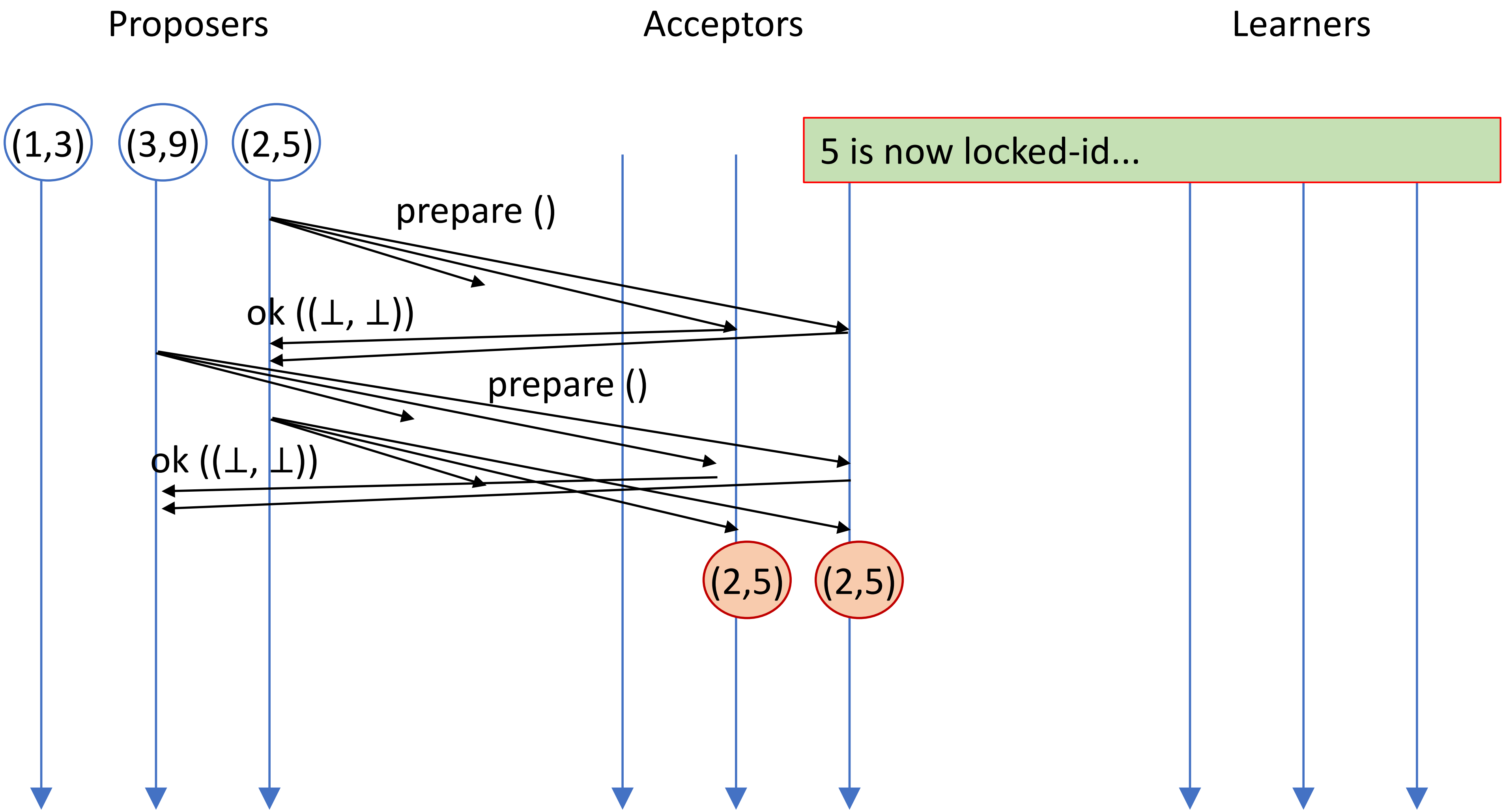
Is this enough to ensure that we don't decide two different values?



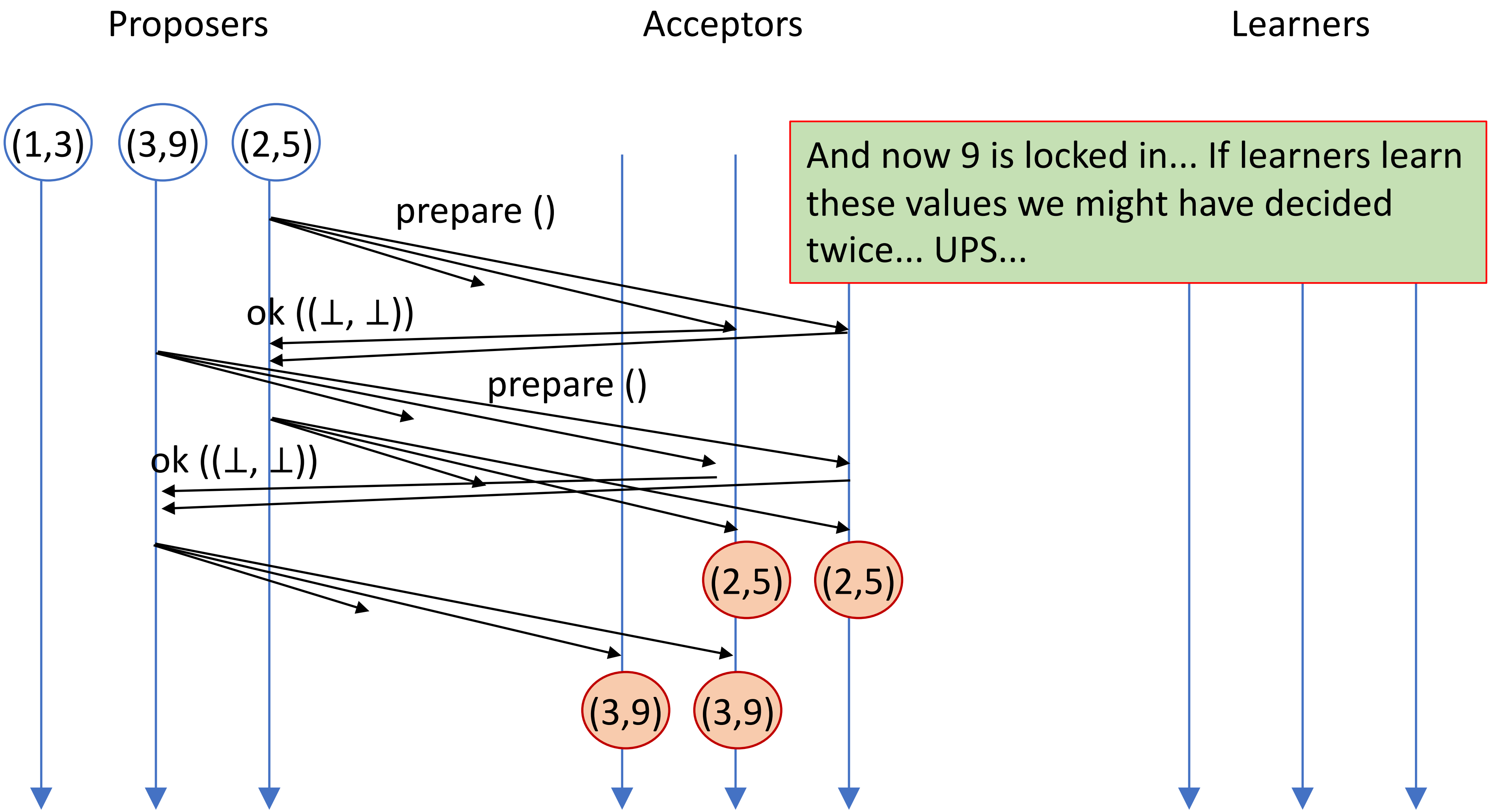
Is this enough to ensure that we don't decide two different values?



