Algorithms and Distributed Systems 2019/2020 (Lecture Seven)

MIEI - Integrated Master in Computer Science and Informatics

Specialization block

João Leitão (jc.leitao@fct.unl.pt)

Lecture structure:

- Solution(s) for Homework 3
- Revise Paxos
- Paxos and State Machine Replication
- Multi Paxos

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)

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)~~ ppropose(v)
    - - Indication: pdecided(v)
  - Reliable Broadcast
    - - Request: broadcast(m)
    - - Indication: deliver(m)

Interface of your protocol:

Request: - tobcast(m)

Indication: - todeliver (m)

Total Order Broadcast (V1)

Algorithm 1: Total Order Broadcast (Using Paxos and Reliable Broadcast)

Interface:
Requests:
toBCast (m **)**
Indications:
toDeliver (m **)**

State:
delivered `//set of Ids of messages and respective payload already delivered`
pending `//Messages to be ordered`
paxos `//Ordered Instances of Paxos`
currentInstance `//Current instance of Paxos executing`
waiting `//Boolean indicating if something is being ordered`

Upon Init () do:
delivered $\longleftarrow \{\}$
pending $\longleftarrow \{\}$
paxos $\longleftarrow \{\}$
currentInstance $\longleftarrow 0$
waiting $\longleftarrow false$

Upon toBCast(m **) do:**
mid $\longleftarrow$ generateUniqueID(m)
Trigger broadcast($\{mid, m\}$) `//trigger local reliable bcast`

Upon deliver($\{mid, m\}$ **) do:**
If mid $\notin$ delivered $\wedge \{mid, m\} \notin$ pending **do:**
pending $\longleftarrow$ pending $\cup \{mid, m\}$
Call orderForDelivery()

Upon pdecided($\{mid, m\}$ **) do:**
pending $\longleftarrow$ pending $\setminus \{mid, m\}$
delivered $\longleftarrow$ delivered $\cup \{mid\}$
Trigger toDeliver (m)
waiting $\longleftarrow false$
Call orderForDelivery()

Procedure orderForDelivery():
If $\neg$ waiting $\wedge \exists e \in$ pending **do:**
currentInstance $\longleftarrow$ currentInstance $+1$
paxos[currentInstance] $\longleftarrow$ initPaxosInstance()
$\{mid, m\} \longleftarrow$ pickAtRandom(pending)
Trigger paxos[currentInstance].ppropose($\{mid, m\}$)
waiting $\longleftarrow true$

Total Order Broadcast (V2)

Algorithm 2: Total Order Broadcast (Using Paxos)

Interface:

Requests:

toBCast (m **)**

Indications:

toDeliver (m **)**

State:

pending `//Messages to be ordered`
paxos `//Ordered Instances of Paxos`
currentInstance `//Current instance of Paxos executing`
waiting `//Boolean indicating if something is being ordered`

Upon Init () do:

pending $\longleftarrow \{\}$
paxos $\longleftarrow \{\}$
currentInstance $\longleftarrow 0$
waiting $\longleftarrow false$

Upon toBCast(m **) do:**

mid $\longleftarrow$ generateUniqueID(m)
pending $\longleftarrow$ pending $\cup \{mid, m\}$
Call orderForDelivery()

Upon pdecided($\{mid, m\}$ **) do:**

pending $\longleftarrow$ pending $\setminus \{mid, m\}$
Trigger toDeliver (m)
waiting $\longleftarrow false$
Call orderForDelivery()

Procedure orderForDelivery():

If $\neg$ waiting $\wedge \exists e \in$ pending **do:**

currentInstance $\longleftarrow$ currentInstance $+1$
paxos[currentInstance] $\longleftarrow$ initPaxosInstance()
$\{mid, m\} \longleftarrow$ pickAtRandom(pending)
Trigger paxos[currentInstance].ppropose($\{mid, m\}$)
waiting $\longleftarrow true$

Total Order Broadcast (V3)

Algorithm 3: Total Order Broadcast (Using Paxos – Somewhat Simplified Interface)

Interface:

Requests:

toBCast (m **)**

Indications:

toDeliver (m **)**

State:

pending `//Messages to be ordered`

waiting `//Boolean indicating if something is being ordered`

currentInstance `//Current instance of Paxos executing`

Upon Init () do:

pending $\longleftarrow \{\}$

waiting $\longleftarrow false$

currentInstance $\longleftarrow 0$

Upon toBCast(m **) do:**

mid $\longleftarrow$ generateUniqueID(m)

pending $\longleftarrow$ pending $\cup \{mid, m\}$

Call orderForDelivery()

Upon pdecided($\{mid, m\}$ **) do:**

pending $\longleftarrow$ pending $\setminus \{mid, m\}$

Trigger toDeliver (m)

waiting $\longleftarrow false$

Call orderForDelivery()

Procedure orderForDelivery():

If $\neg$ waiting $\wedge \exists e \in$ pending **do:**

$\{mid, m\} \longleftarrow$ pickAtRandom(pending)

currentInstance $\longleftarrow$ currentInstance $+1$

Trigger ppropose(currentInstance, $\{mid, m\}$)

waiting $\longleftarrow true$

Algorithm for proposer

```
PROPOSE(v)
  while(true) do
    choose unique sn, higher than any n seen so far
    send PREPARE(sn) to all acceptors
    if PREPARE_OK(sna, va) from majority then
       va = va with highest sna (or choose v otherwise)
       send ACCEPT (sn, va) to all acceptors
       if ACCEPT_OK(n) from majority then
          send DECIDED(va) to client
          break
       else //timeout on waiting ACCEPT_OK
          continue
    else //timeout on waiting PREPARE_OK
       continue
```

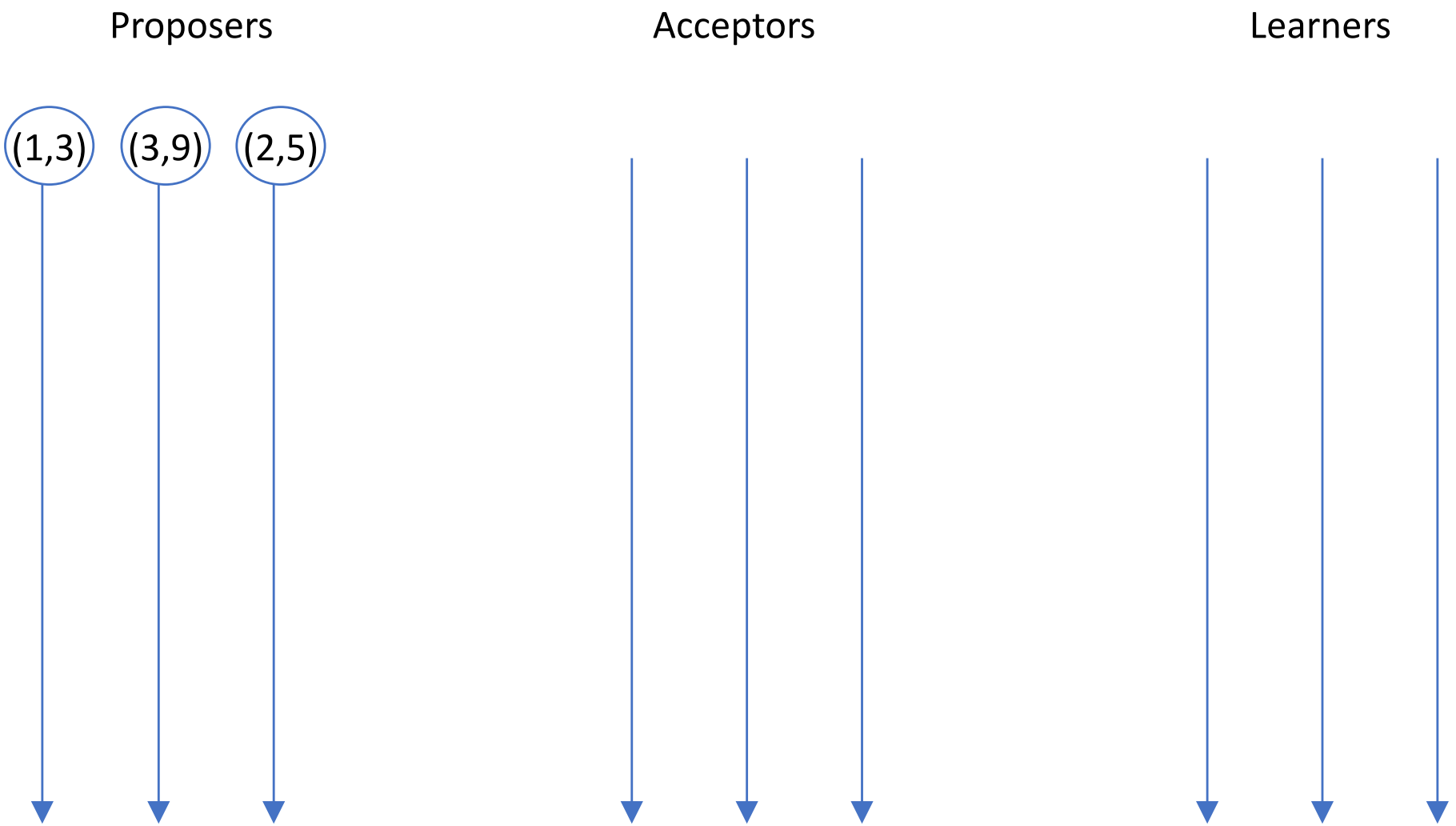
Algorithm for acceptor

```
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 anything <n
     reply <PREPARE_OK,na,va>

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

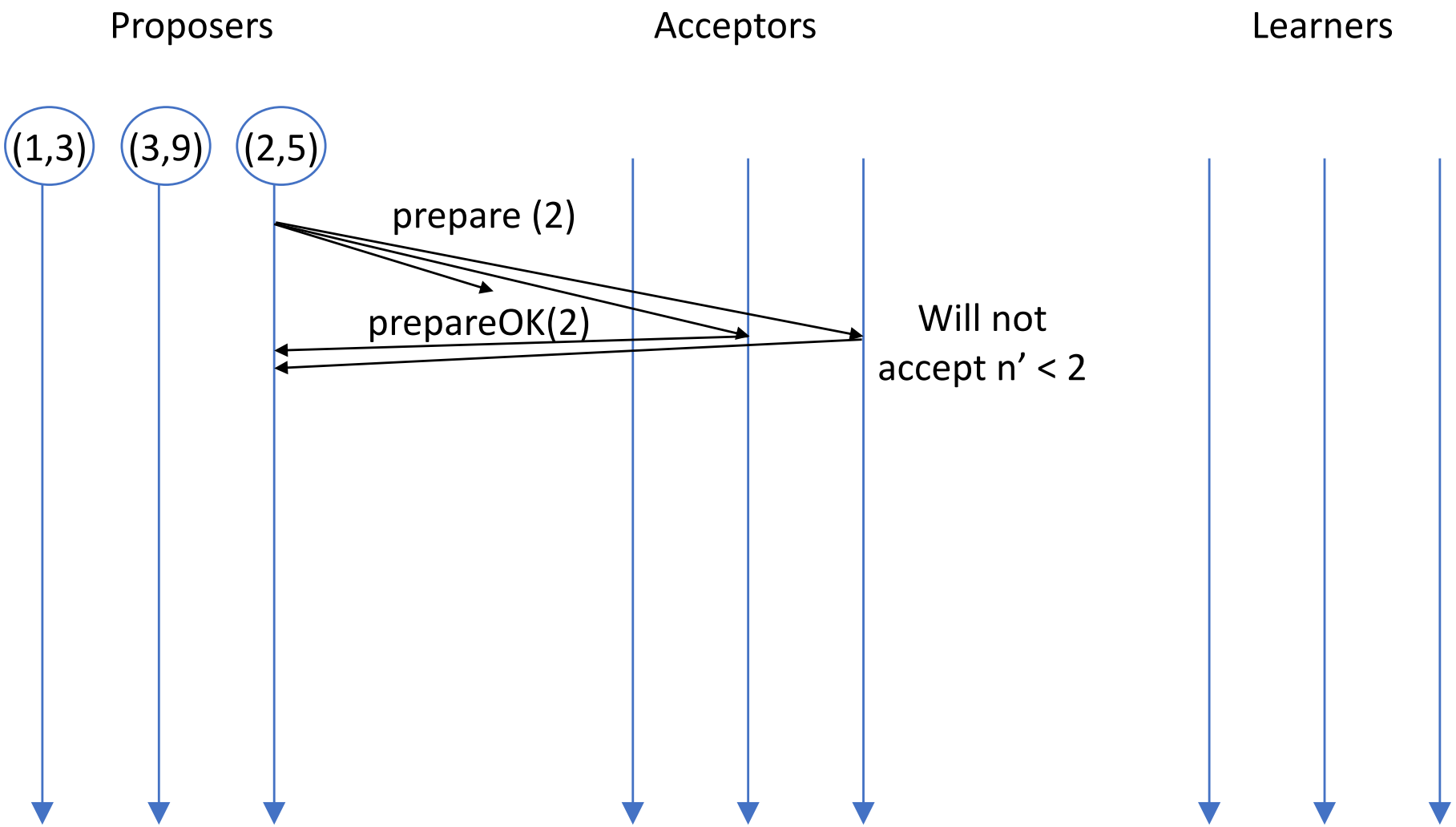
Algorithm for learner