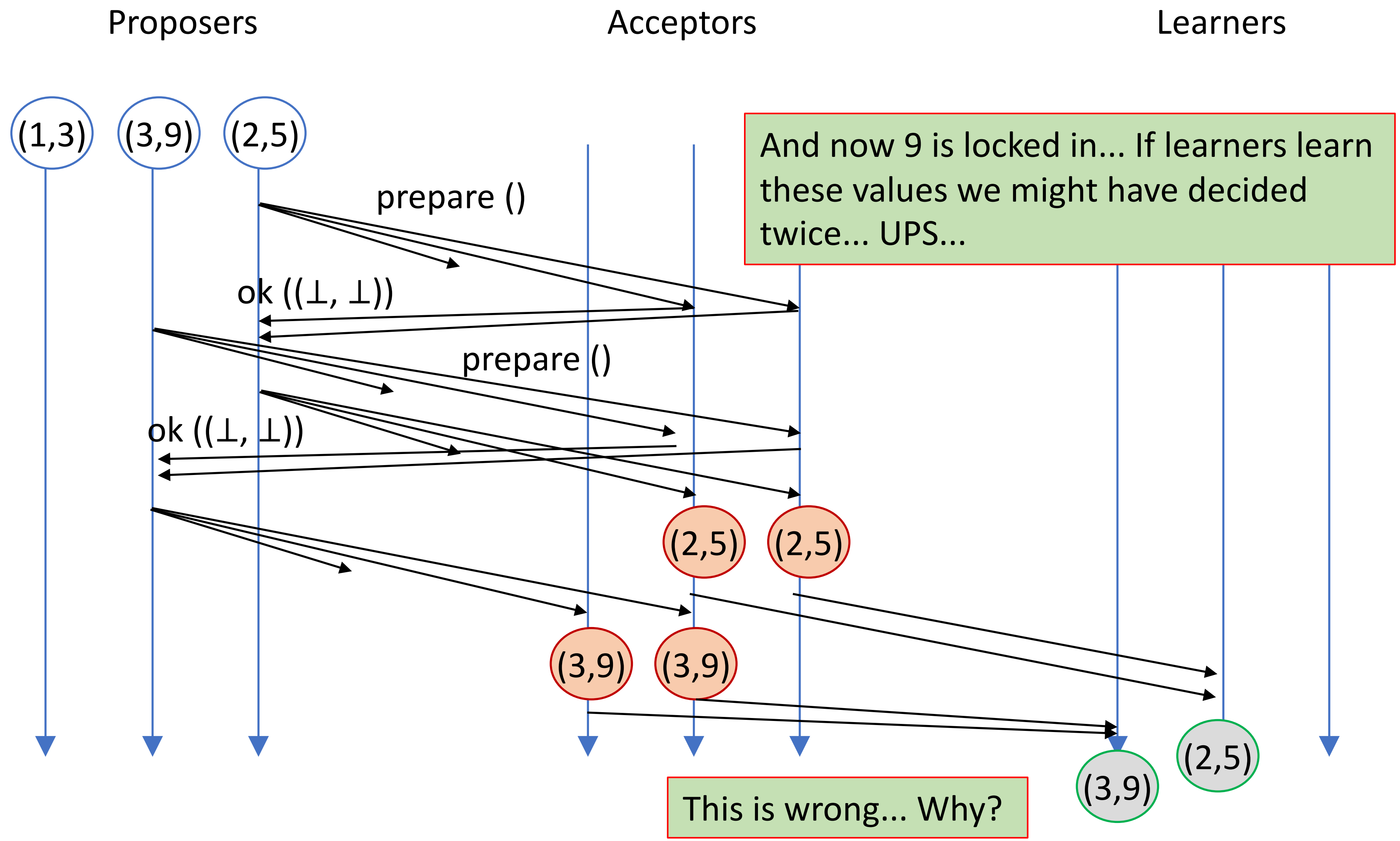
Is this enough to ensure that we don’t decide two different values?



Is this enough to ensure that we don't decide two different values?



Is this enough to ensure that we don't decide two different values?



And now 9 is locked in... If learners learn these values we might have decided twice... UPS...

This is wrong... Why?

What went wrong there?

- By checking if some proposal was already selected in a majority of acceptors, the proposer can be sure that things in his past (i.e., proposals with psn below his own) have not yet locked-in a value…
- However, this does not provides a guaranteee to the proposer that in the future there will be no proposal with a psn below his own that will lock-in a value.

What went wrong there?

- By checking if some proposal was already selected in a majority of acceptors, the proposer can be sure that things in his past (i.e., proposals with psn below his own) have not yet locked-in a value…
- However, this does not provides a guaranteee to the proposer that in the future there will bo no proposal with a psn below his own that will lock-in a value.

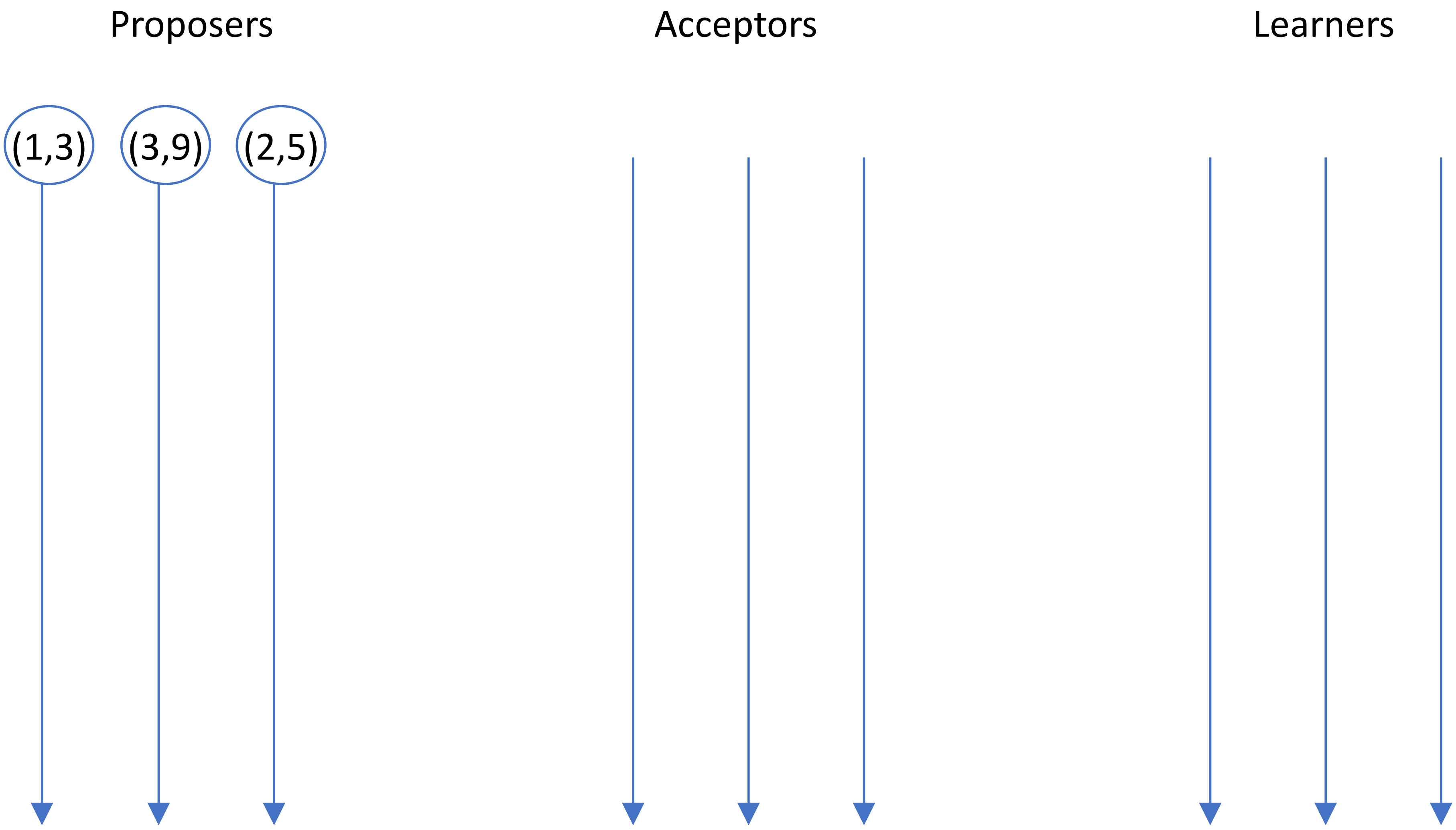
- In some sense we covered the past, but we also need to make sure that proposals with lower psn cannot affect the future. How to do this?