```
State: decision, na, va, aset

      // receive message ACCEPT_OK from acceptor a
ACCEPTED(n,v) from a
  if n > na
     na = n
     va = v
     aset.reset()
  else if n < na
     return
  aset.add(a)
  if aset is a (majority) quorum
     decision = va
```

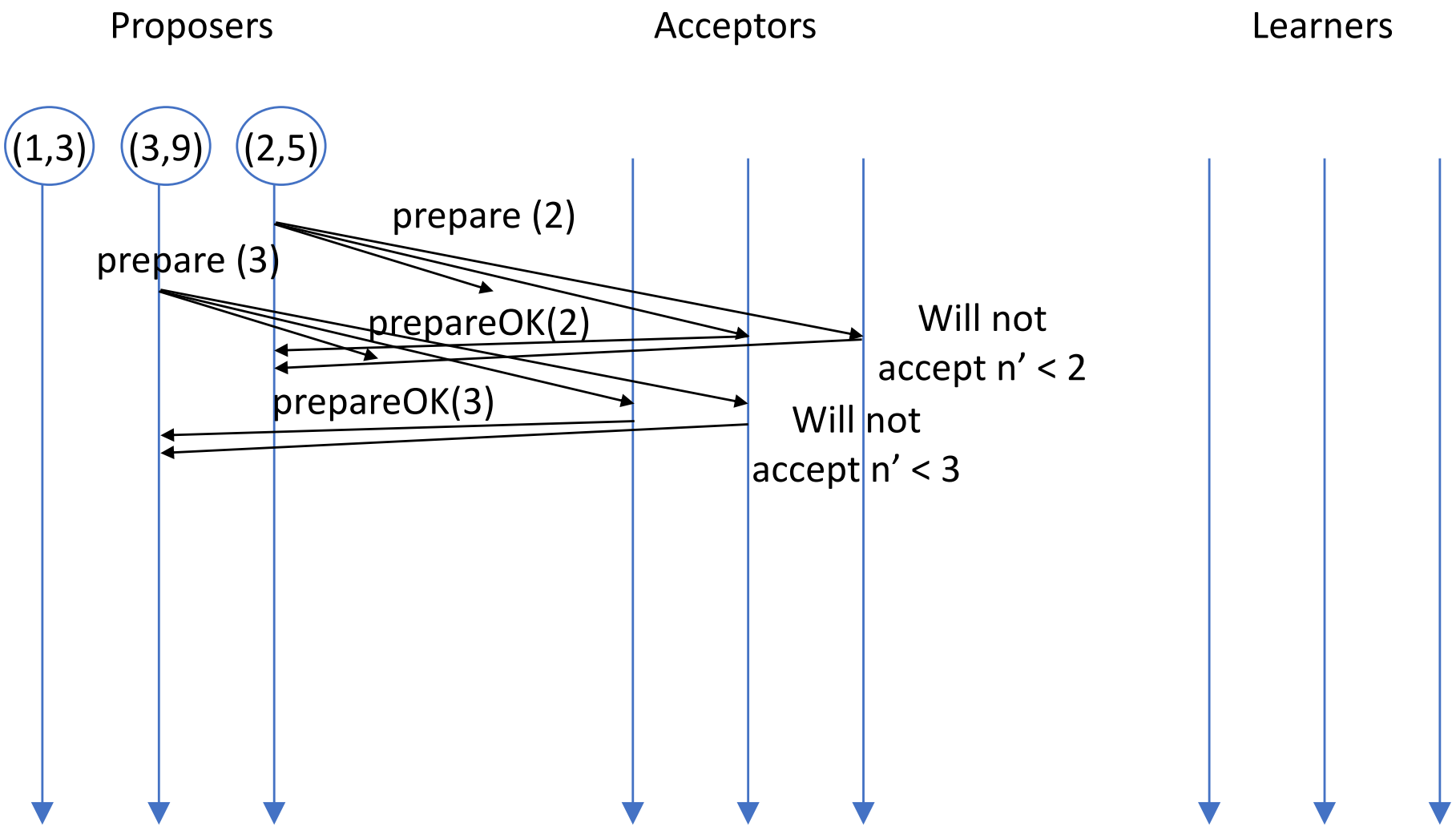
Paxos running



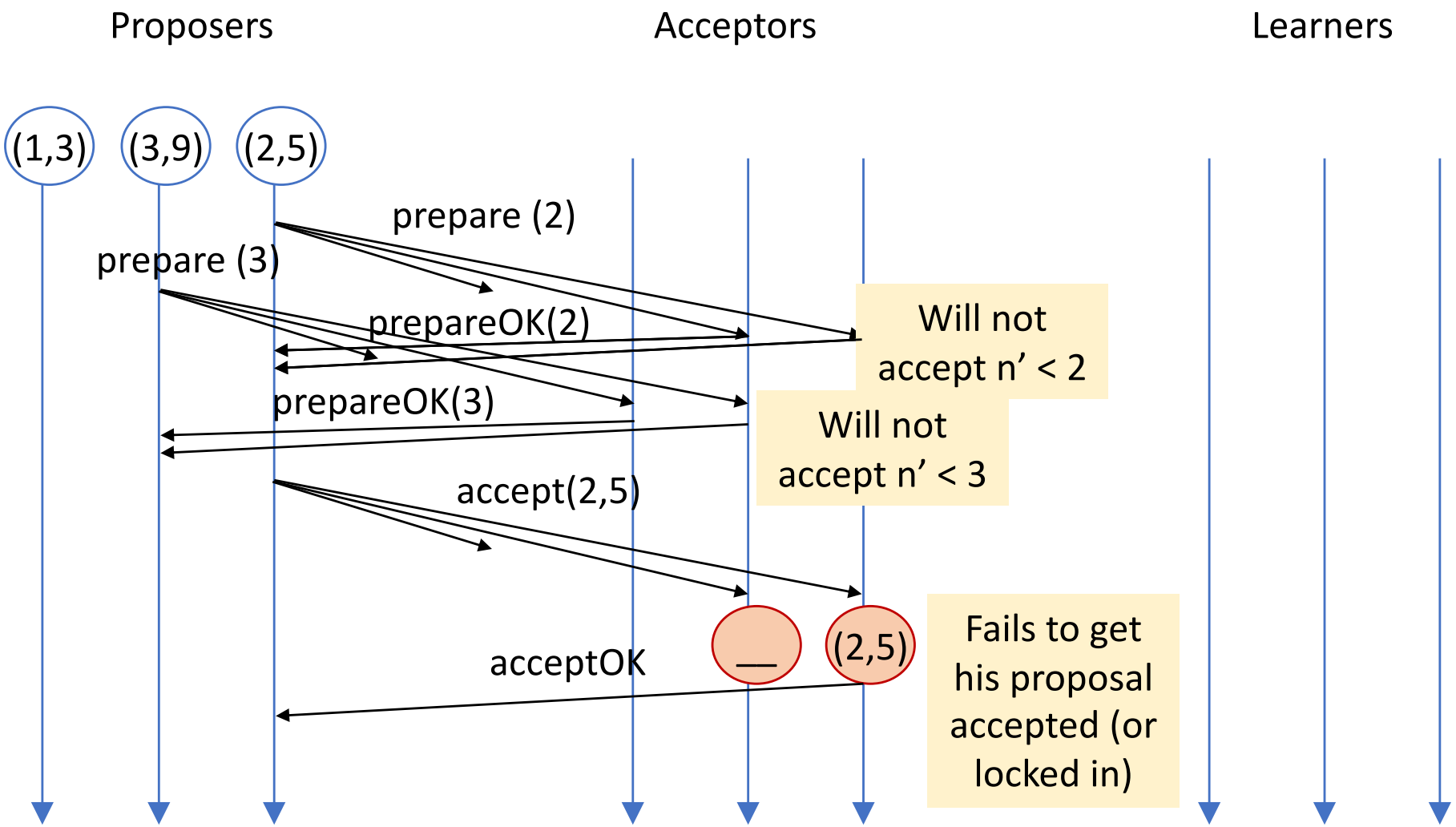
Paxos running



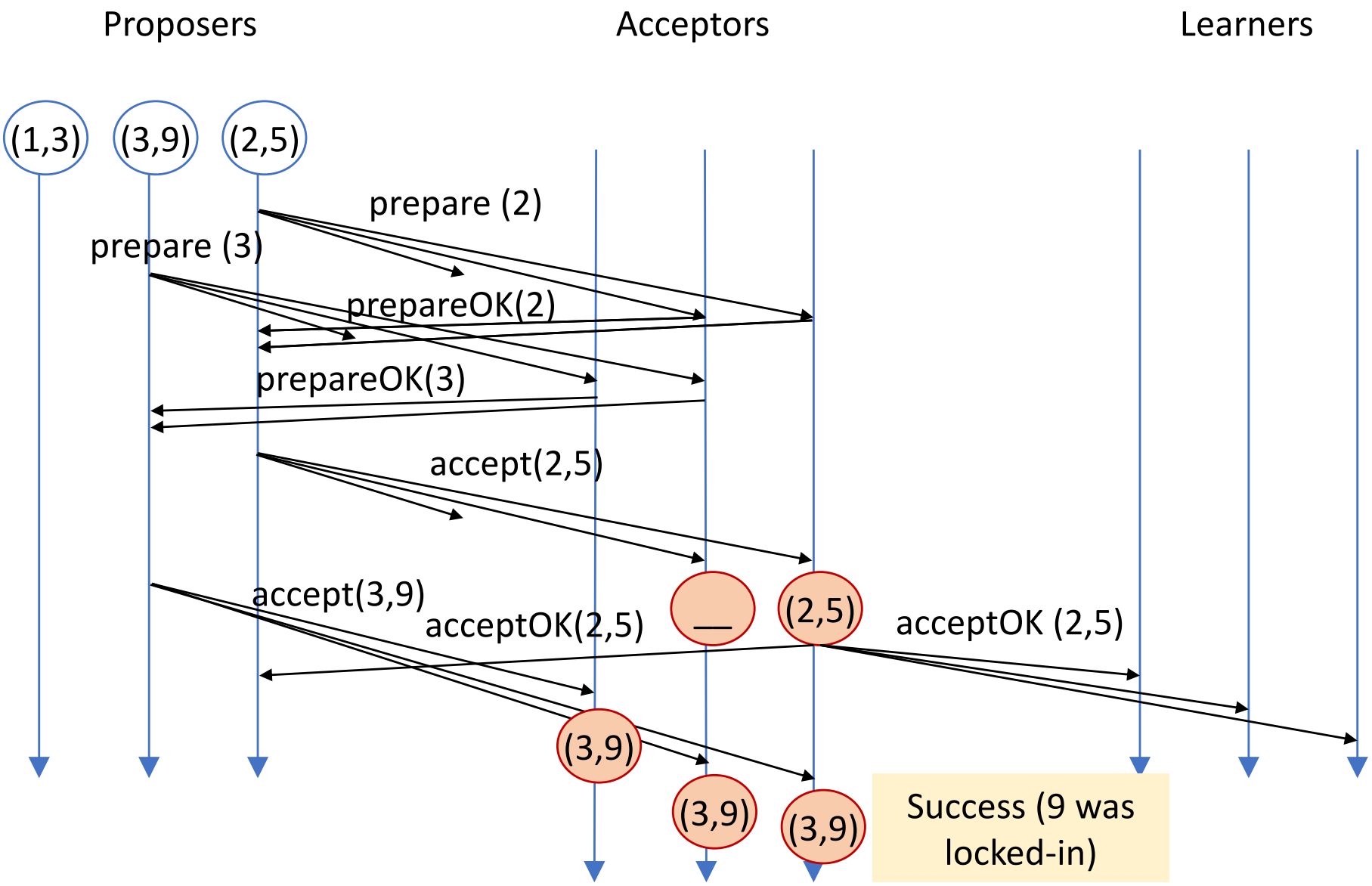
Paxos running



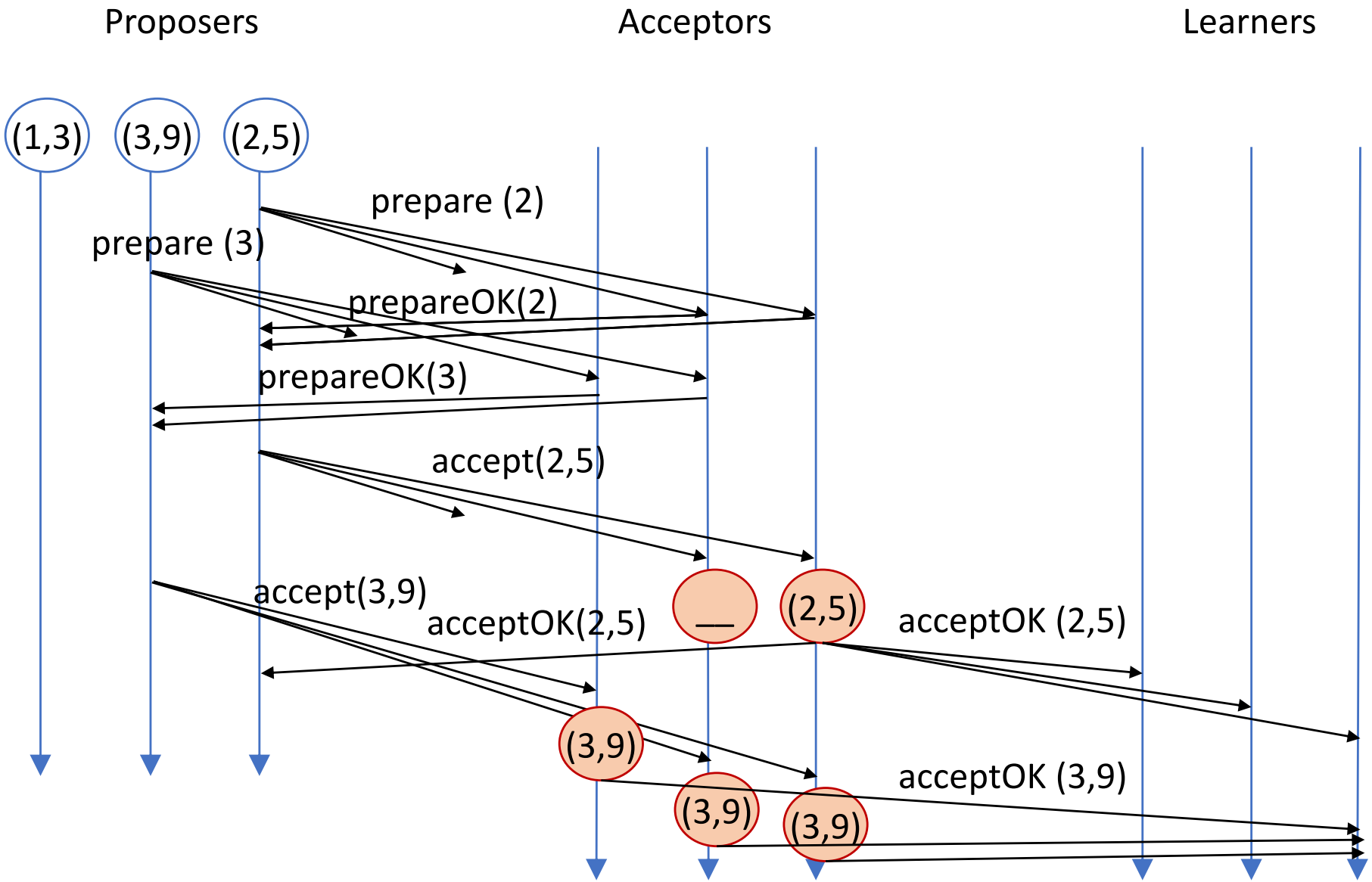
Paxos running



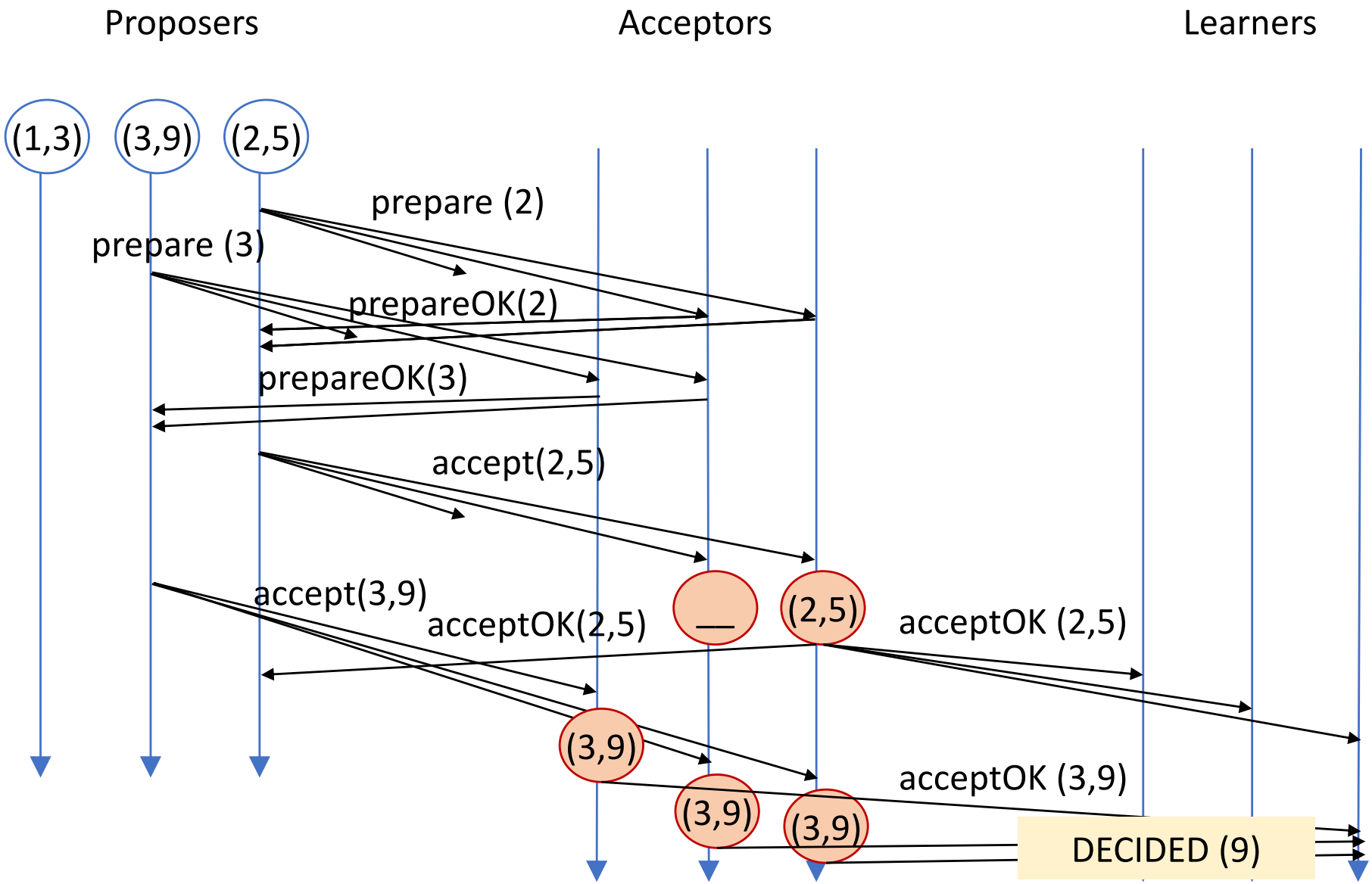
Paxos running



Paxos running



Paxos running



Propagation of inforamtion to Learners...

1. Whenever an Acceptor accepts a value it sends that value (and the sequence number) to the Learners
2. Proposers send information to Learners when they know that there is a value locked-in (i.e., when they gather a majority quorum of ACCEPT_OK)
3. Learners contact acceptors periodically to know which values have been accepted (until they obtain a majority quorum of consistent decisions)

Evidently these different approaches do have trade-offs...

Applying Paxos to State Machine Replication

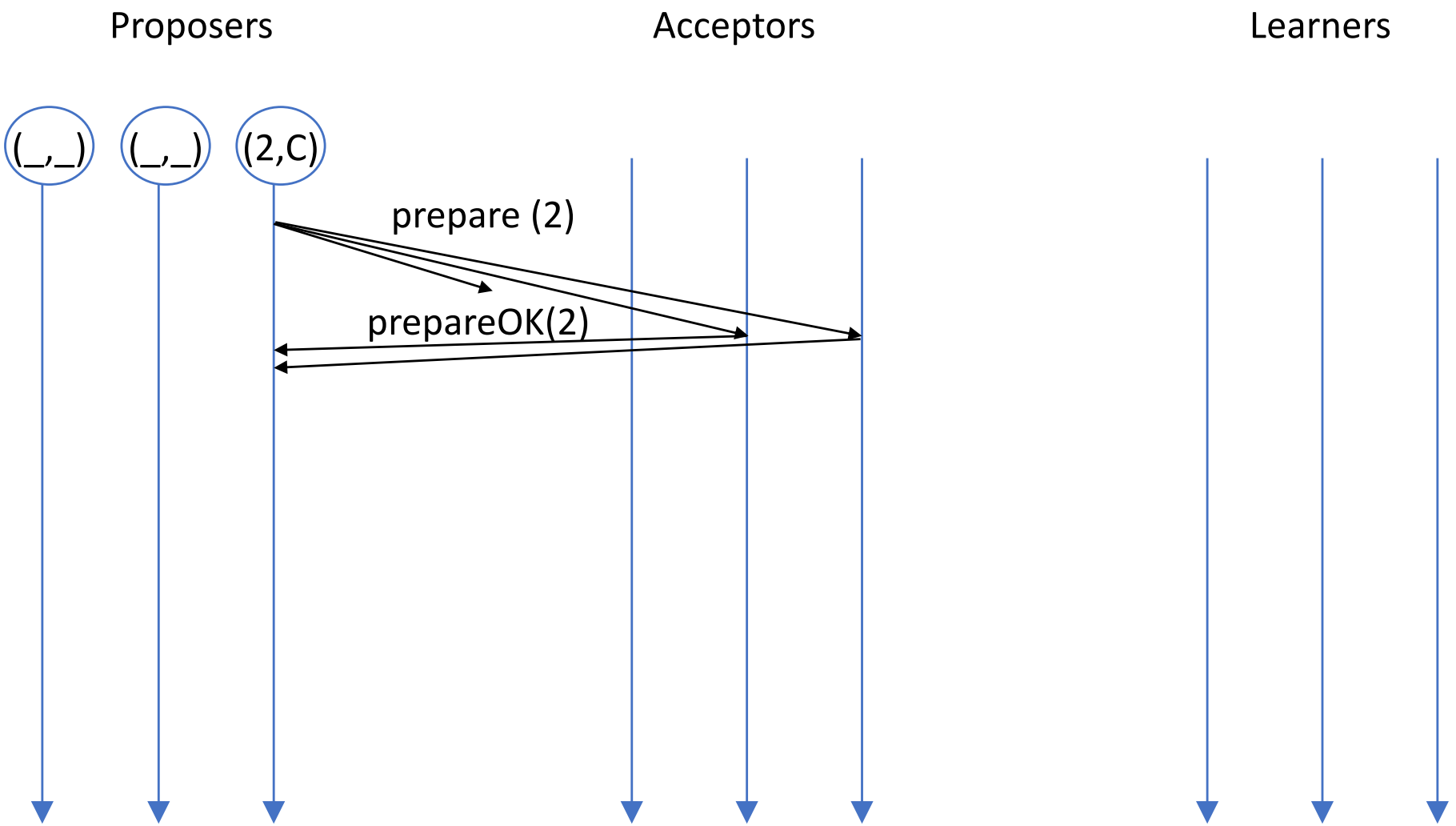
- How can we do this?

Applying Paxos to State Machine Replication

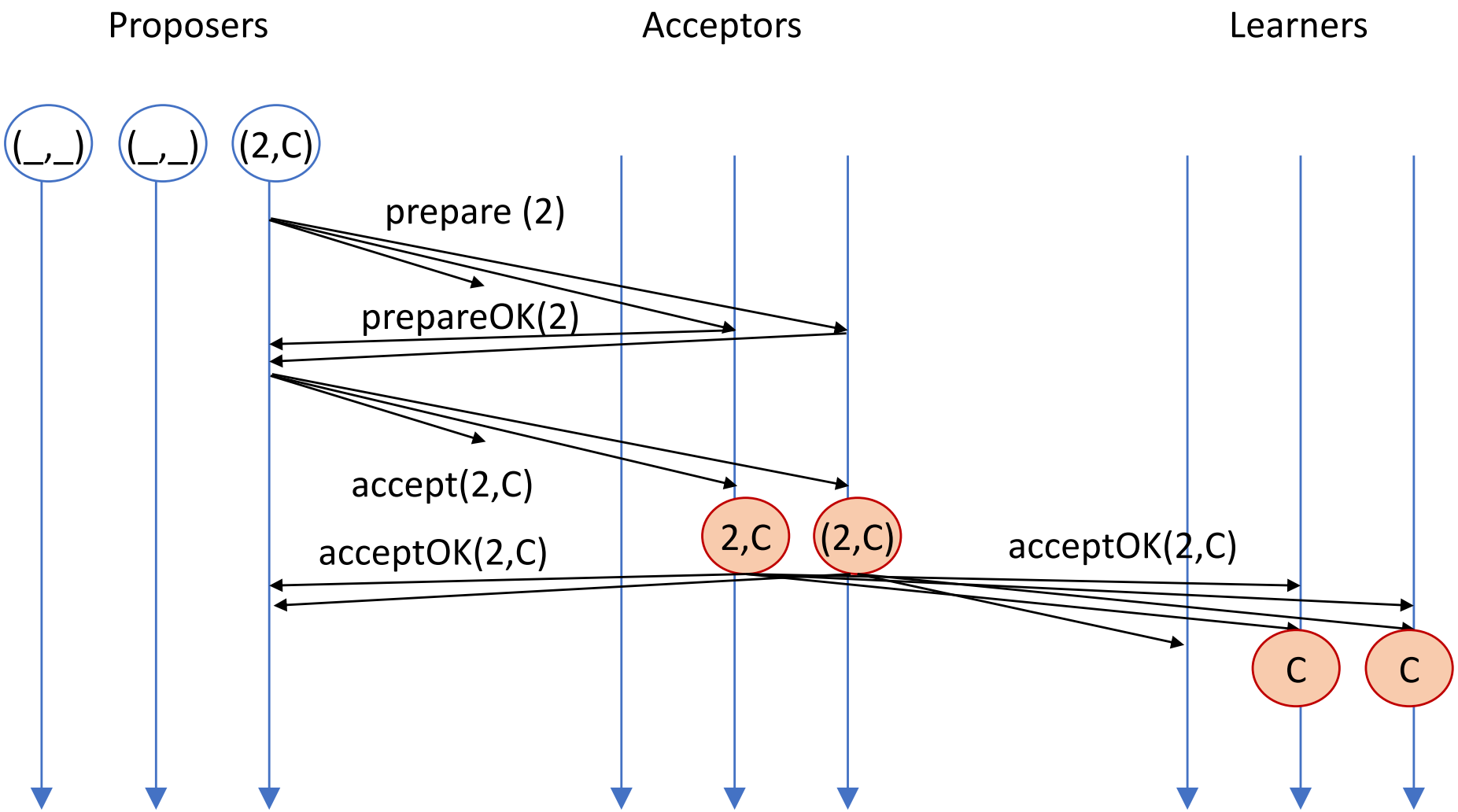
- How can we do this?

- First intuition: We use Paxos to decide a command to execute, and upon deciding, we execute the command.
- Then we use Paxos again to decide the following command to execute, and upon deciding, we execute the command…
- ... And so on and so forth...

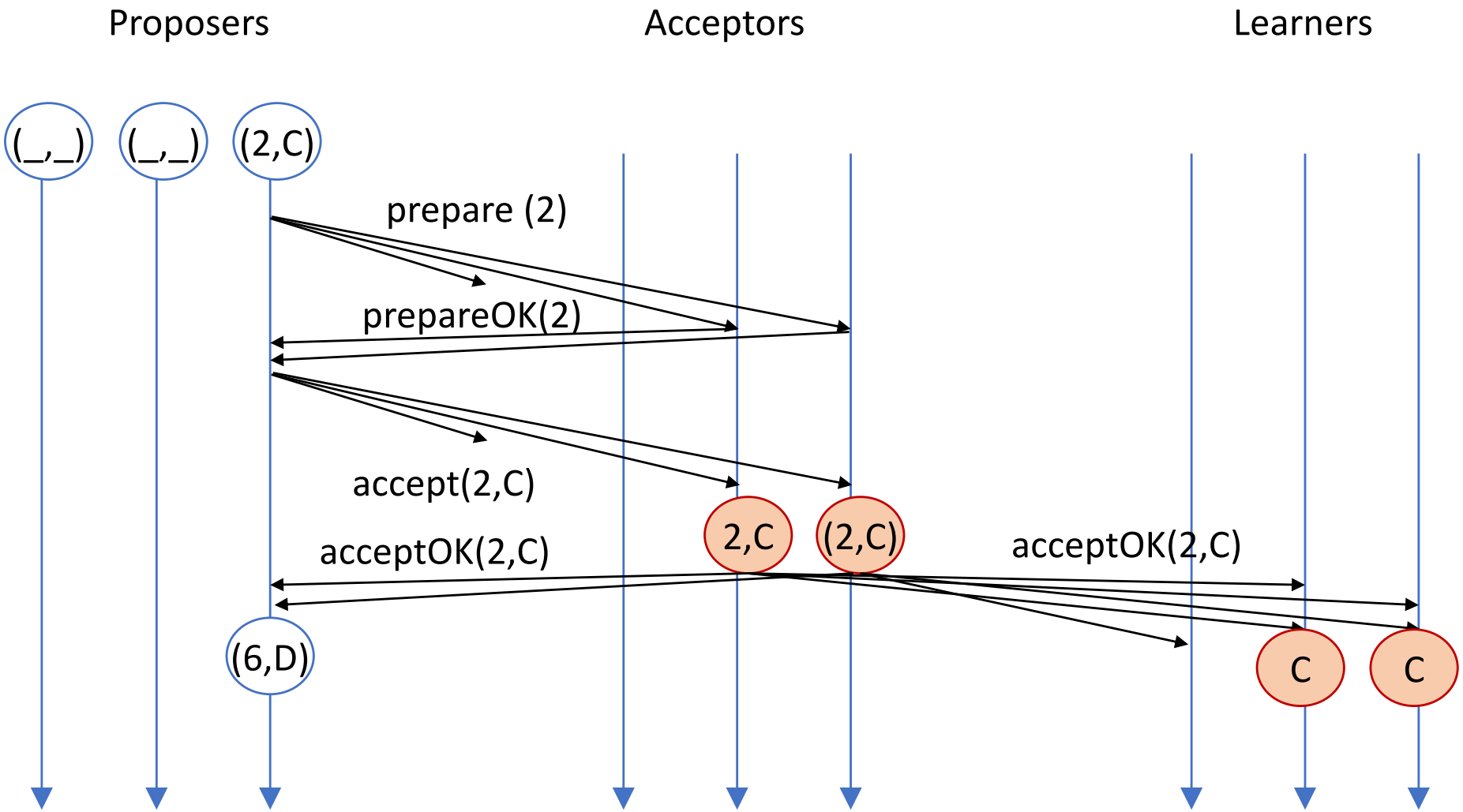
Leveraging Paxos for State Machine Replication



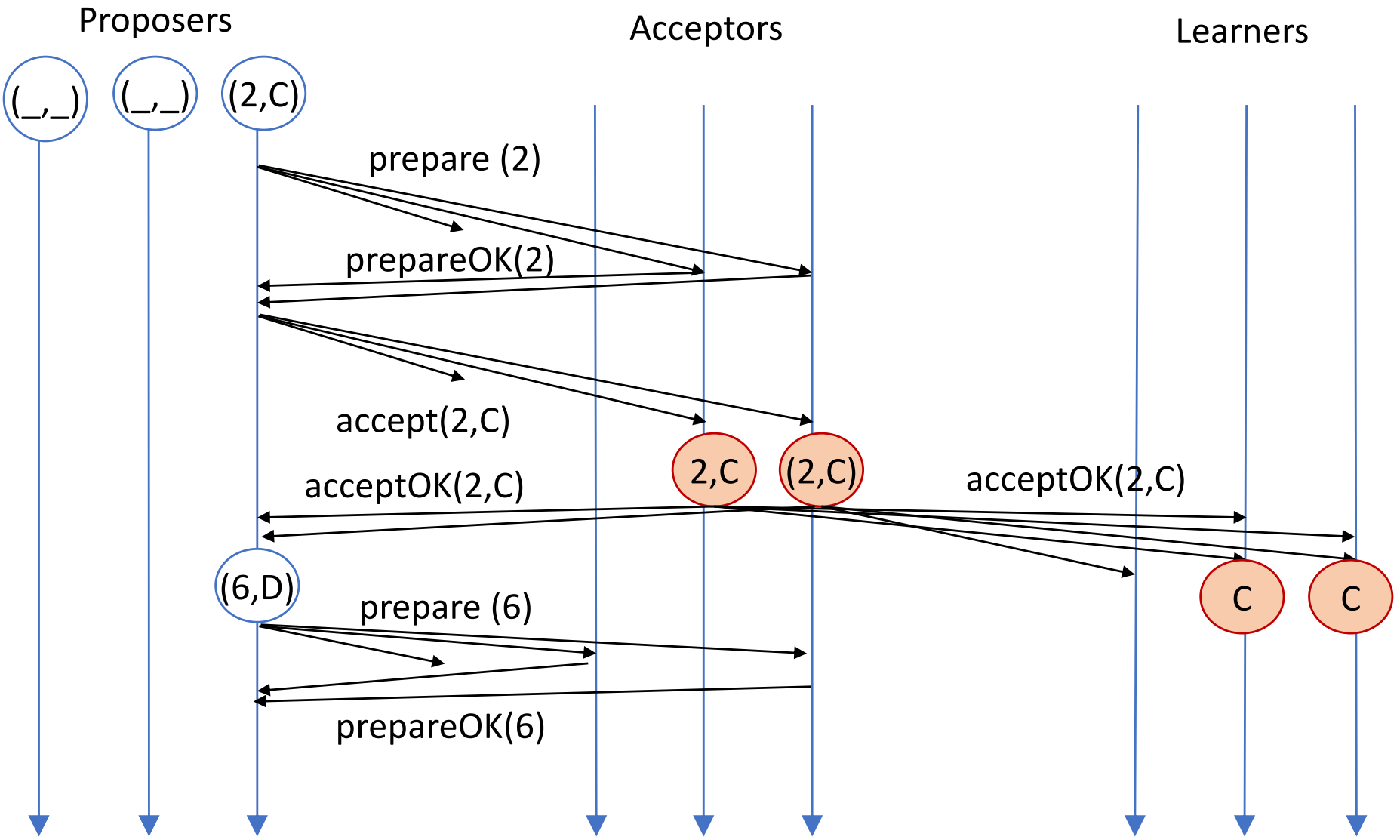
Leveraging Paxos for State Machine Replication



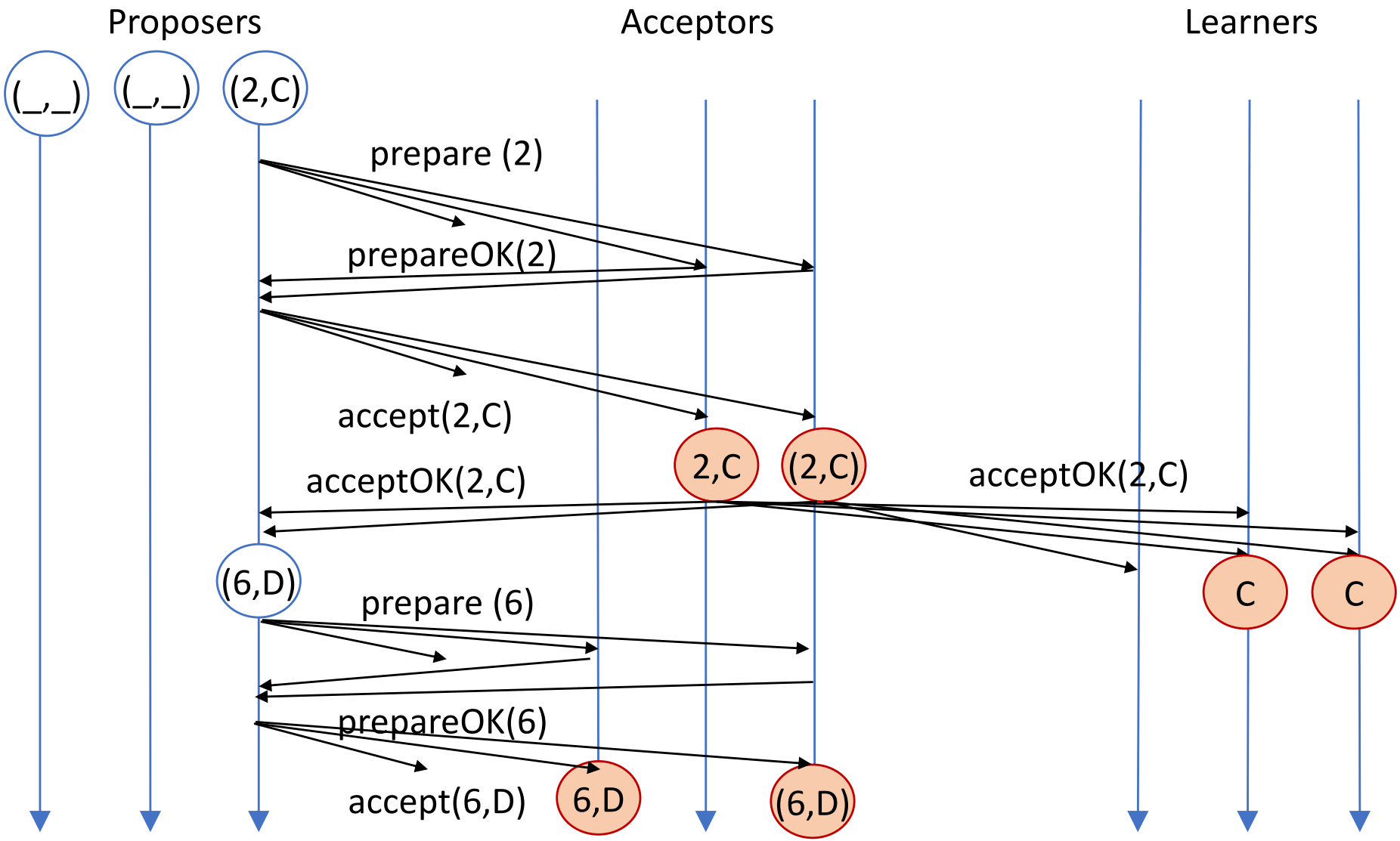
Leveraging Paxos for State Machine Replication



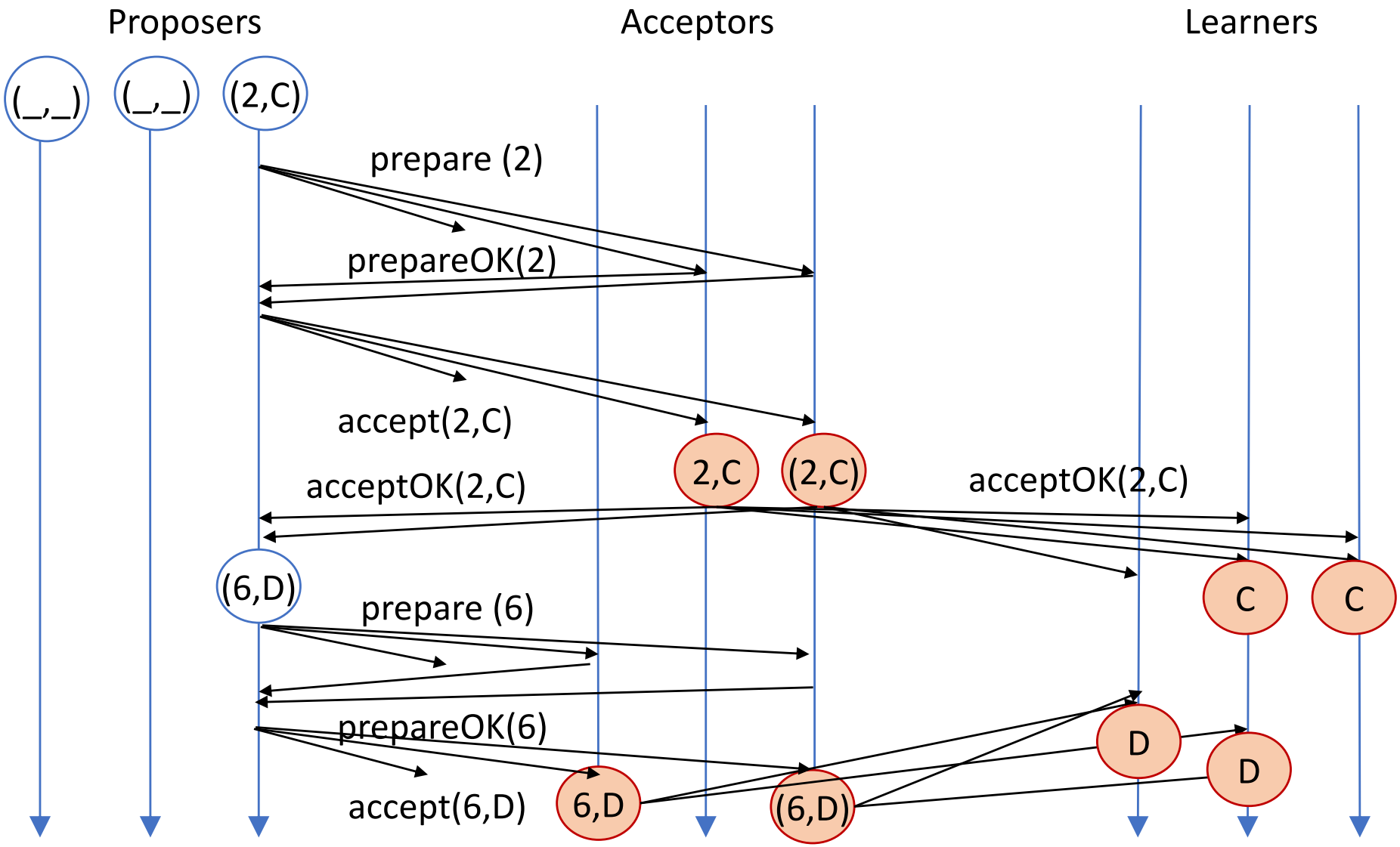
Leveraging Paxos for State Machine Replication



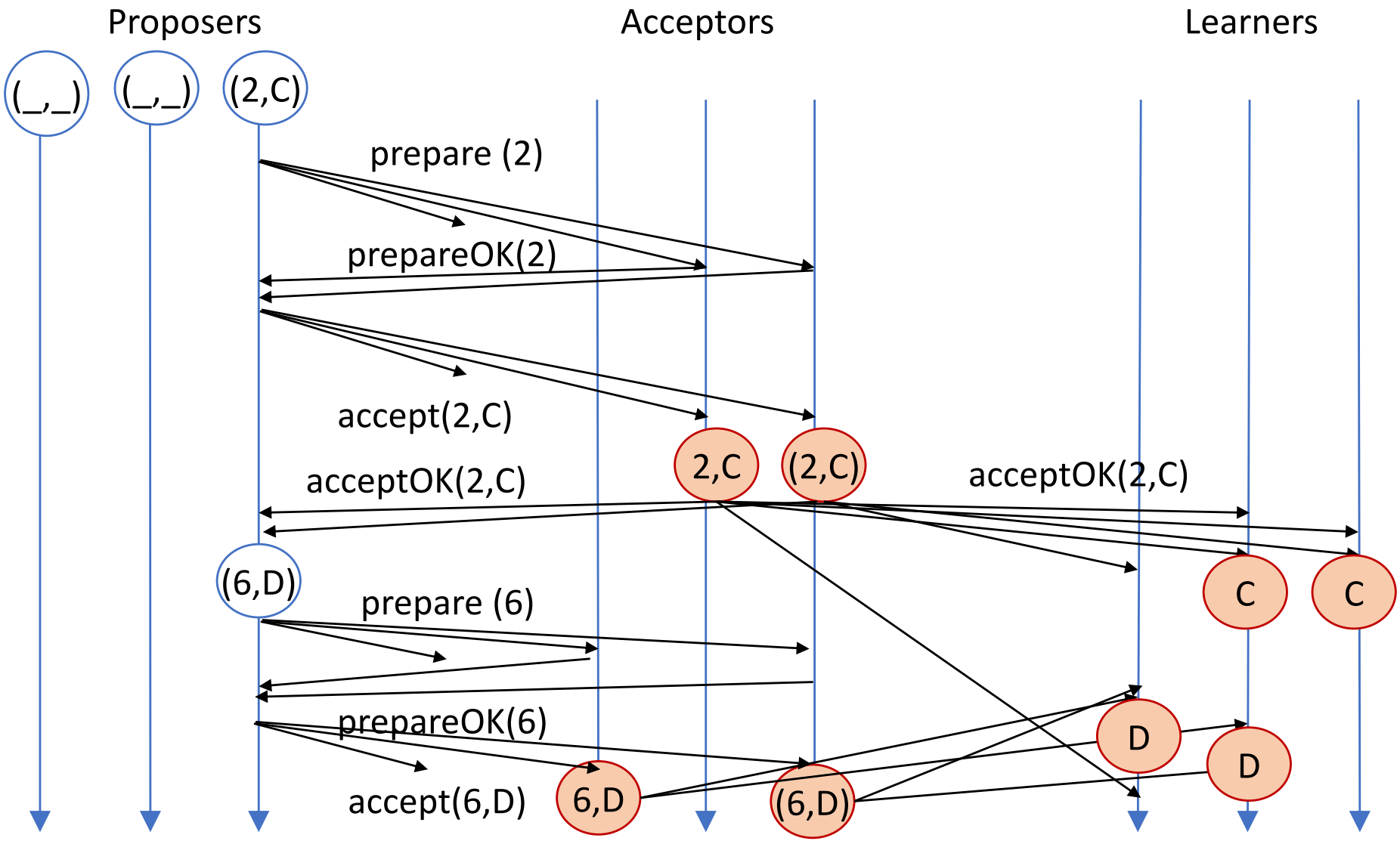
Leveraging Paxos for State Machine Replication



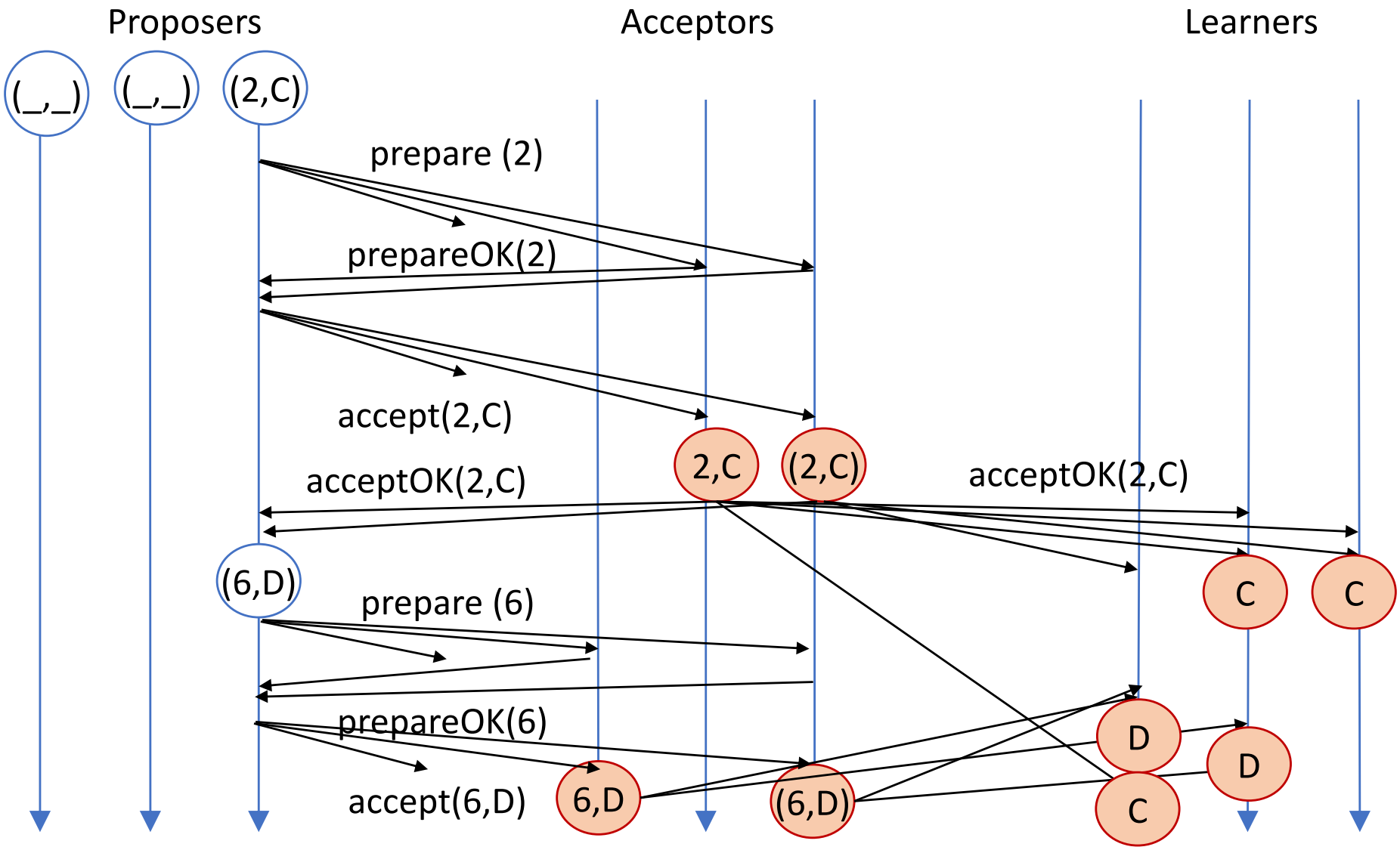
Leveraging Paxos for State Machine Replication



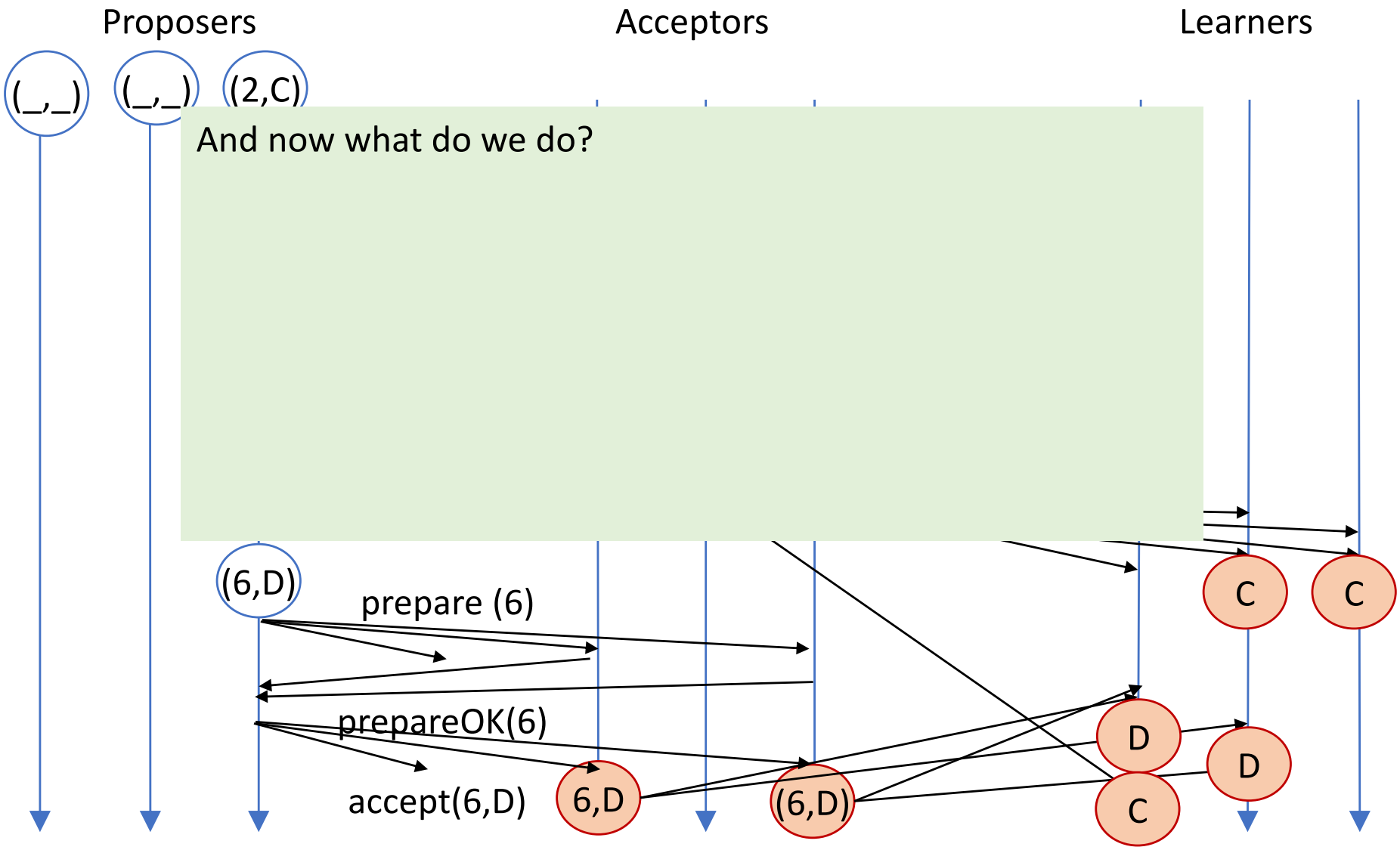
Leveraging Paxos for State Machine Replication



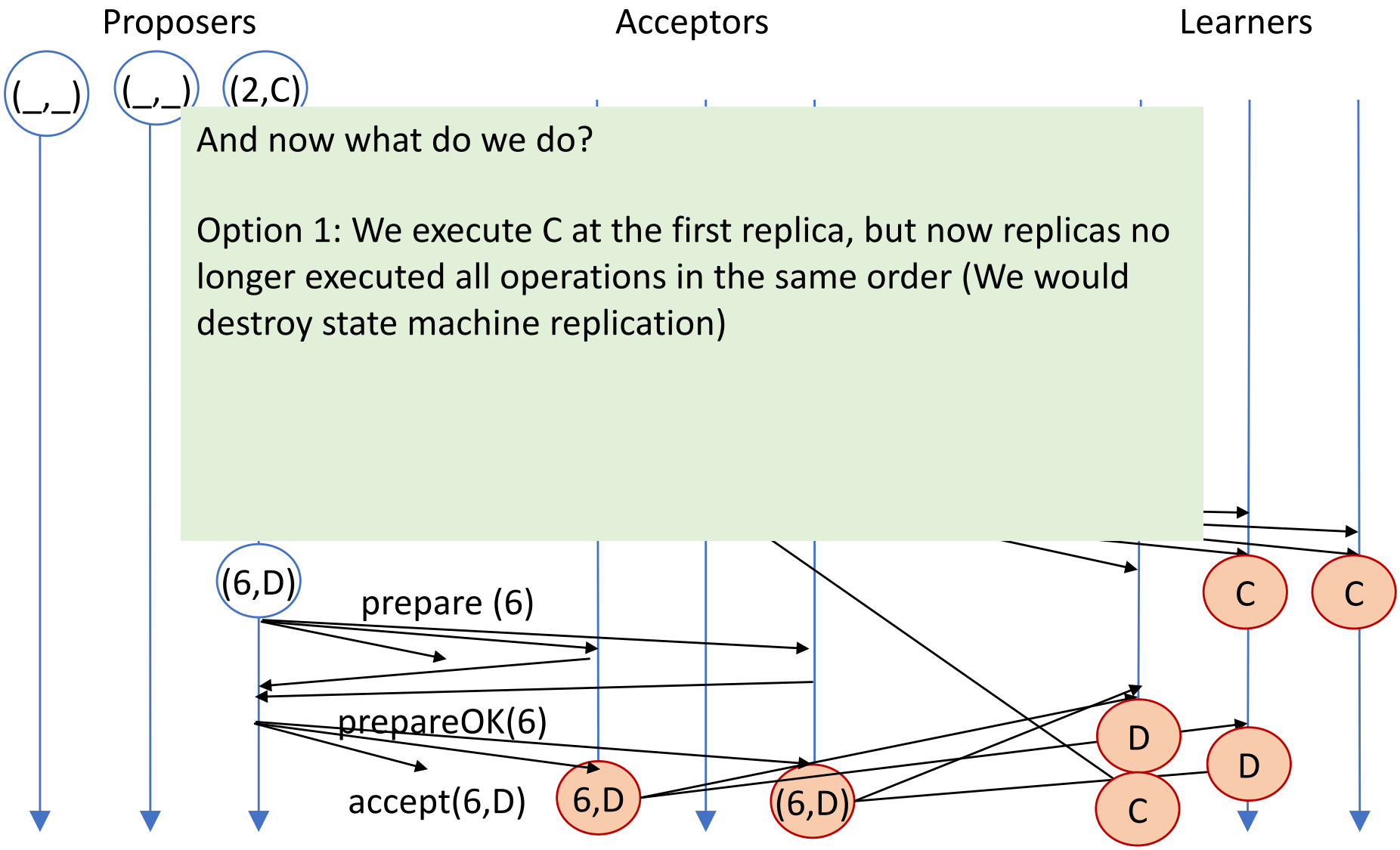
Leveraging Paxos for State Machine Replication



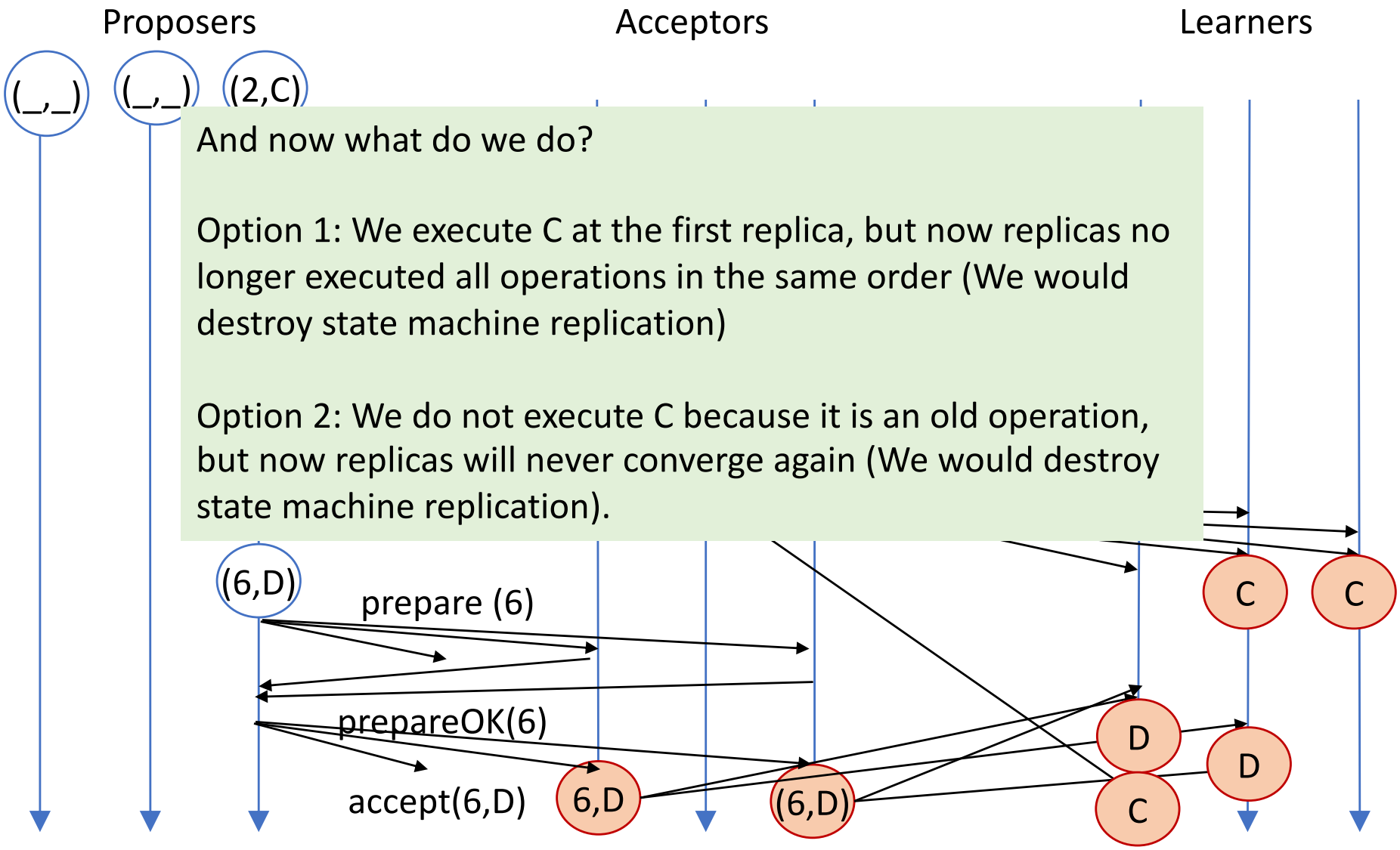
Leveraging Paxos for State Machine Replication



Leveraging Paxos for State Machine Replication



Leveraging Paxos for State Machine Replication



Leveraging Paxos for State Machine Replication

- What was the problem is the previous example?

Leveraging Paxos for State Machine Replication

- What was the problem is the previous example?
- Fundamentally, the problem arises because Paxos operates with majority quorums. However in state machine replication **all replicas** have to execute all operations in order. This must happen independently of that replica participating or not in the quorum that decided a given operation.

- **How can we address this issue?**

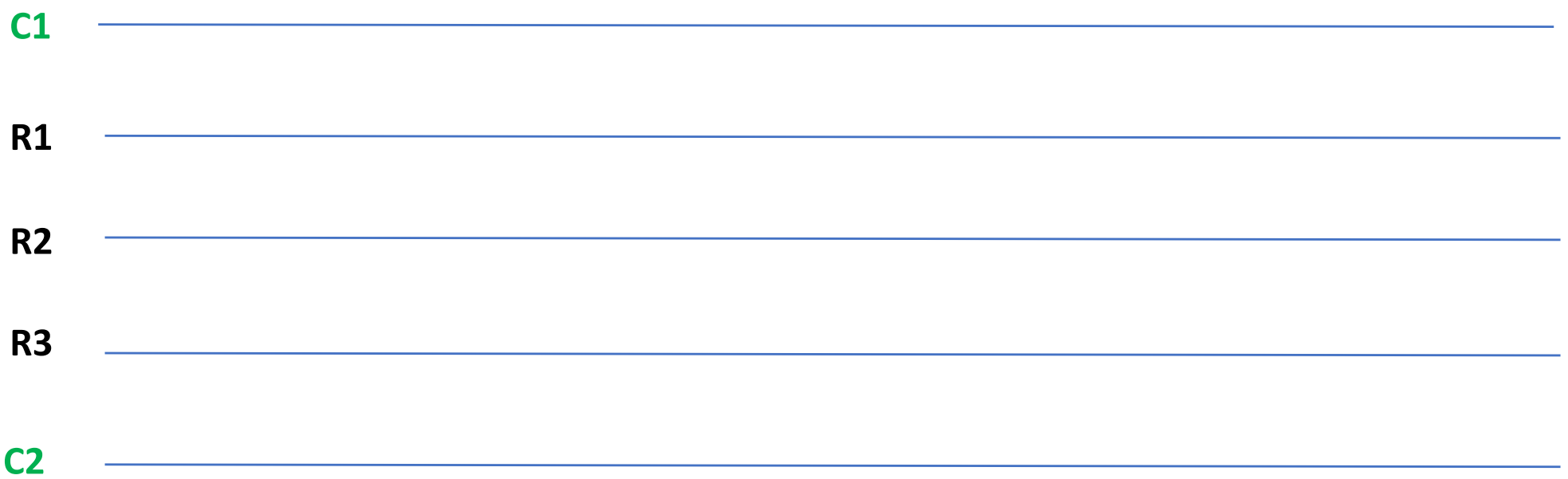
Leveraging Paxos for State Machine Replication

- Assume that there is an infinite sequence of commands, that are numbered sequentially, from 0 to infinity.
- Instead of using Paxos to decide the next operation to be executed, we use an independent instance of Paxos to (sequentially) decide which operation will be executed for each of the positions in the sequence of commands.
- Whenever a value (i.e, a command is decided for position ***n****)* we start the Paxos instance to decide the next command (***n****+1).*
- *Replicas have to execute commands in order* ***strictly following this sequence****, which is not necessarily the order in which they learn decided commands.*

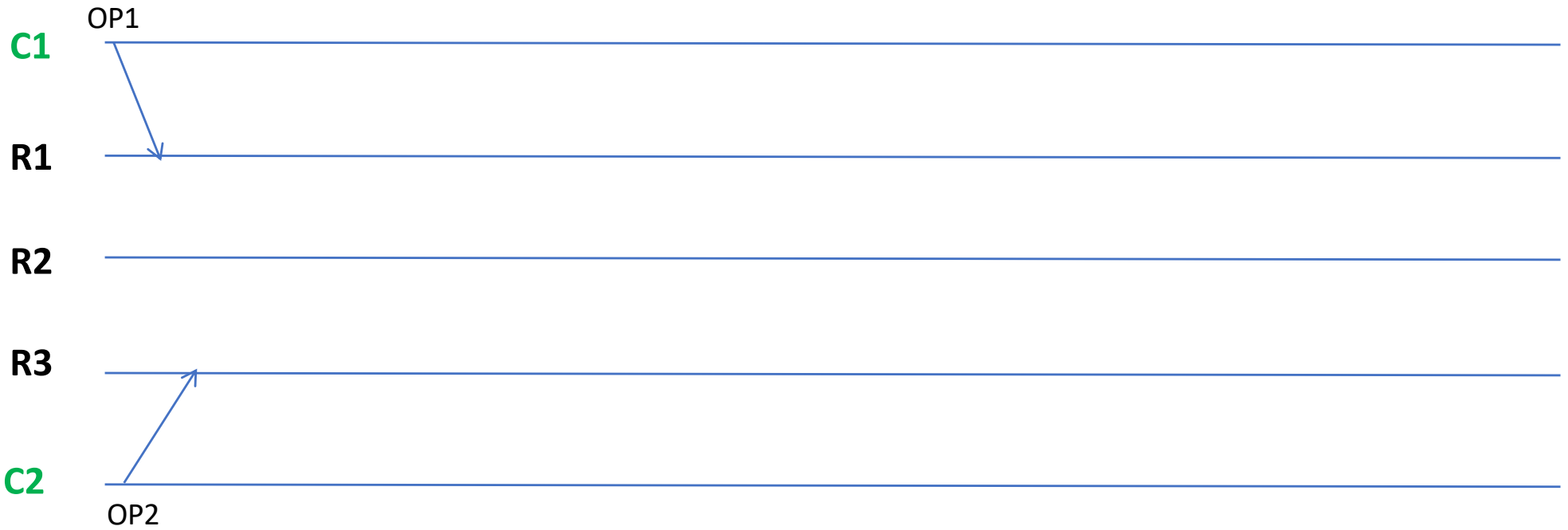
Implementing State Machine Replication with Paxos