Ensuring that the past no longer affects the future…

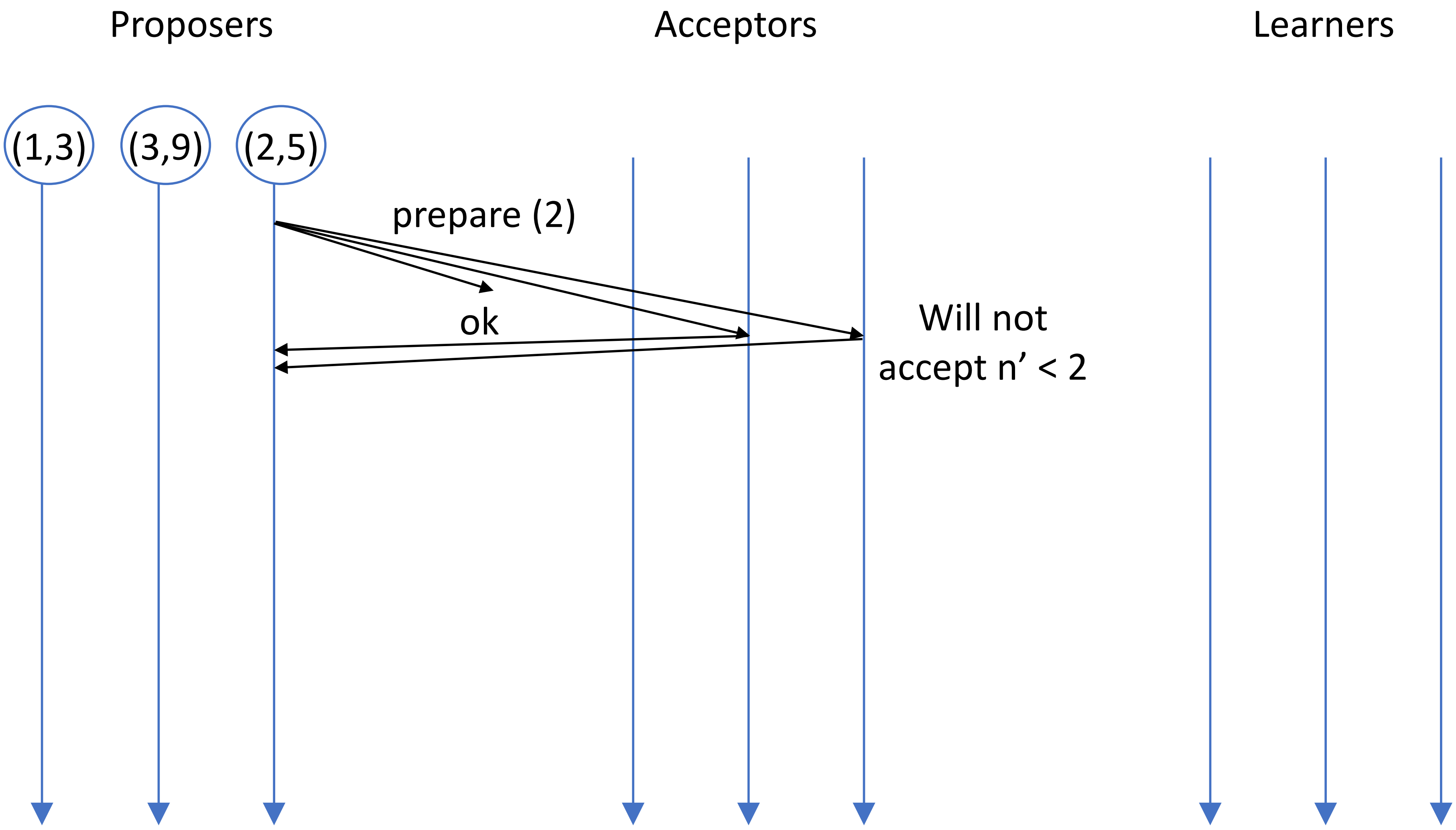
- Future manipulation is tricky, and the proposer on its own will not be able to control it.
- We need to get assistance from the acceptors.

- In particular, when the proposer checks if there is already a locked-in value, he can inform the acceptors (a majority in this case) of the sequence number that he is planning on using. Acceptor that reply to him **also promise to not accept any proposal with a psn below that one** (hence *the past cannot affect the future*)