- Each replica of the service executes all three roles of Paxos (Proposer, Acceptor, and Learner).
- Client sends operation *op* to replica R
- Replica R proposes the received operation in the next Paxos instance
- If the result of Paxos is the proposed operation, return ok to the client (and eventually the result)
- Otherwise, propose op in the next Paxos instance…

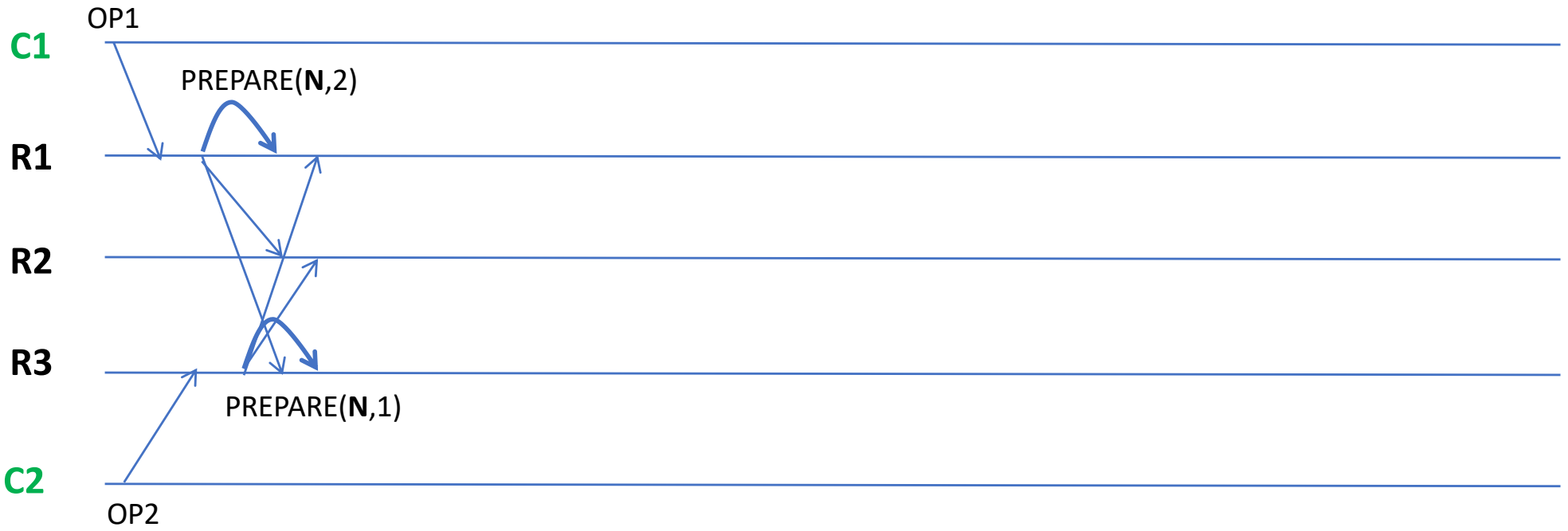
Implementing State Machine Replication with Paxos



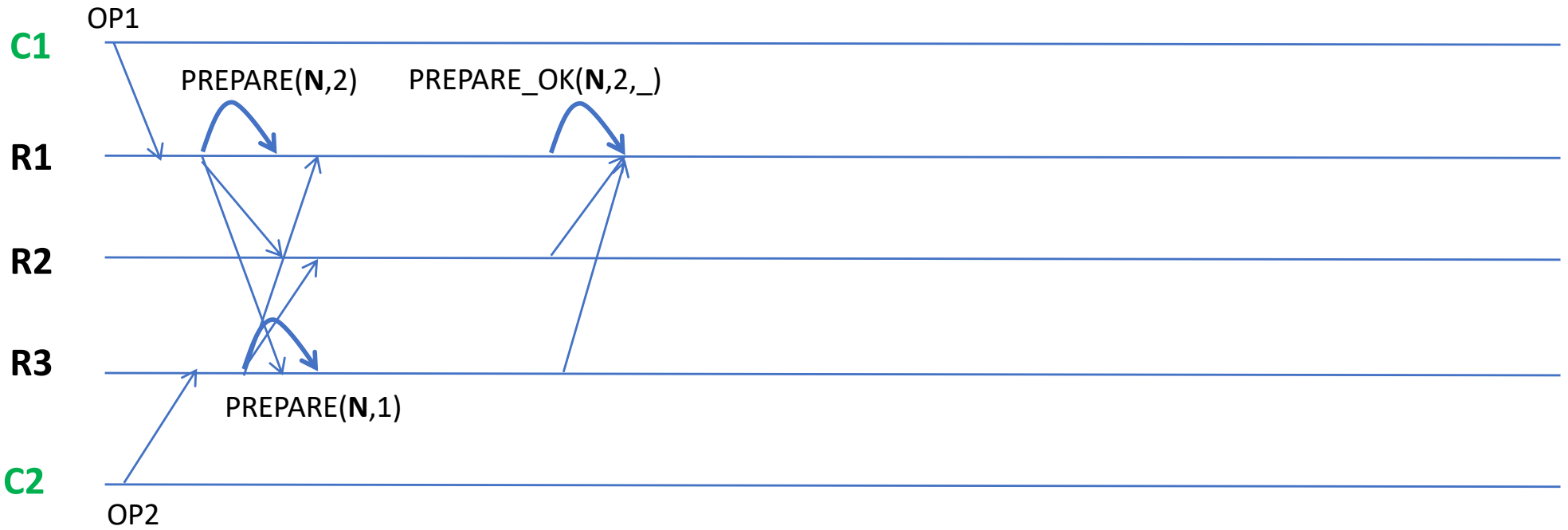
Implementing State Machine Replication with Paxos



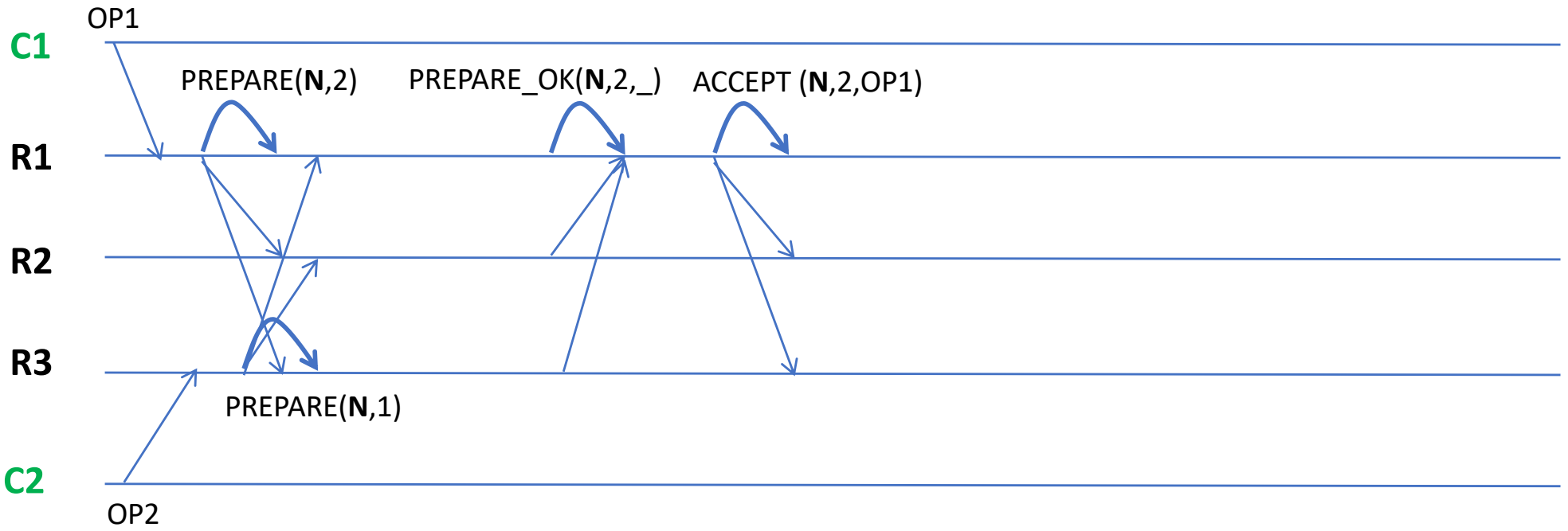
Implementing State Machine Replication with Paxos



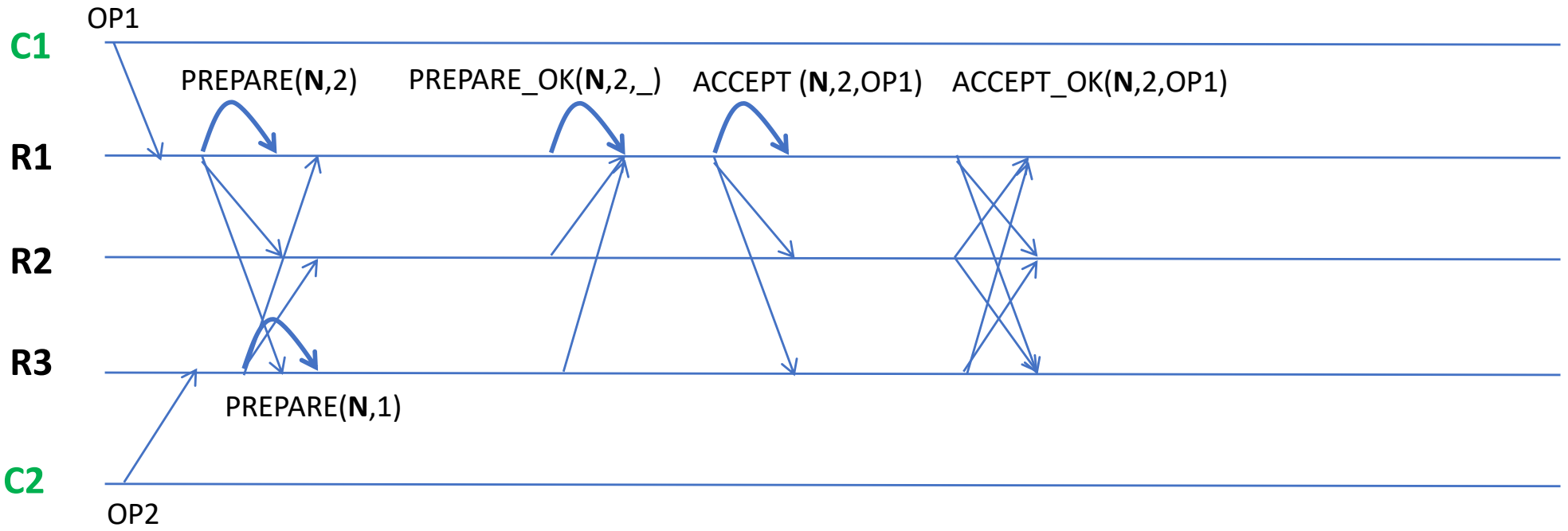
Implementing State Machine Replication with Paxos



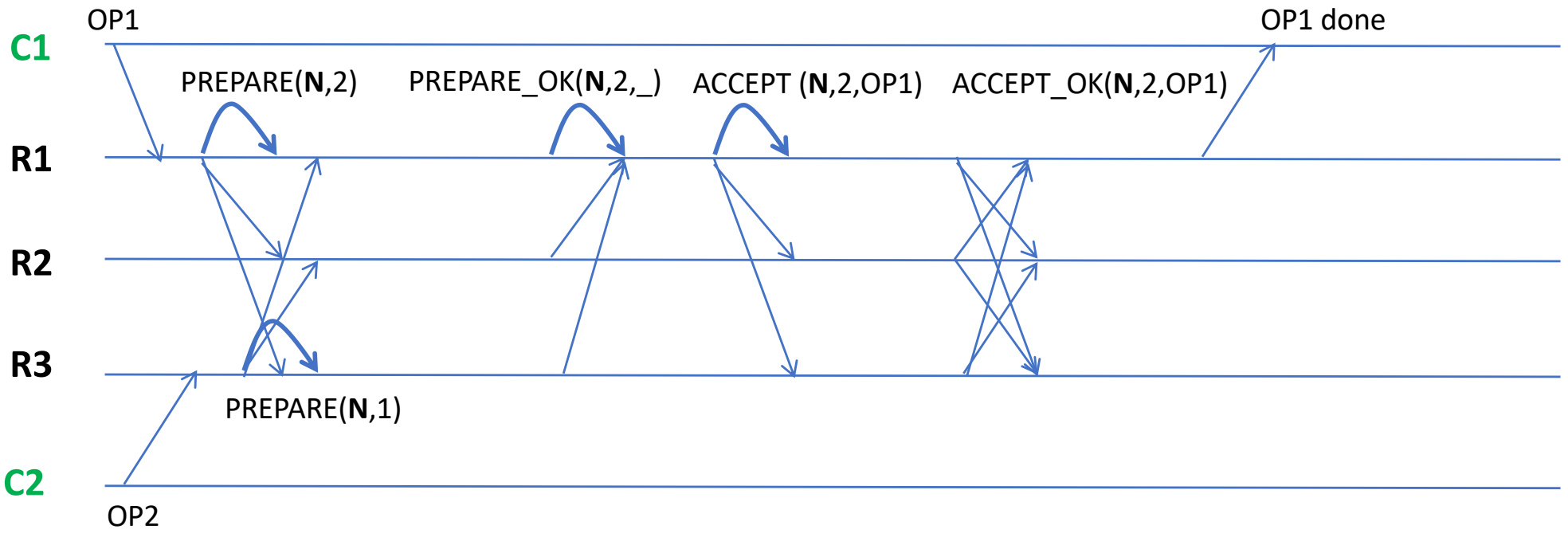
Implementing State Machine Replication with Paxos



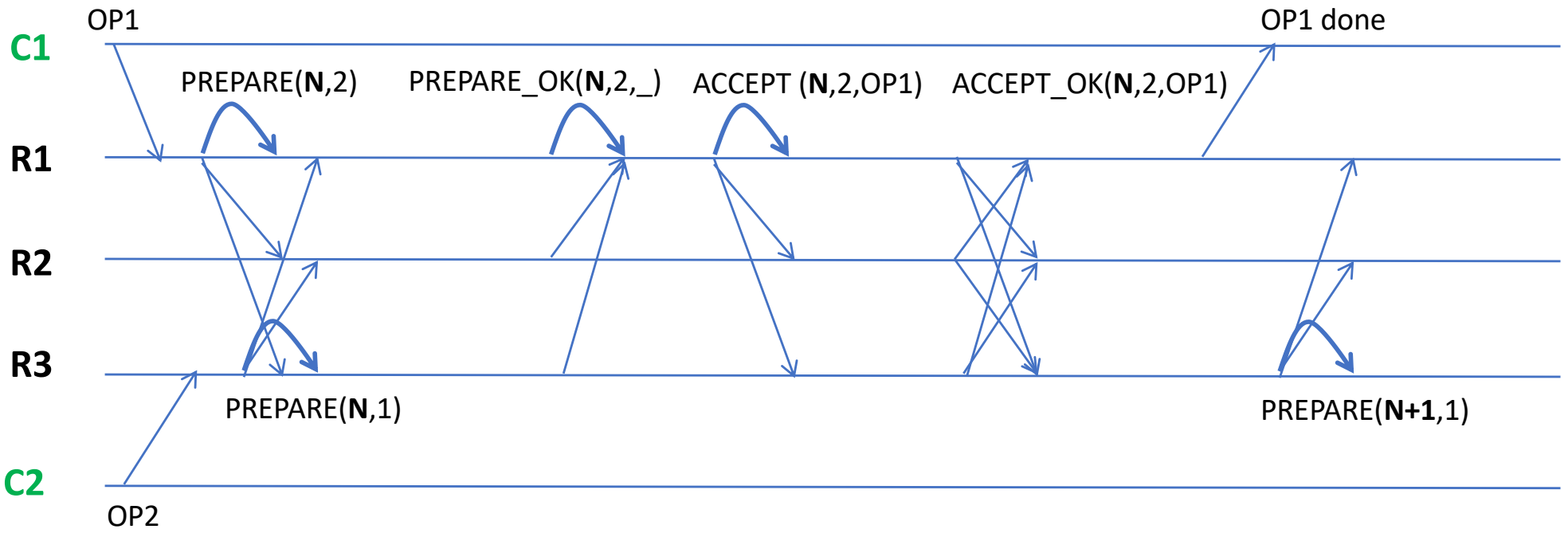
Implementing State Machine Replication with Paxos



Implementing State Machine Replication with Paxos

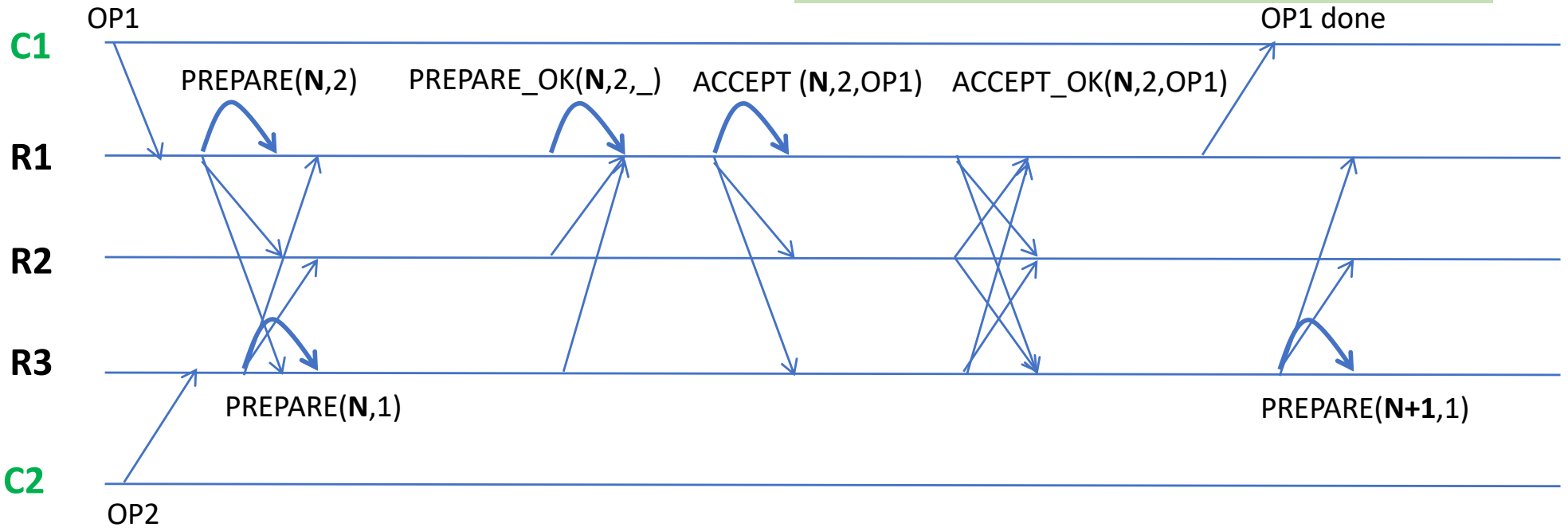


Implementing State Machine Replication with Paxos



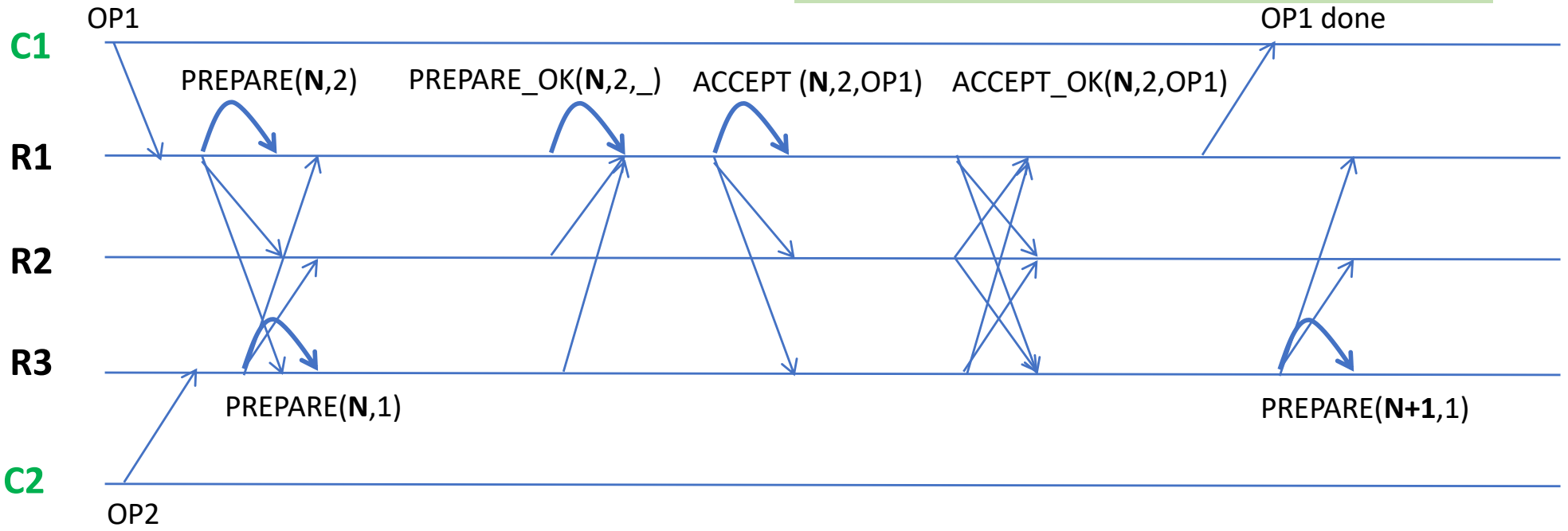
Implementing State Machine Replication with Paxos

OP1 done can also carry the reply, since the replica can execute OP1 locally before replying…



Implementing State Machine Replication with Paxos

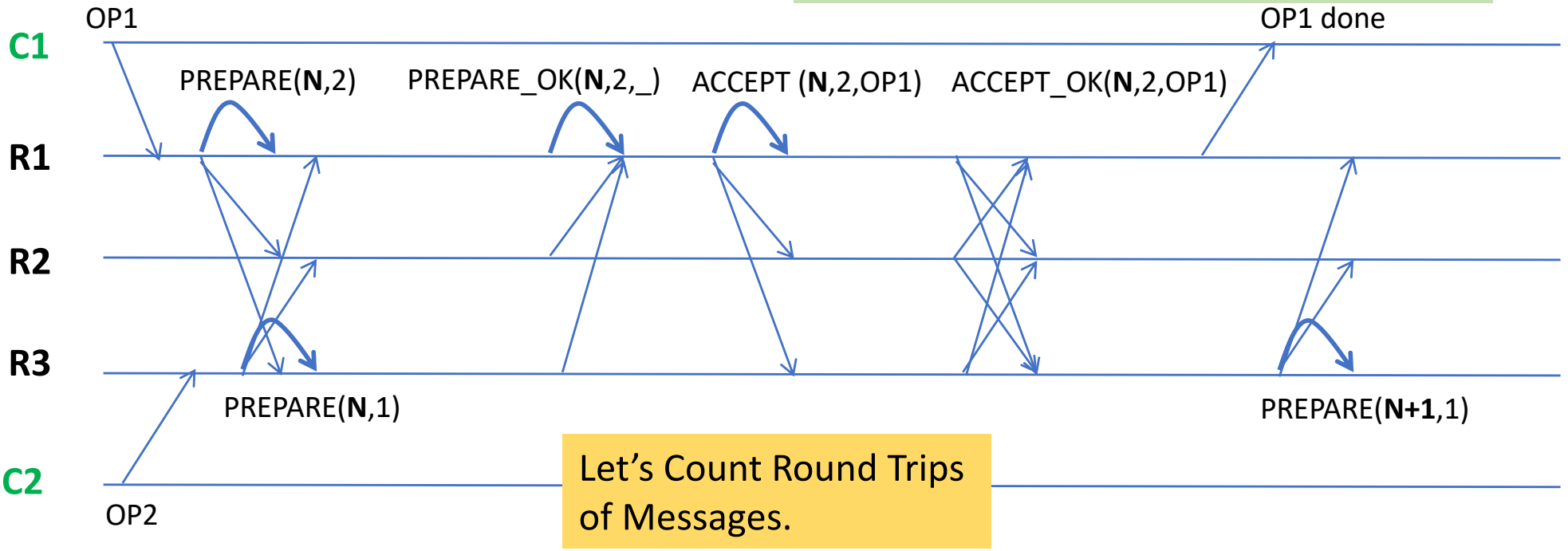
OP1 done can also carry the reply, since the replica can execute OP1 locally before replying…



For Client 1, this is the good case, in which his operation is decided in the consensus instance following the reception of his request. Is this good?

Implementing State Machine Replication with Paxos

OP1 done can also carry the reply, since the replica can execute OP1 locally before replying…

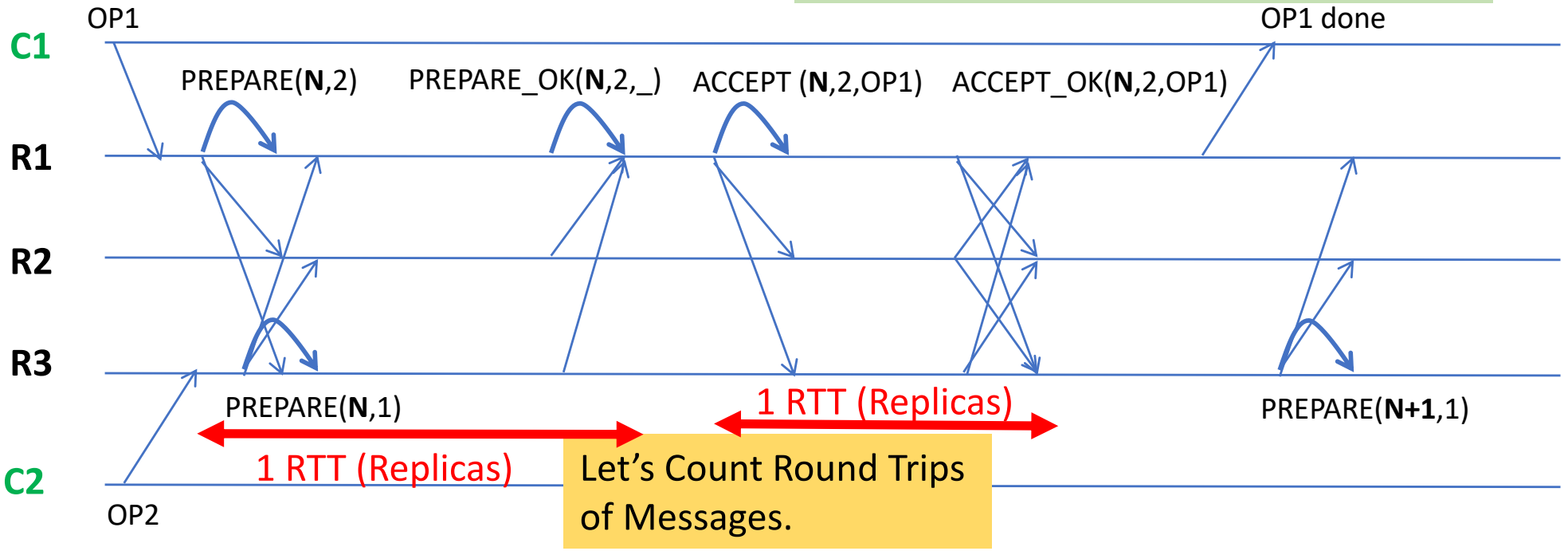


Let's Count Round Trips of Messages.

For Client 1, this is the good case, in which his operation is decided in the consensus instance following the reception of his request. Is this good?

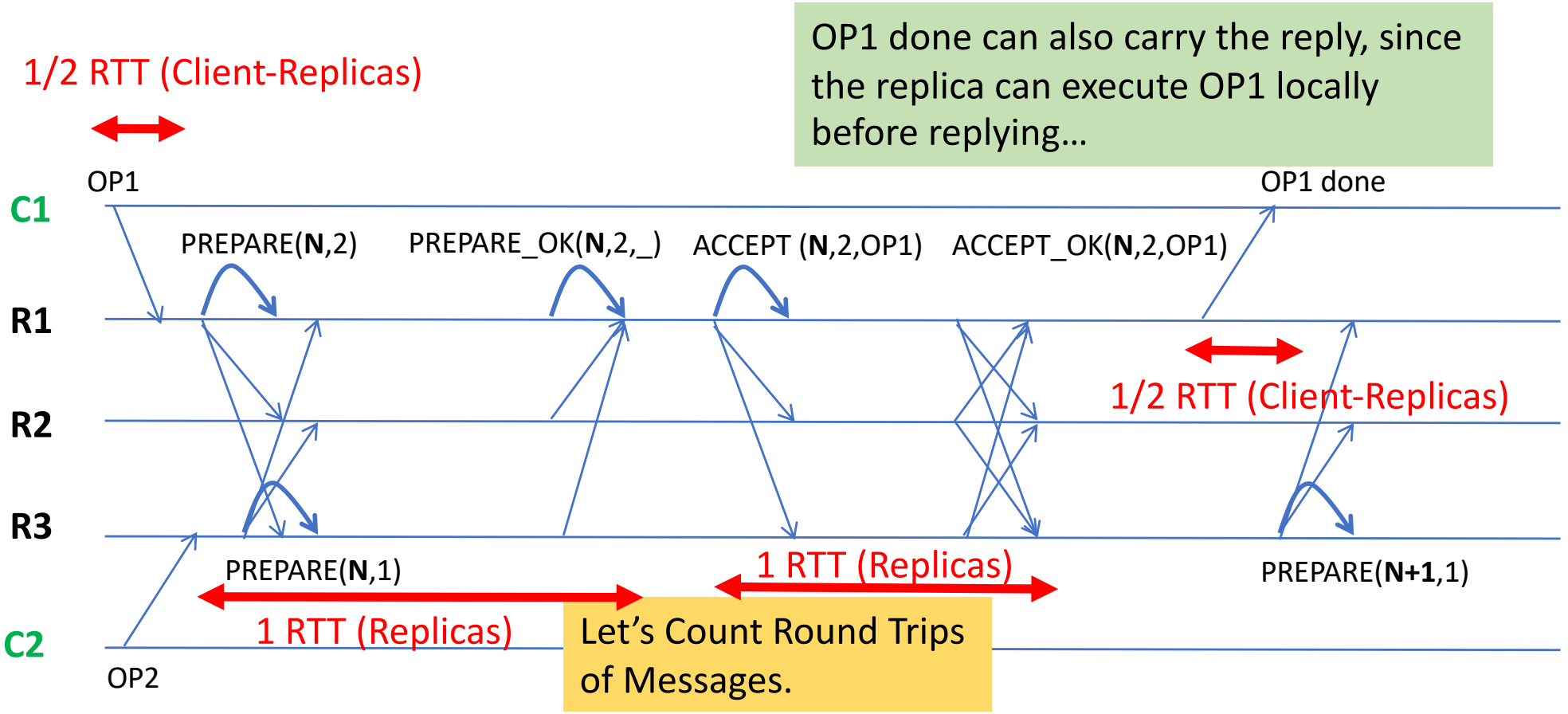
Implementing State Machine Replication with Paxos

OP1 done can also carry the reply, since the replica can execute OP1 locally before replying…



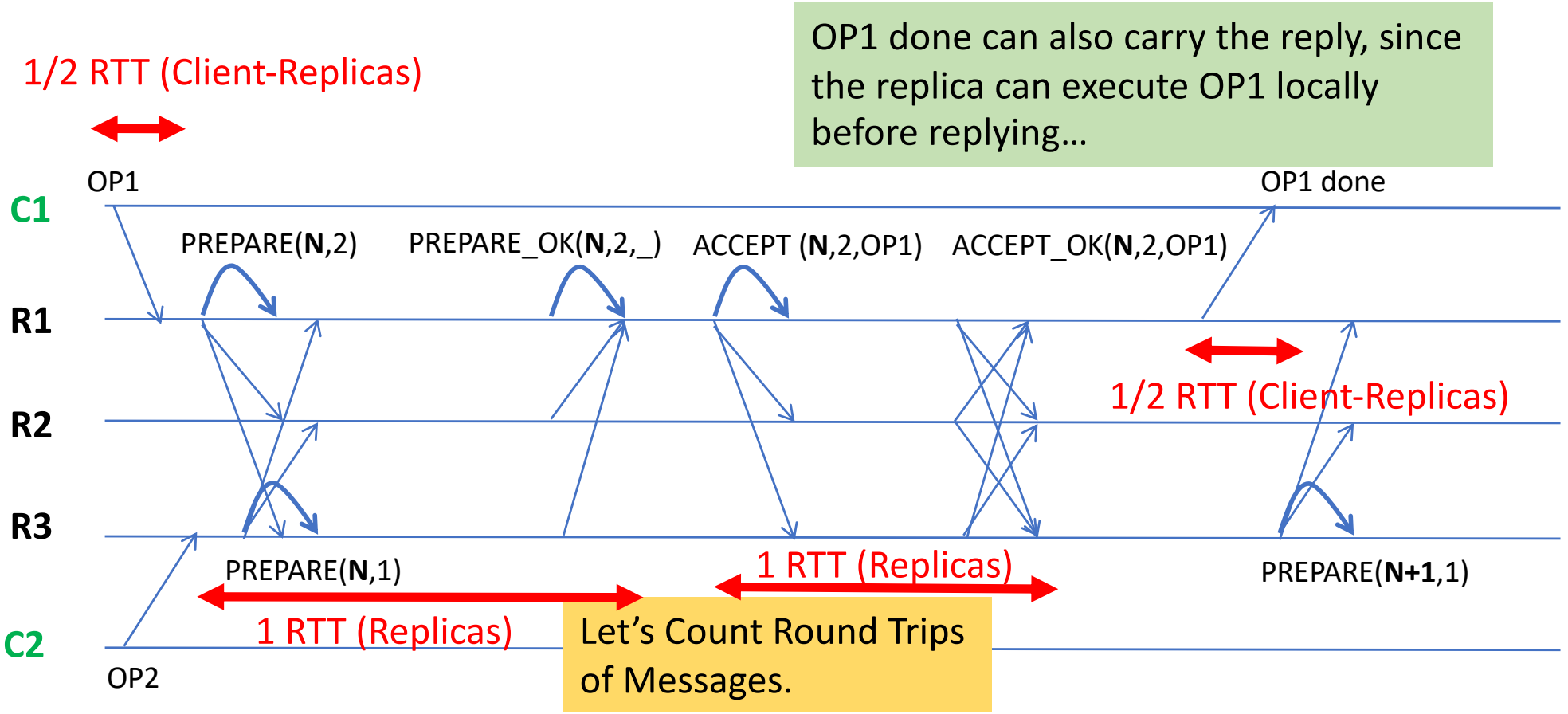
For Client 1, this is the good case, in which his operation is decided in the consensus instance following the reception of his request. Is this good?

Implementing State Machine Replication with Paxos



For Client 1, this is the good case, in which his operation is decided in the consensus instance following the reception of his request. Is this good?

Implementing State Machine Replication with Paxos



Even in the best case, Paxos requires two round trips between replicas to decide a value.

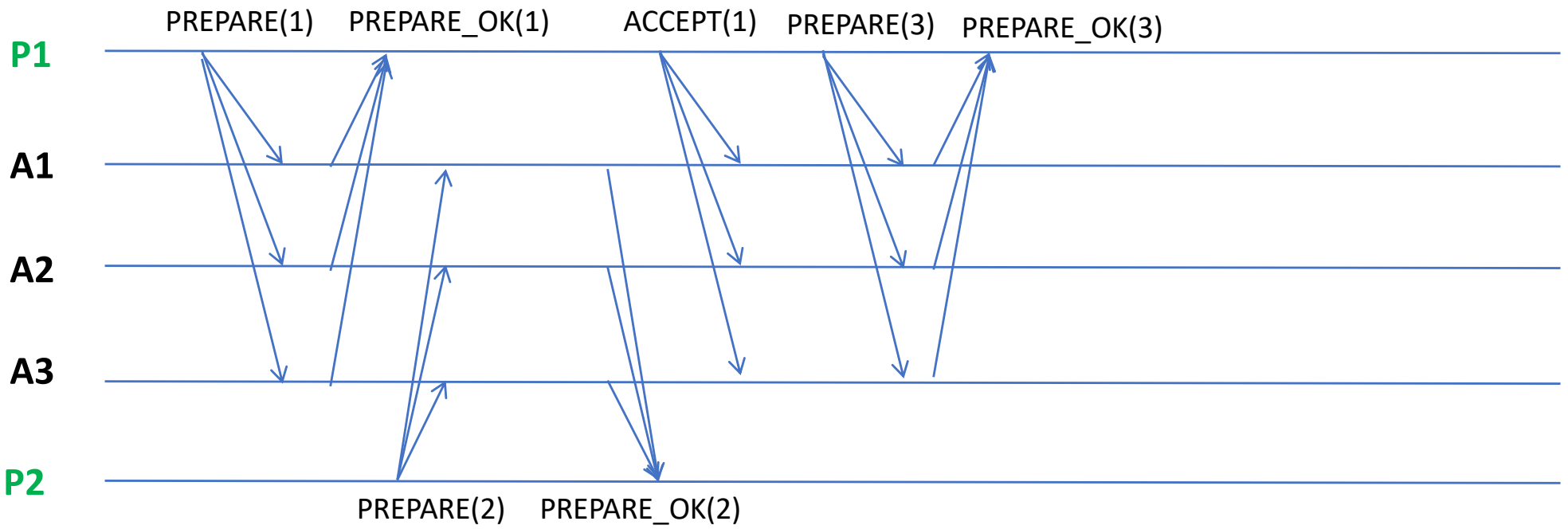
How can we improve this scenario?

Is this the only challenge associated with using Paxos?

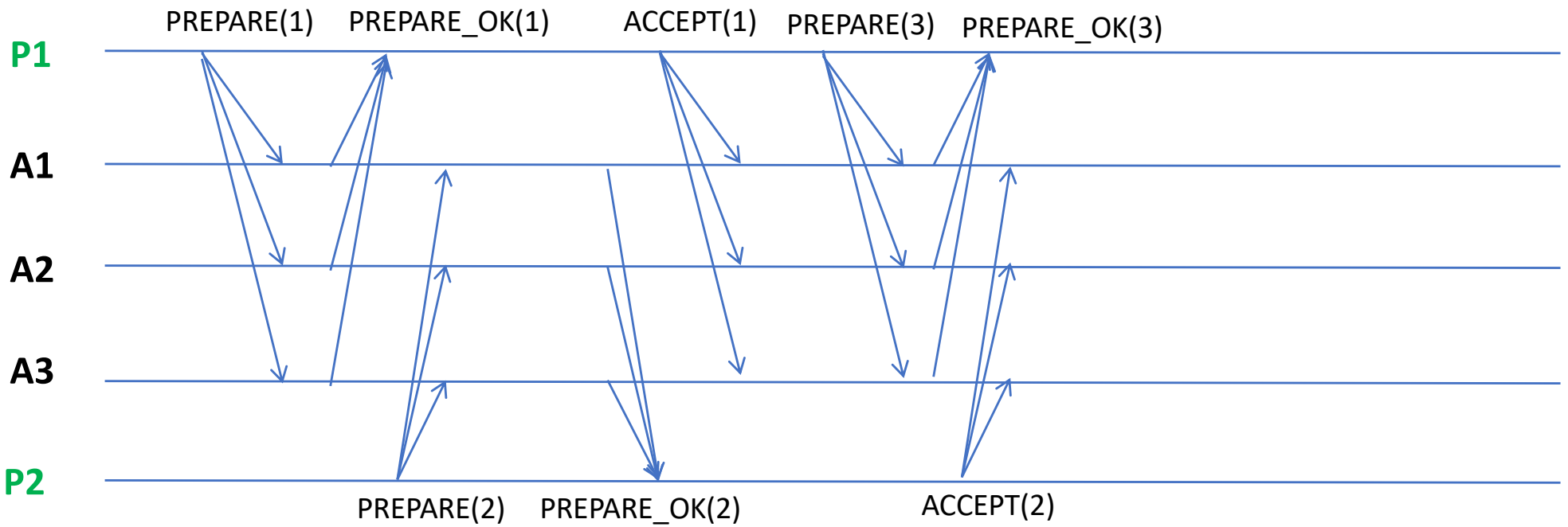
Is this the only challenge associated with using Paxos?

- No… remember the issue of liveness not being guaranteed?

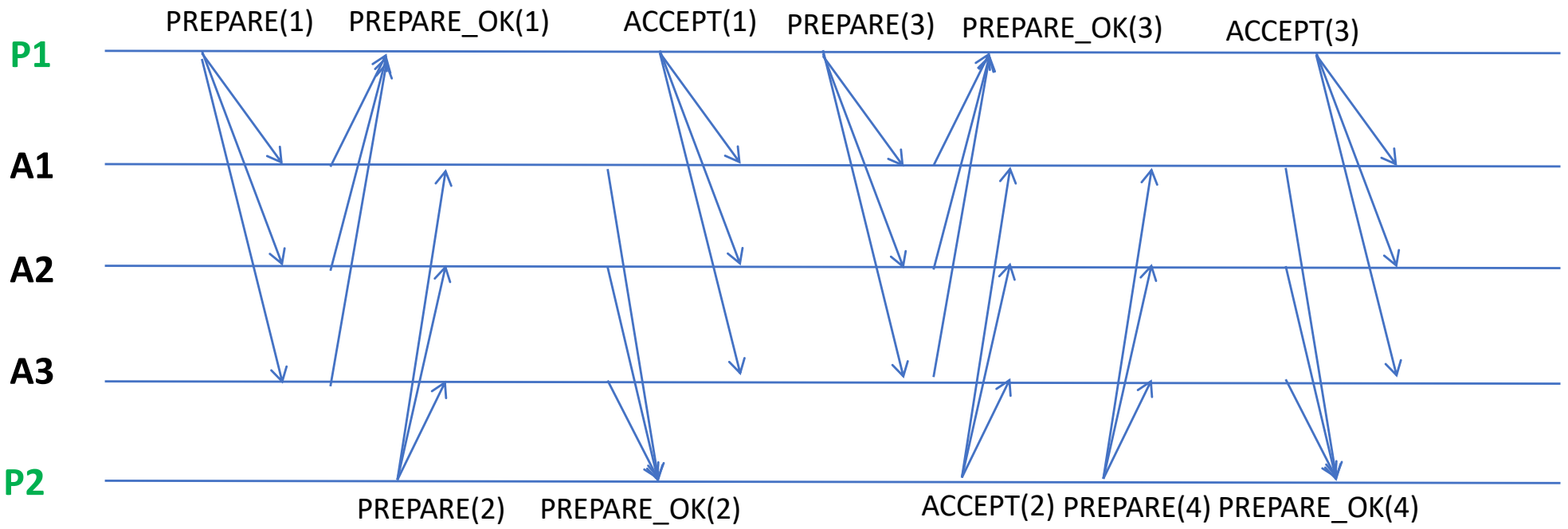
Liveness is not guaranteed (Termination Property)



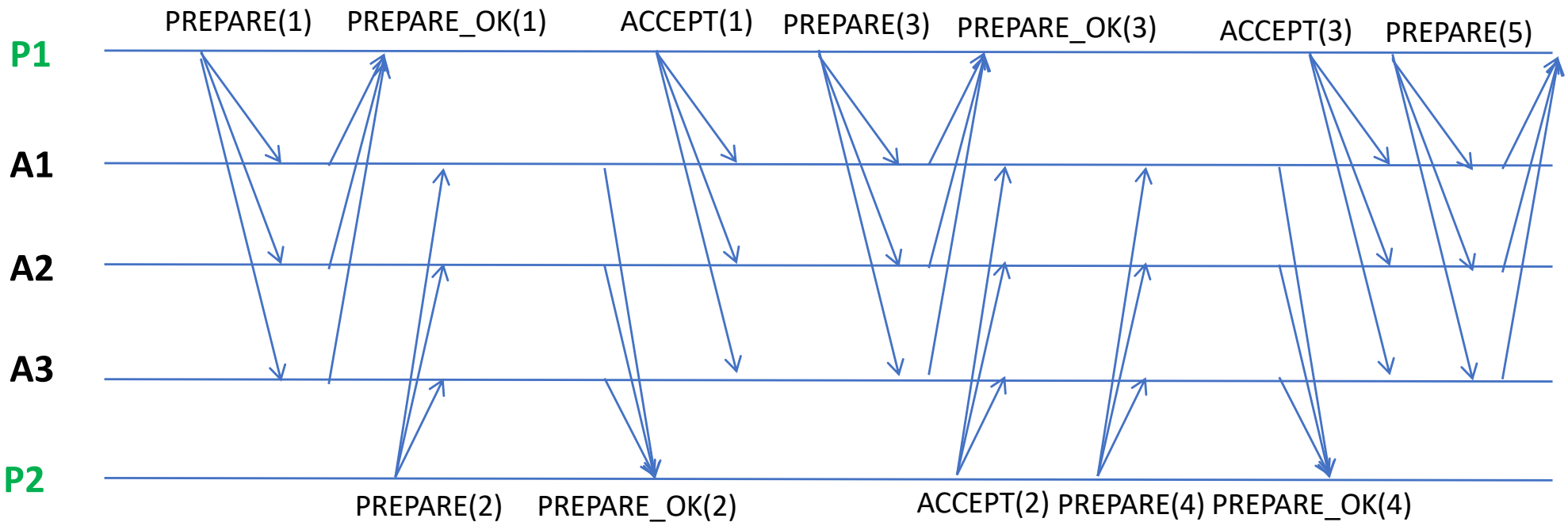
Liveness is not guaranteed (Termination Property)



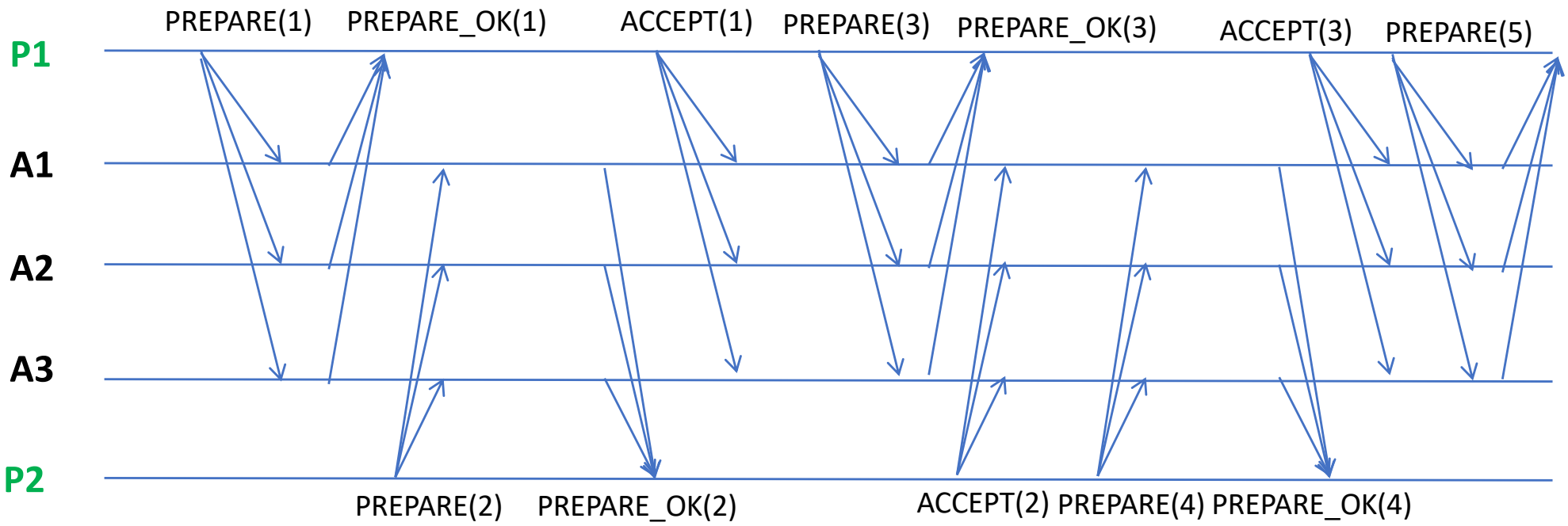
Liveness is not guaranteed (Termination Property)



Liveness is not guaranteed (Termination Property)

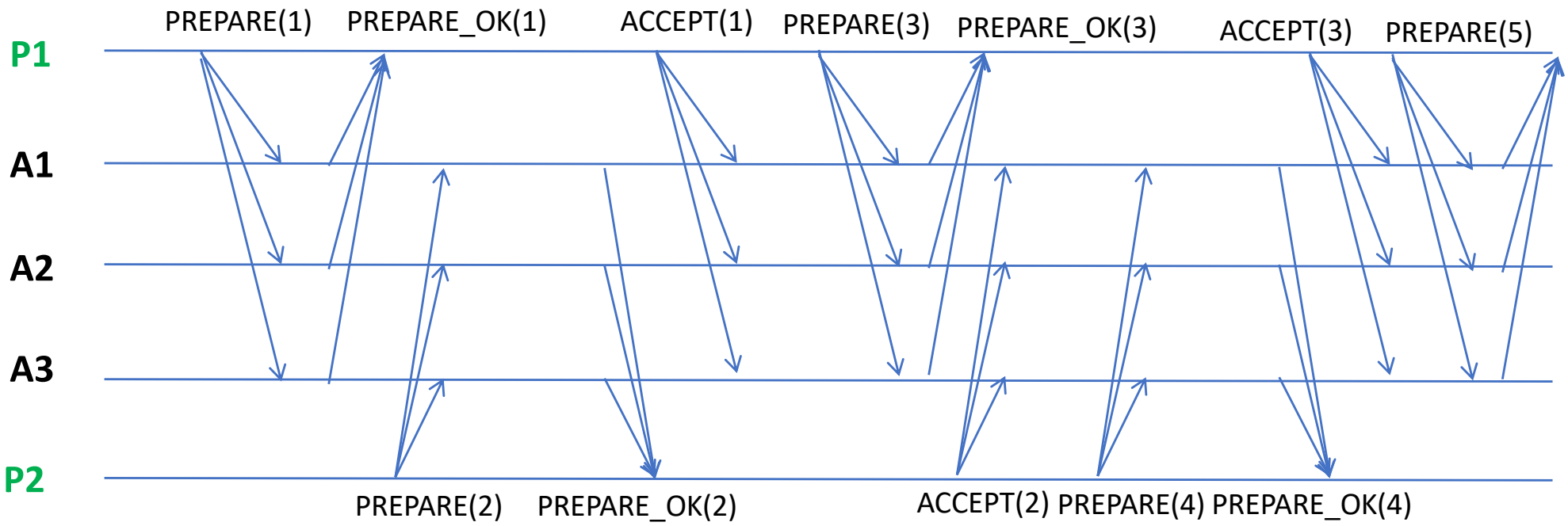


Liveness is not guaranteed (Termination Property)



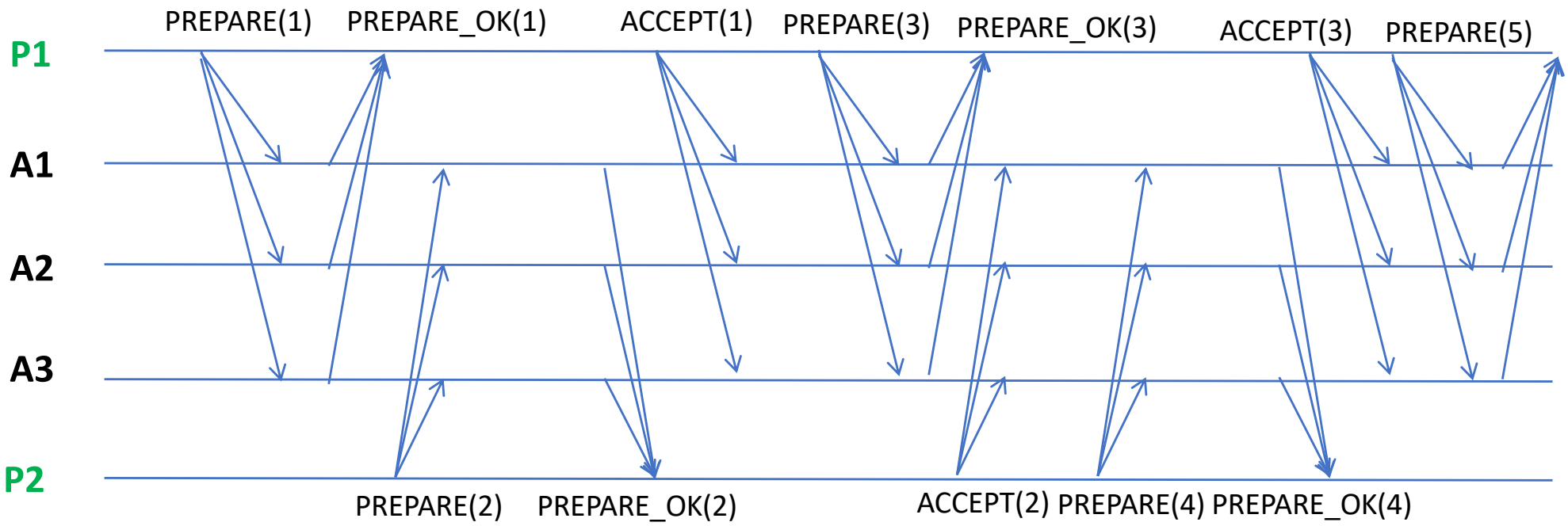
… For ever and ever

Liveness is not guaranteed (Termination Property)



… For ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever …

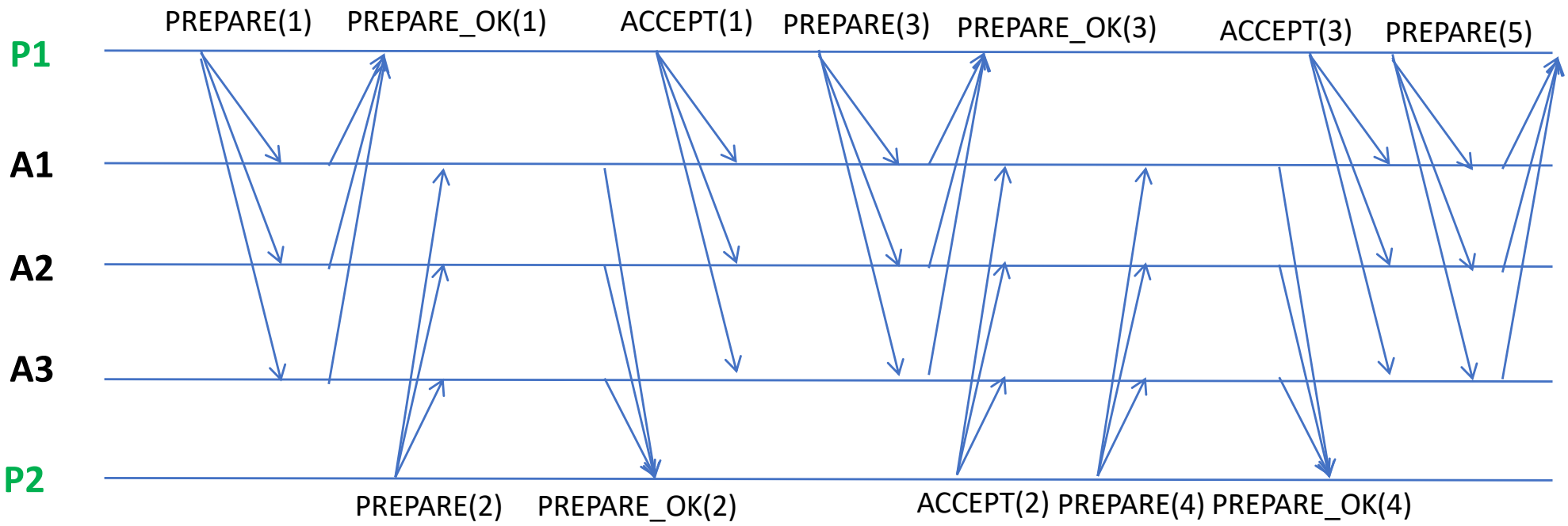
Liveness is not guaranteed (Termination Property)



… For ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever …

Can we completely solve this?