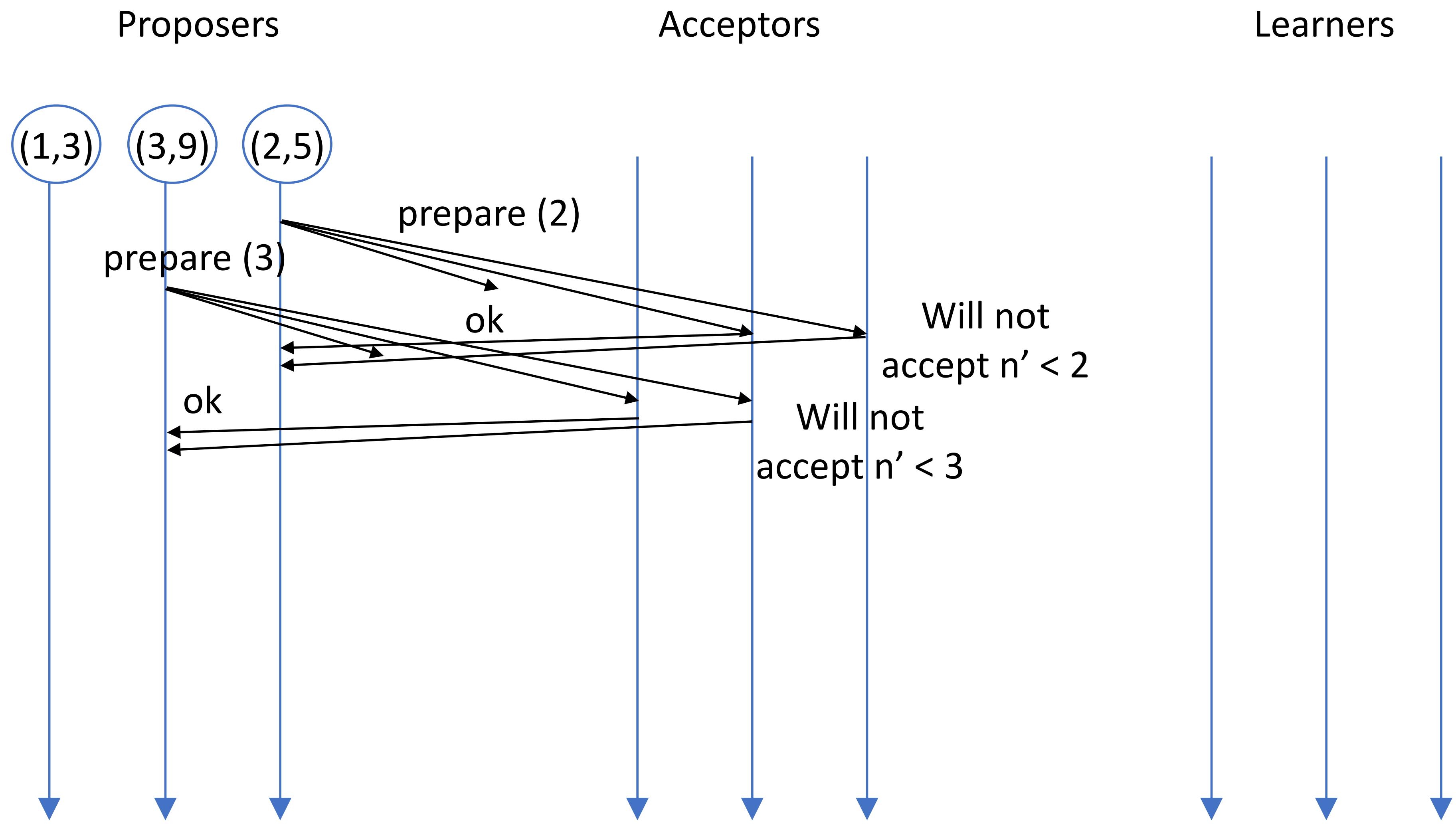
How to determine if there is a previous proposal that is either locked-in or can become locked-in?



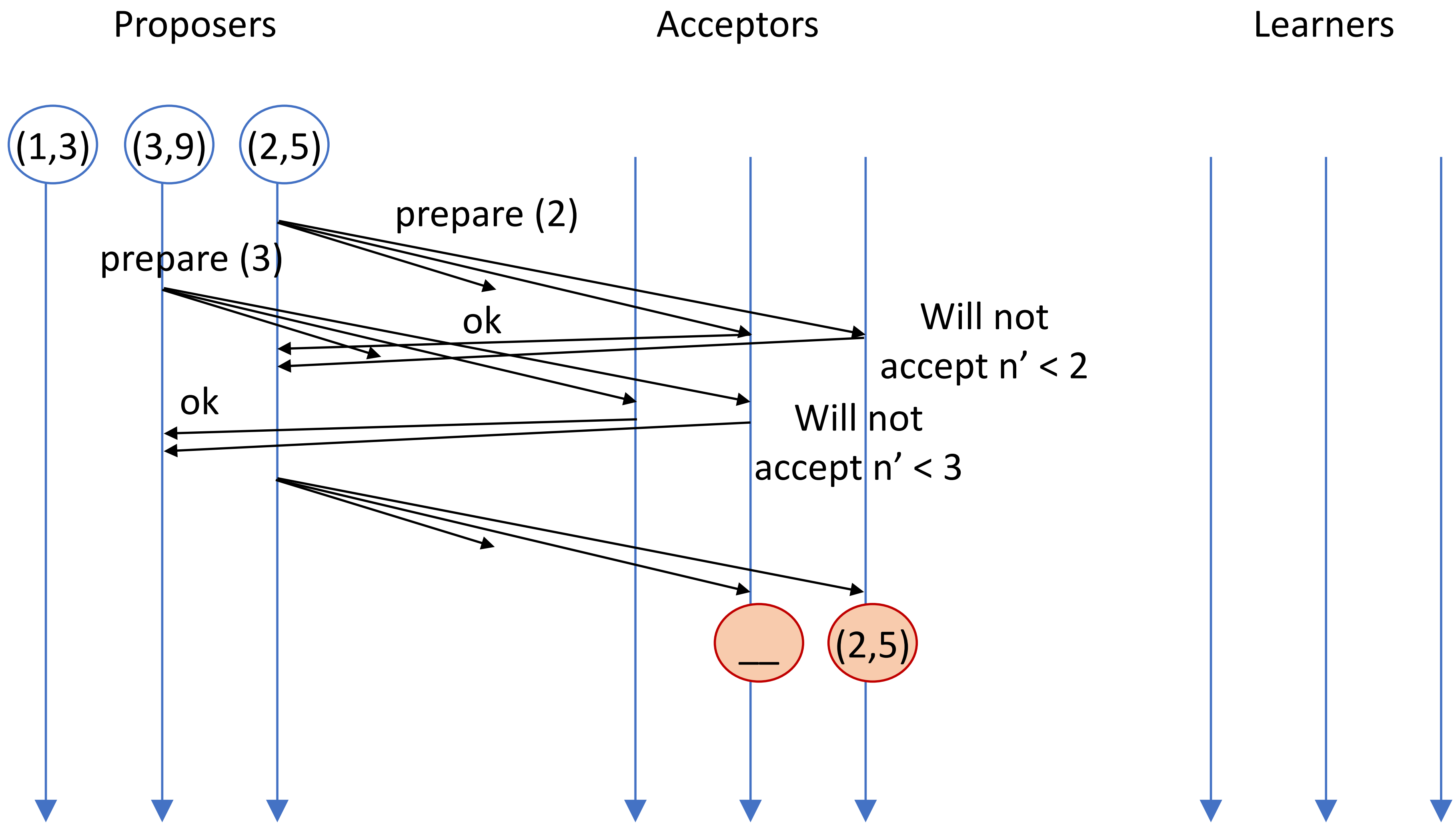
How to determine if there is a previous proposal that is either locked-in or can become locked-in?