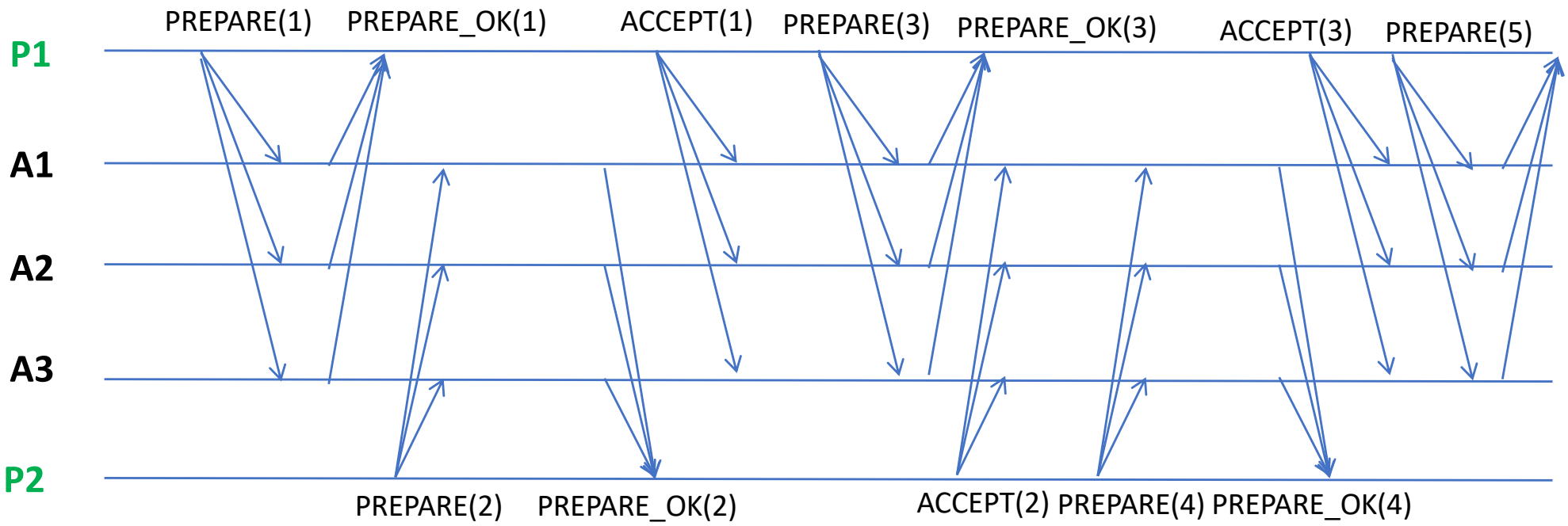
Liveness is not guaranteed (Termination Property)



… For ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever …

Can we completely solve this? Not completely (that would imply breaking FLP)

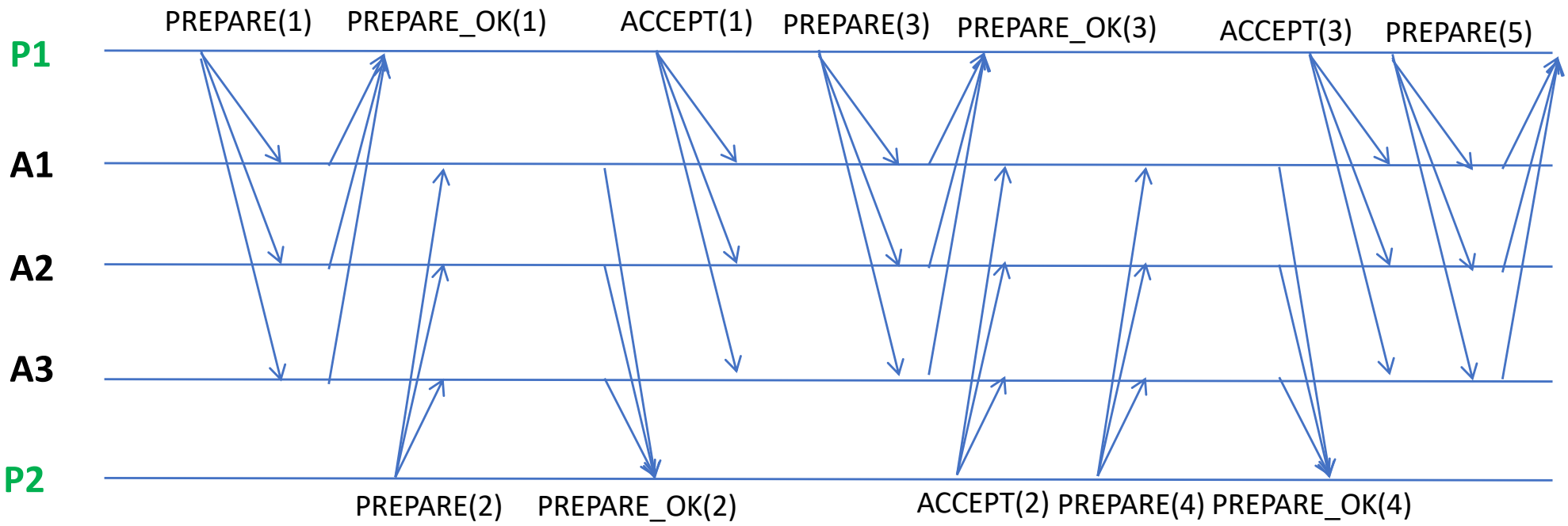
Liveness is not guaranteed (Termination Property)



… For ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever …

Can we completely solve this? Not completely (that would imply breaking FLP)
Is this really a problem?

Liveness is not guaranteed (Termination Property)



… For ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever and ever …

Can we completely solve this? Not completely (that would imply breaking FLP)
Is this really a problem? Might actually be!

Is there a condition where progress is easier to achieve?

- What was the problem in the previous execution?

Is there a condition where progress is easier to achieve?

- What was the problem in the previous execution?
  - There were two proposers trying to get their value decided simultaneously.
  - The activity of one of them makes it impossible for the other to achieve progress and vice versa.
  - **Liveness is potentially very hard to achieve with concurrent proposers.**
- How could this be addressed?

Is there a condition where progress is easier to achieve?

- What was the problem in the previous execution?
  - There were two proposers trying to get their value decided simultaneously.
  - The activity of one of them makes it impossible for the other to achieve progress and vice versa.
  - **Liveness is potentially very hard to achieve with concurrent proposers.**
- How could this be addressed?
  - Make sure that there is a single proposer…

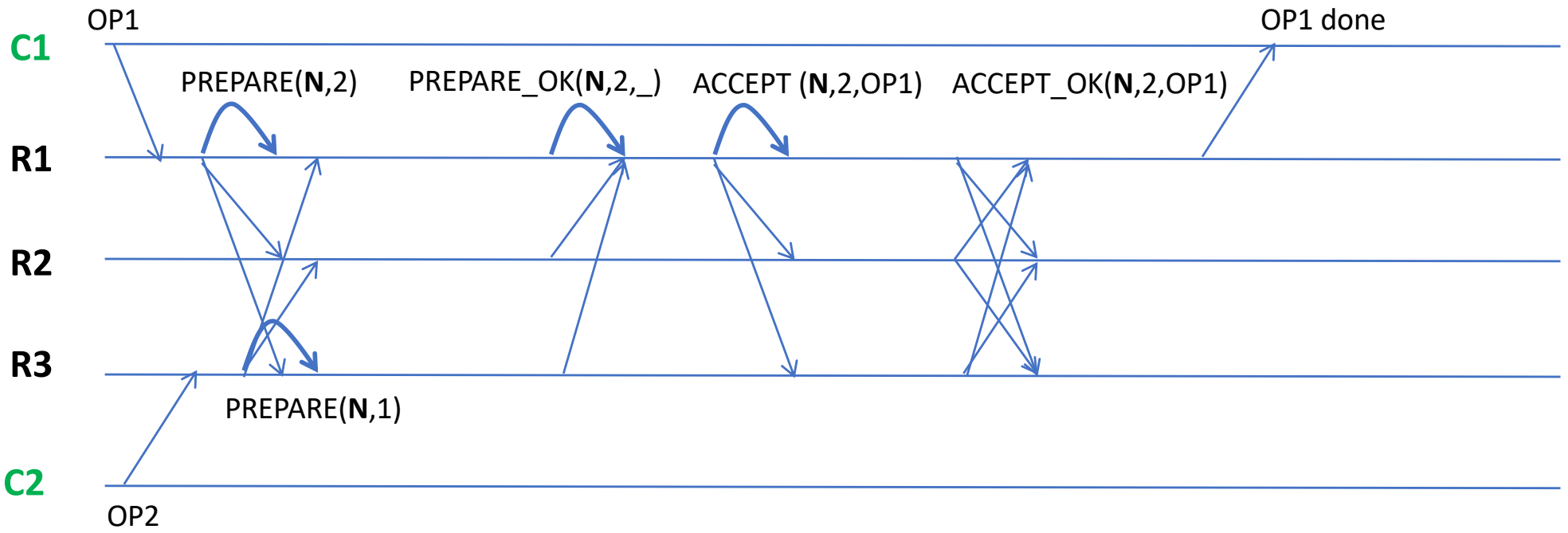
Are any other issues that we should consider?

Are any other issues that we should consider?

- What if replicas fail?

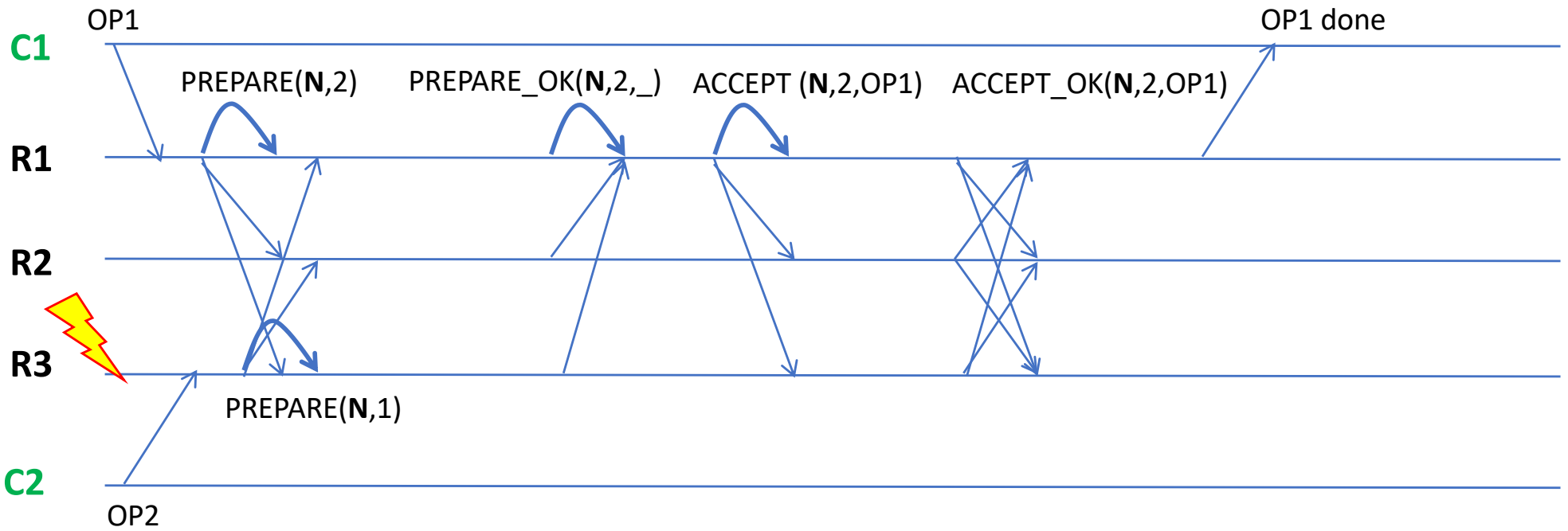
- What if the relevance of a service increases and you need to increase the fault tolerance of the service?

Membership issues...



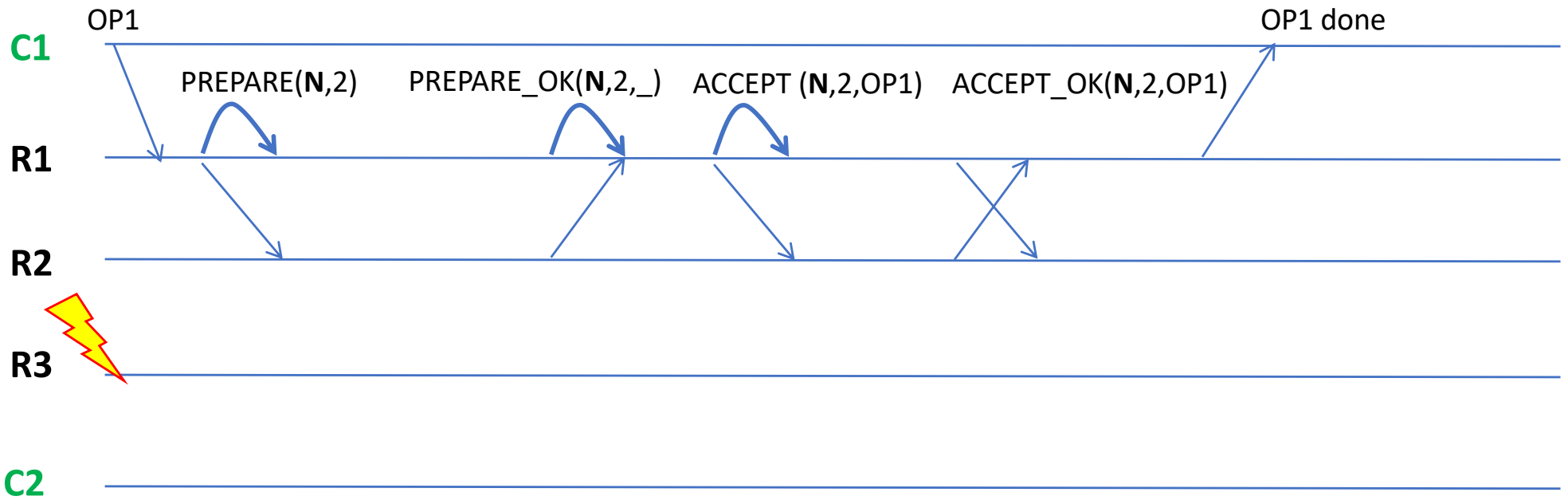
Look at the previous execution where OP1 from Client 1 is decided to be executed at operation slot N of the state machine.

Membership issues...



Look at the previous execution where OP1 from Client 1 is decided to be executed at operation slot N of the state machine.

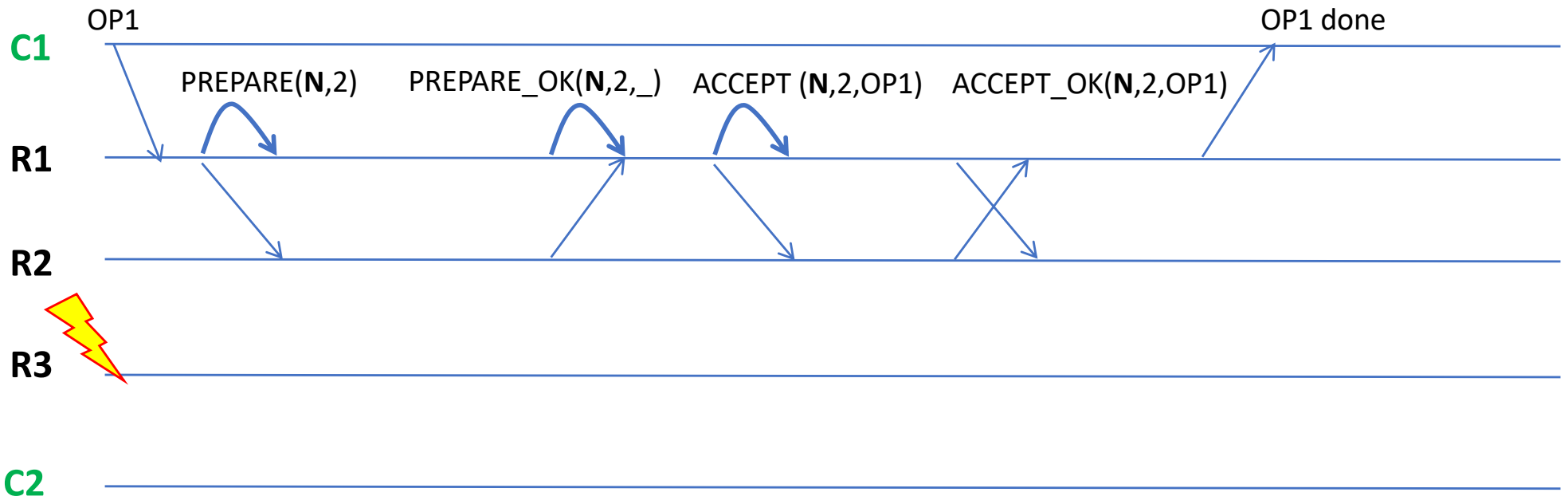
Membership issues...



Look at the previous execution where OP1 from Client 1 is decided to be executed at operation slot N of the state machine.

Membership issues...

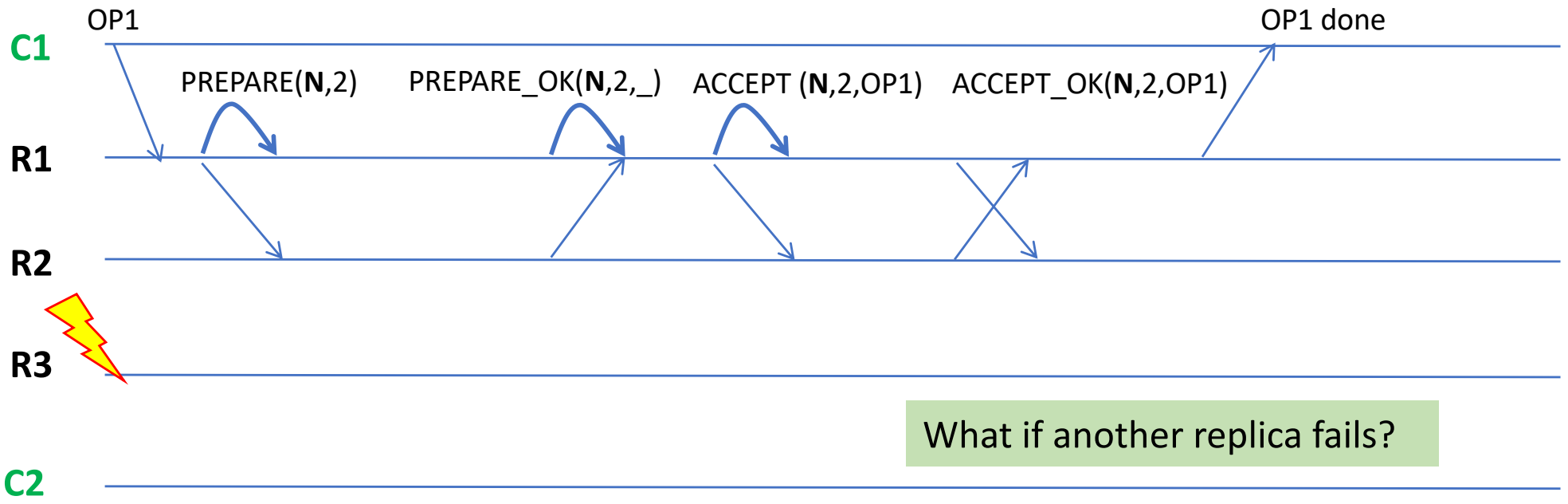
Still fine… we lose a replica but we still make progress (some client operations might be lost but they can always try again by issuing their operation to another replica...



Look at the previous execution where OP1 from Client 1 is decided to be executed at operation slot N of the state machine.

Membership issues...

Still fine… we lose a replica but we still make progress (some client operations might be lost but they can always try again by issuing their operation to another replica...

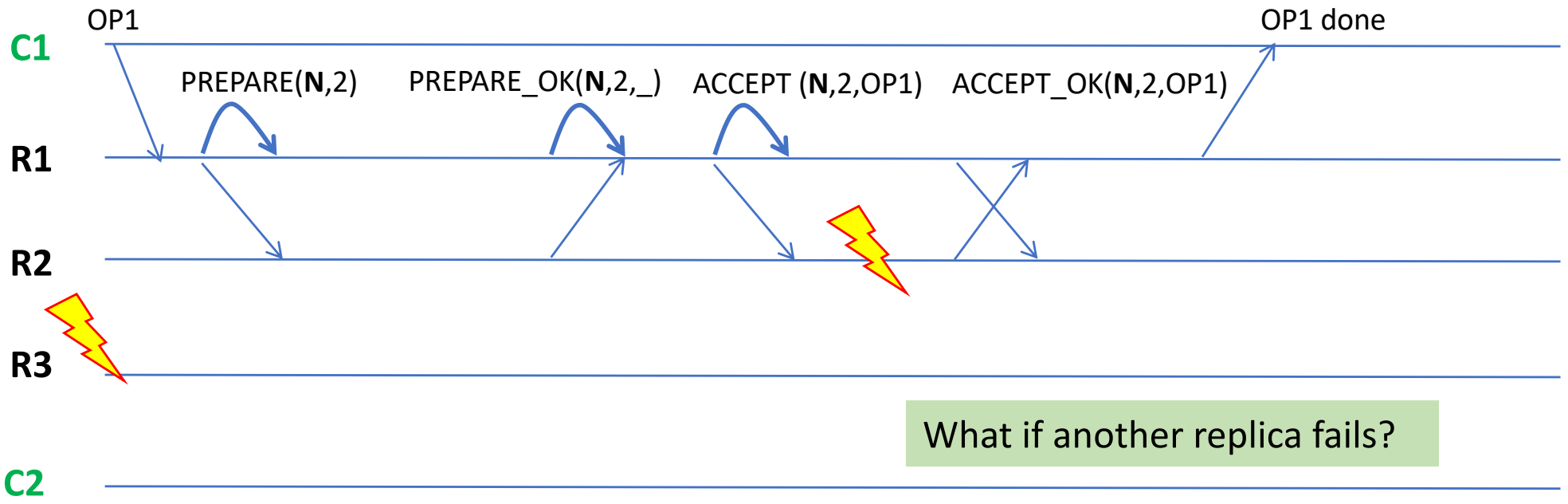


What if another replica fails?

Look at the previous execution where OP1 from Client 1 is decided to be executed at operation slot N of the state machine.

Membership issues...

Still fine… we lose a replica but we still make progress (some client operations might be lost but they can always try again by issuing their operation to another replica...

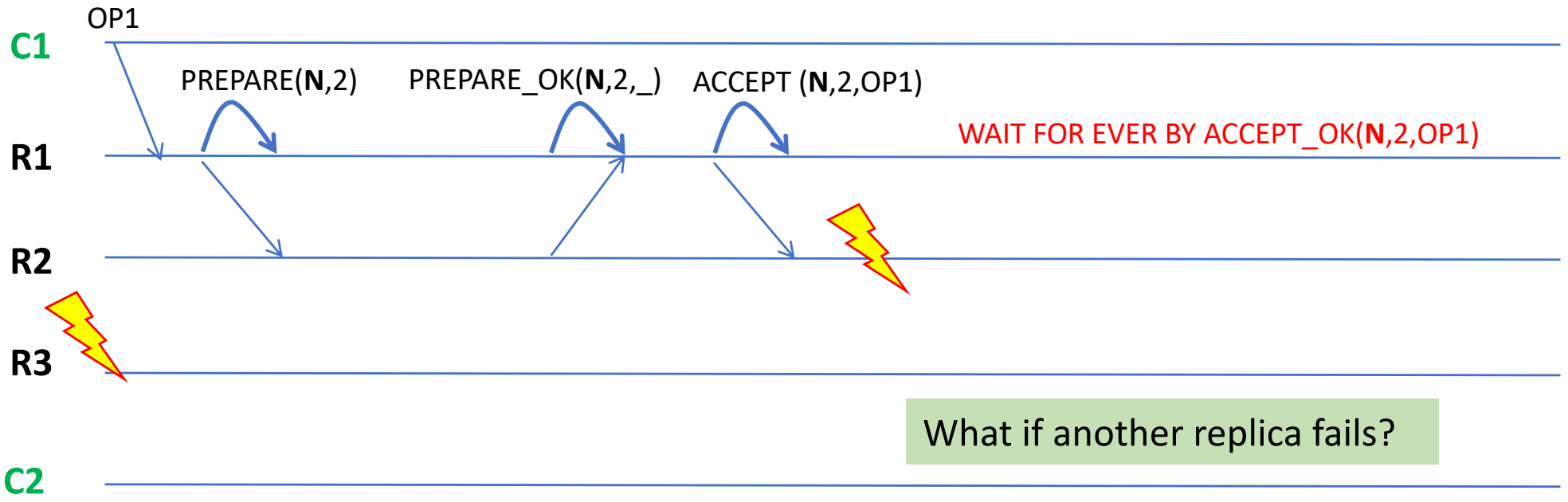


What if another replica fails?

Look at the previous execution where OP1 from Client 1 is decided to be executed at operation slot N of the state machine.

Membership issues...

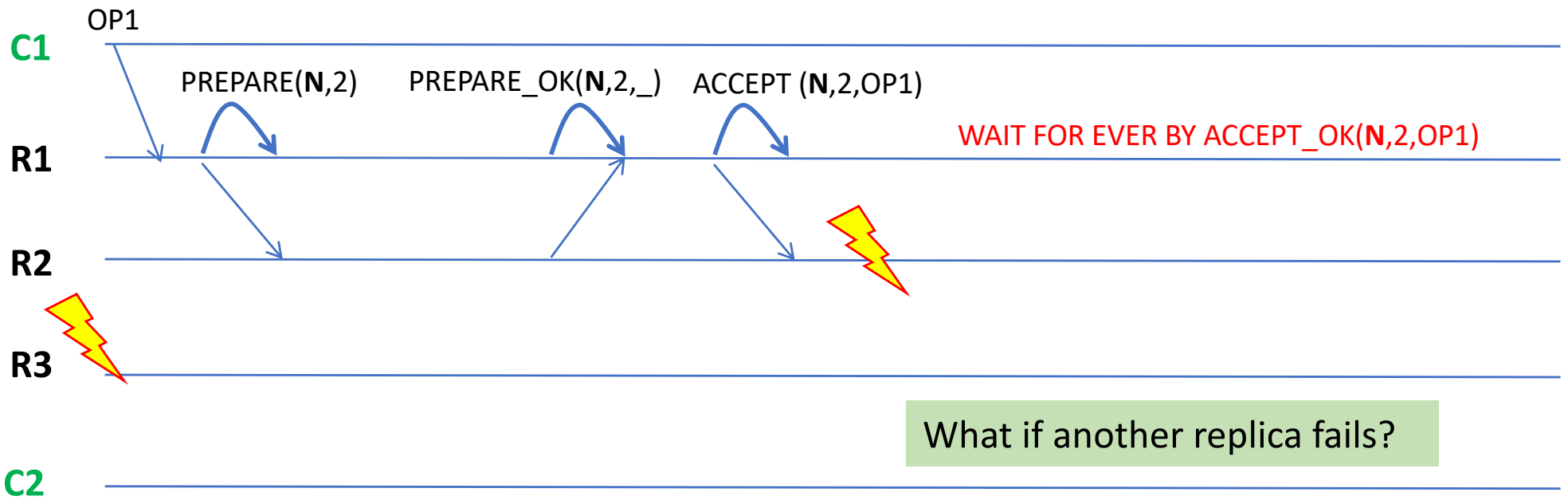