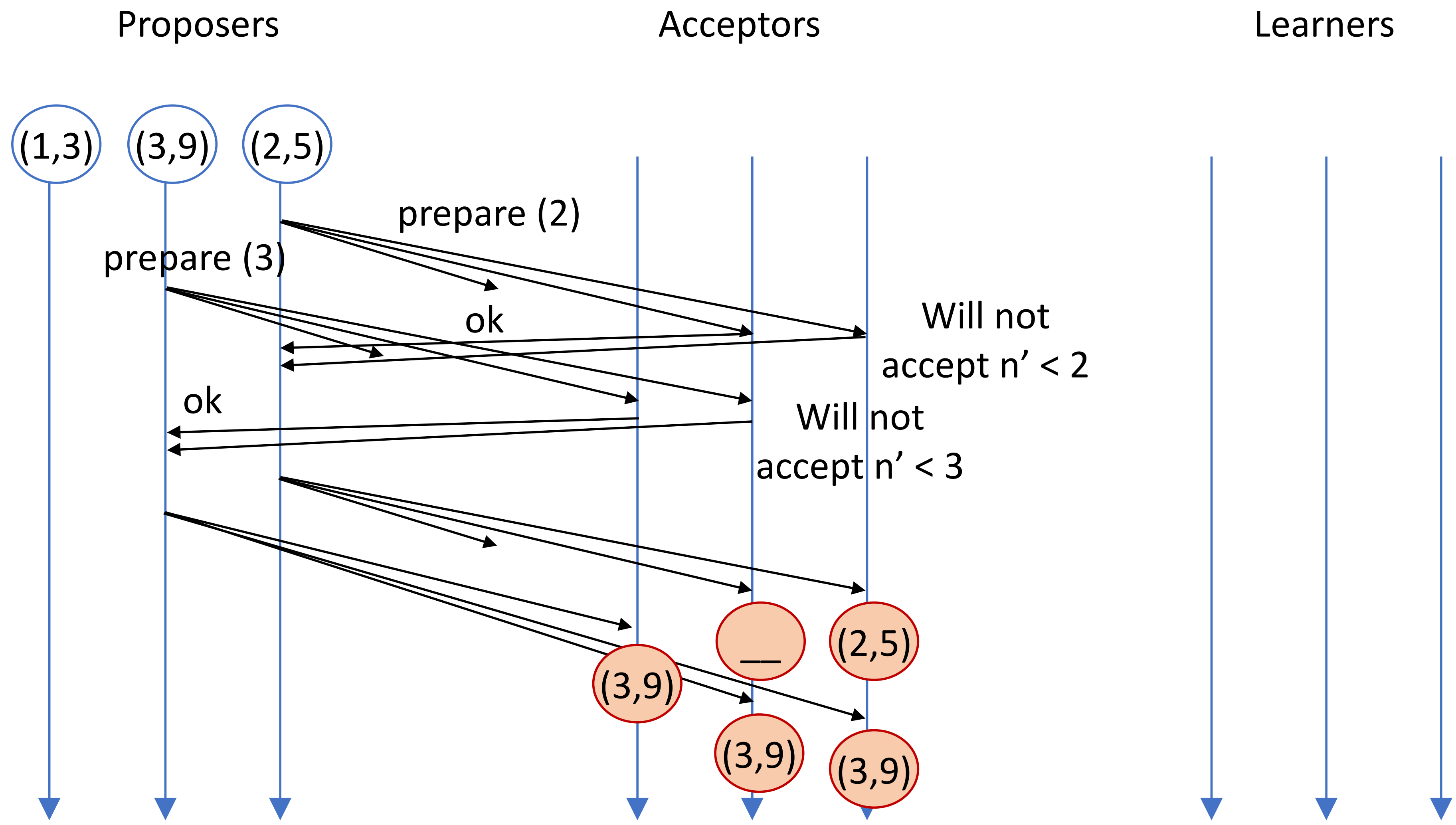
How to determine if there is a previous proposal that is either locked-in or can become locked-in?



How to determine if there is a previous proposal that is either locked-in or can become locked-in?



How to determine if there is a previous proposal that is either locked-in or can become locked-in?



Why does this work?

- When a proposer gathers a majority quorum from acceptors that promise that they will not accept a proposal with a lower psn than his own, it makes it impossible for such a proposal value to be decided (since a majority that will accept a proposal with a lower sequence number becomes impossible to obtain).

Why does this work?

- When a proposer gathers a majority quorum from acceptor that promise that they will not accept a proposal with a psn lower than his own, it makes it impossible for such a proposal value to be decided (since a majority that will accept a proposal with a lower sequence number becomes impossible to obtain).
- From a practical standpoint, this ensures that if a proposer effectively proposes his initial value, then no proposal with a lower sequence number exists that ***was already*** accepted by a majority of acceptor ***or will ever be*** accepted by such a majority.

Proposer Algorithm

```
PROPOSE(v)
  choose unique n, higher than any n seen so far
  send PREPARE(n) to all nodes
  if PREPARE_OK(na, va) from majority then
     va = va with highest na (or choose v
 otherwise)
  send ACCEPT (n, va) to all
  if ACCEPT_OK(n) from majority then
     send DECIDED(va) to all
```

Proposer Algorithm

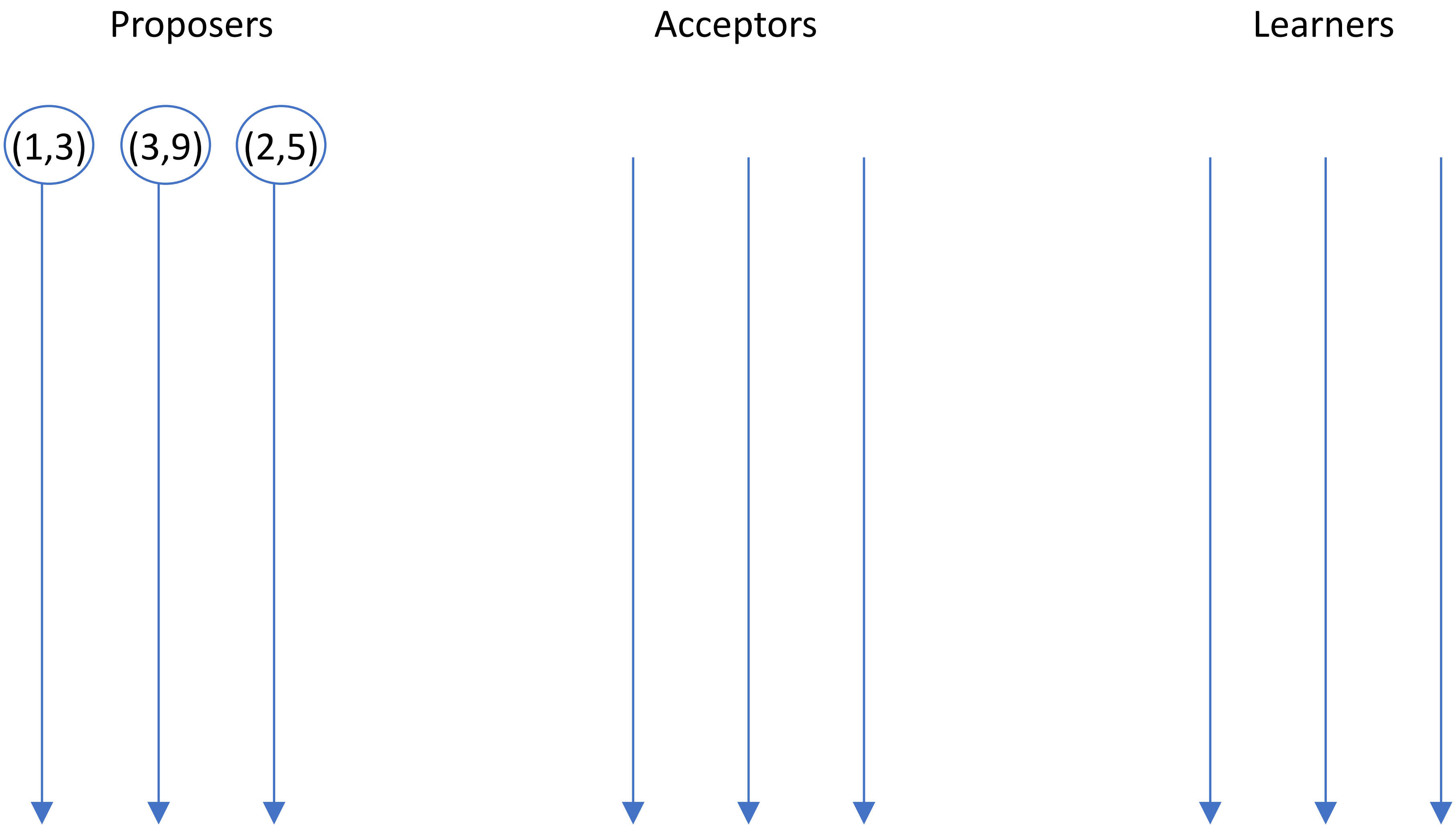
```
PROPOSE(v)
  choose unique n, higher than any n seen so far
  send PREPARE(n) to all nodes
  if PREPARE_OK(na, va) from majority then
     va = va with highest na (or choose v
 otherwise)
  send ACCEPT (n, va) to all
  if ACCEPT_OK(n) from majority then
     send DECIDED(va) to all
```

Evidently, if no valid quorum is gathered of either PREPARE_OK or ACCEPT_OK messages, the proposer should timeout and reset this algorithm using a larger sequence number (n).

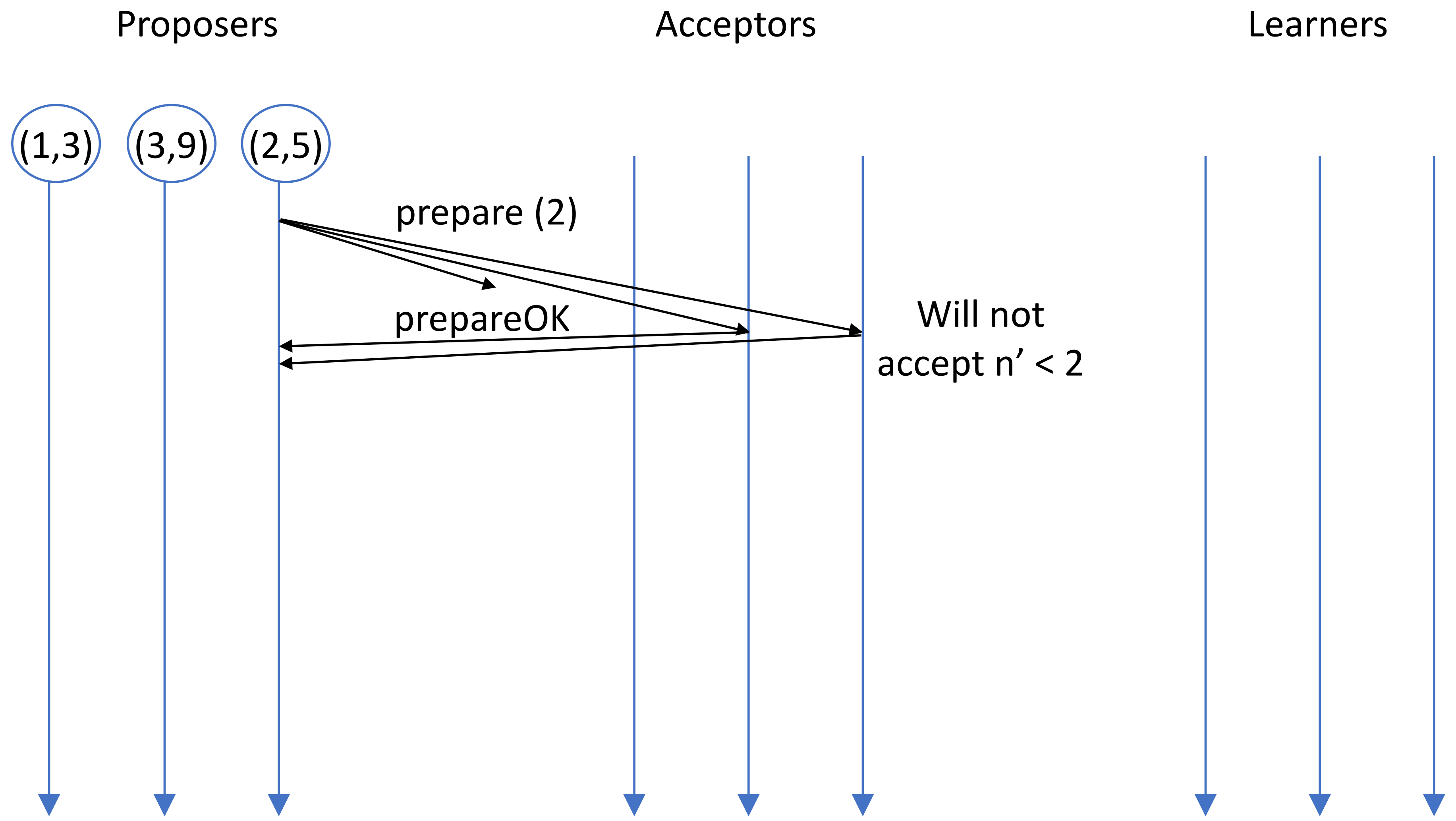
Acceptor Algorithm

```
State: np (highest prepare), na, va (highest accept)
/* This state is maintained in stable storage */

PREPARE(n)
  if n > np then
     np = n   // will not accept with seq. nub. < n
     reply <PREPARE_OK,na,va>

ACCEPT(n, v)
  if n >= np then
    na = n
    va = v
    reply with <ACCEPT_OK,n>
```

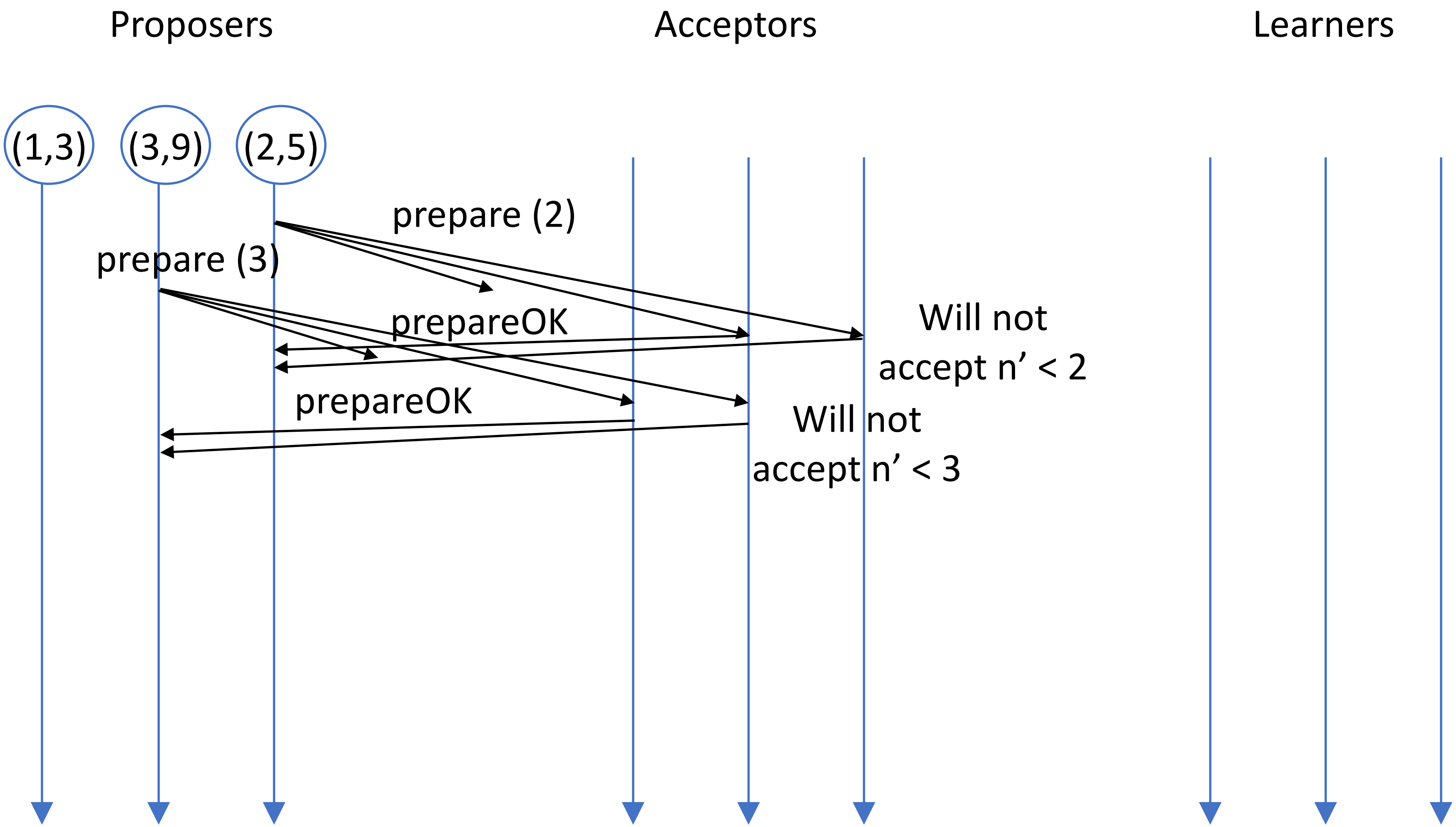
Paxos running



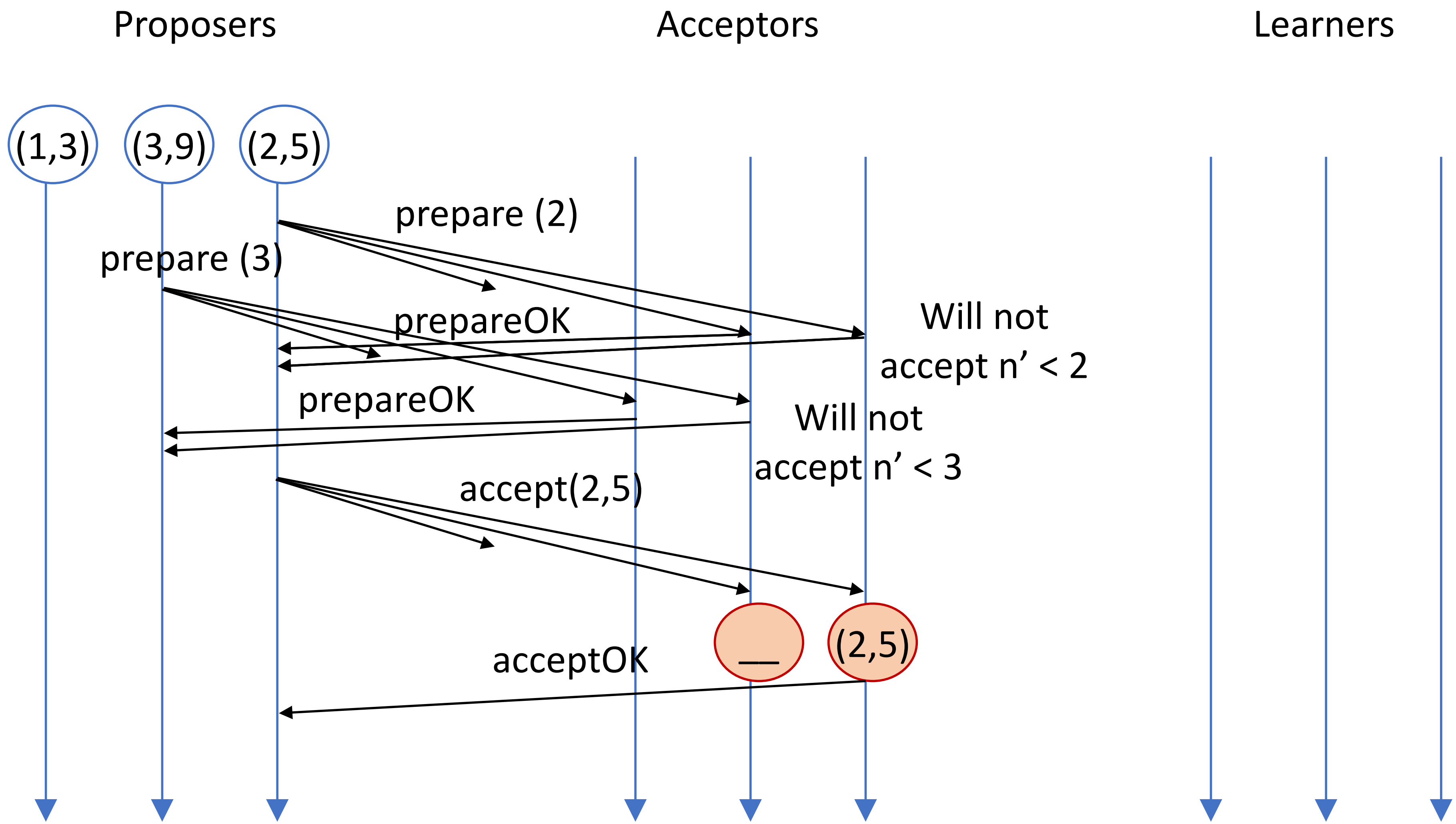
Paxos running



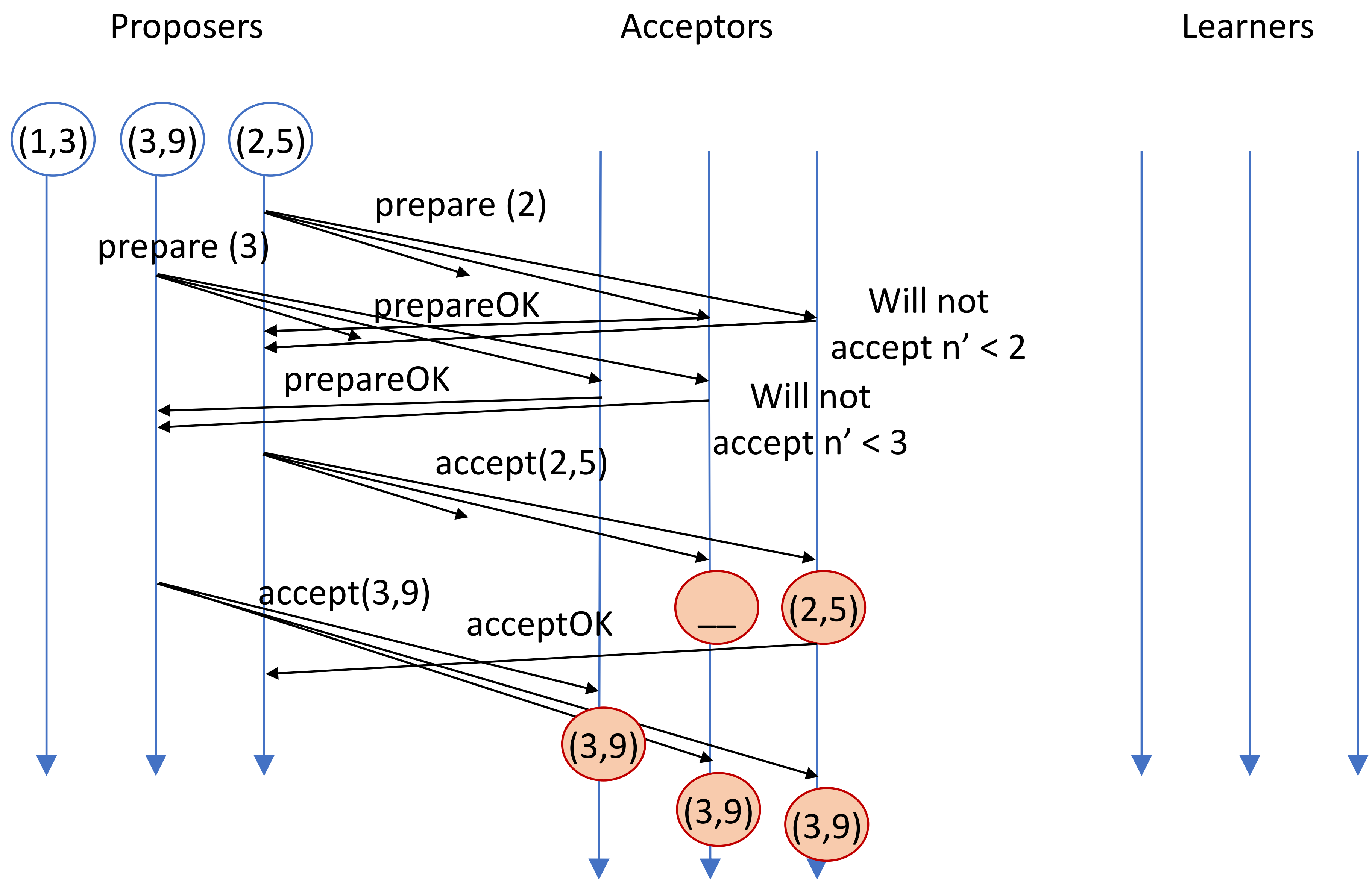
Paxos running



Paxos running



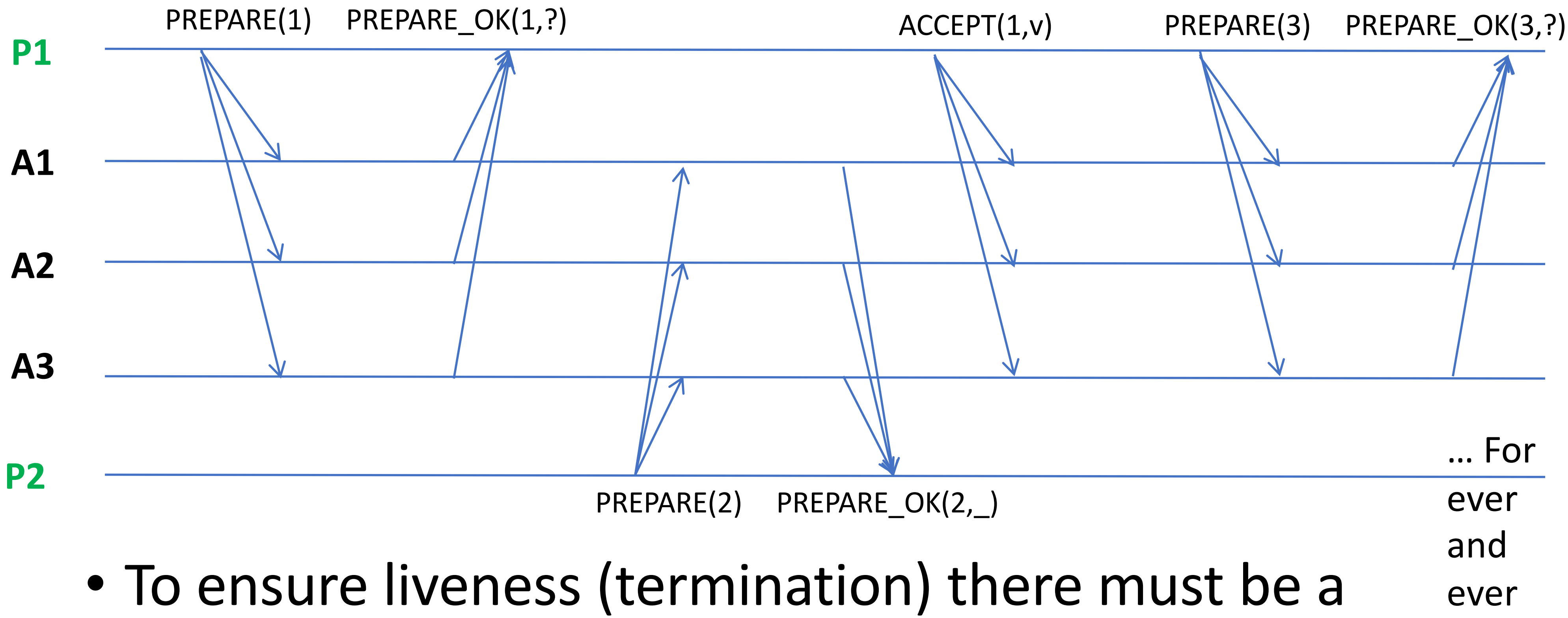
Paxos running



Learners

- Learns can either contact acceptors or be contacted by acceptors to know the value that they have selected.
- When a majority of acceptors select the same value then a decision can be made.

Liveness is not guaranteed (Termination Property)



- To ensure liveness (termination) there must be a single proposer (leader). This is only possible if a long enough period of synchrony happens in the system.

Homework 3:

- Use paxos to build a total order broadcast protocol that operates in an asynchronous system model under the crash fault model:
  - TO (Total Order): Let m_1 and m_2 be any two messages. Let p_i and p_j be any two correct processes that deliver m_1 and m_2. If p_i delivers m_1 before m_2, then p_j delivers m_1 before m_2.
  - RB1 (Validity): If a correct process i broadcasts message m, then i eventually delivers the message.
  - RB2 (No Duplications): No message is delivered more than once.
  - RB3 (No Creation): If a correct process j delivers a message m, then m was broadcast to j by some process i.
  - RB4 (Aggrement): If a message m is delivered by some correct process i, them m is eventually delivered by every correct process j.
- You can use up to two primitives (paxos is mandatory):
  - Paxos
    - - Request: pprepare(v)
    - - Indication: pdecided(v)
  - Reliable Broadcast
    - - Request: broadcast(m)
    - - Indication: deliver(m)

Interface of your protocol:

Request: - tobcast(m)

Indication: - todeliver (m)