Still fine… we lose a replica but we still make progress (some client operations might be lost but they can always try again by issuing their operation to another replica...



Look at the previous execution where OP1 from Client 1 is decided to be executed at operation slot N of the state machine.

Membership issues...

Since we operate with Majority Quoruns we must have a majority of correct processes... In these conditions we cannot make progress and hence compromise liveness permanently...



Look at the previous execution where OP1 from Client 1 is decided to be executed at operation slot N of the state machine.

Paxos membership issues…

- When failures happen and the replica is not recoverable, we need to be able to replace it by another replica…
- Similarly, thinking about a long running system, at some point we might need to decomission a machine (because it is outdated...) this should not be seen as an unplanned failure of a replica.

Paxos membership issues…

- When failures happen and the replica is not recoverable, we need to be able to replace it by another replica…
- Similarly, thinking about a long running system, at some point we might need to decomission a machine (because it is outdated…) this should not be seen as an unplanned failure of a replica.

- **We need mechanisms to manipulate the membership of the system (set π)**

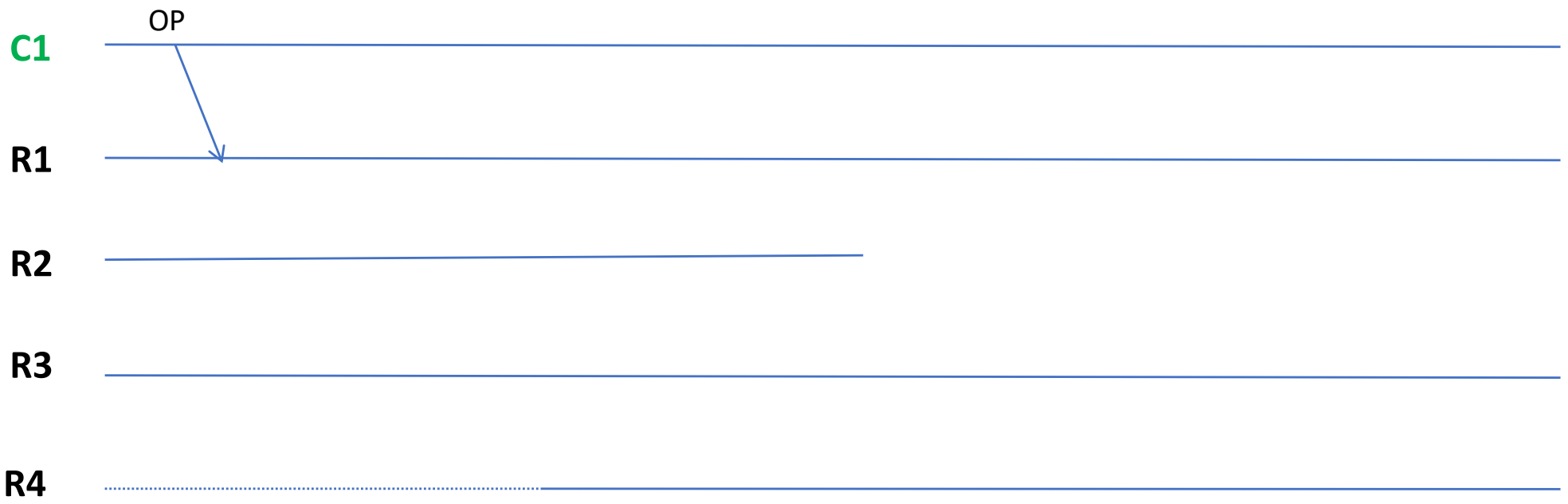
Problems

- Need mechanism to control replica membership (i.e., add / remove replicas)
- Paxos does not guarantee liveness in the presence of concurrent proposals
- Paxos requires two rounds of messages even if a single proposal exists

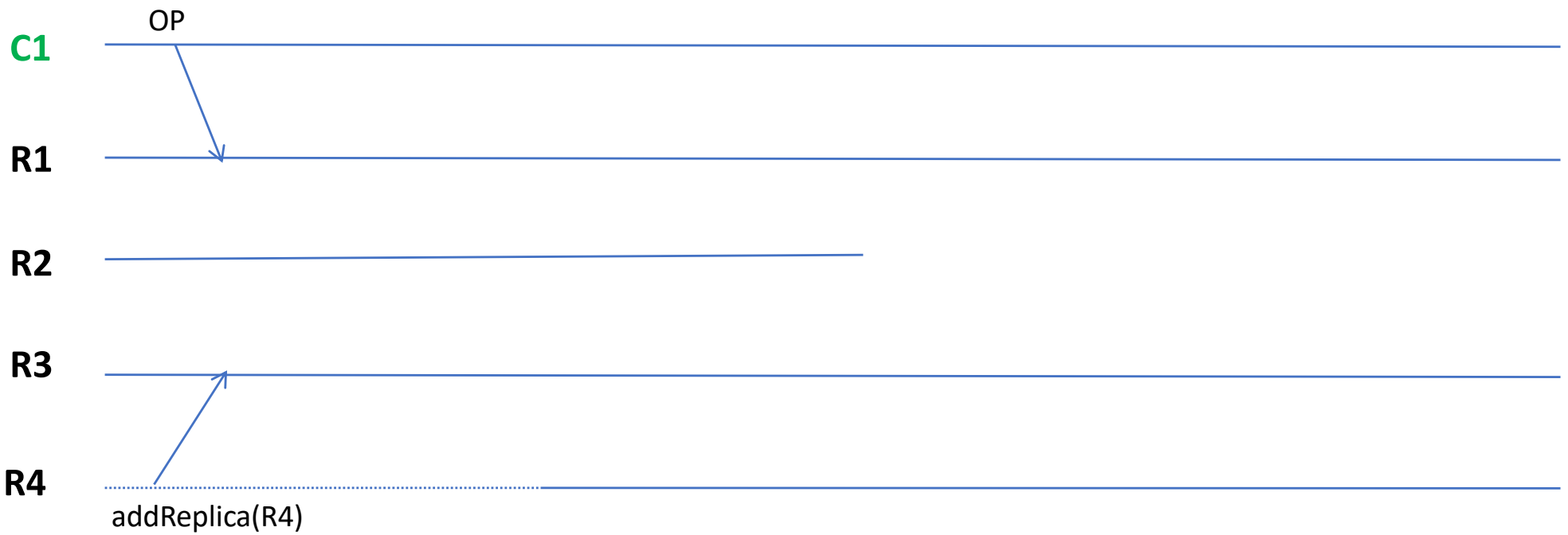
Problems

- Need mechanism to control replica membership (i.e., add / remove replicas)
- Paxos does not guarantee liveness in the presence of concurrent proposals
- Paxos requires two rounds of messages even if a single proposal exists

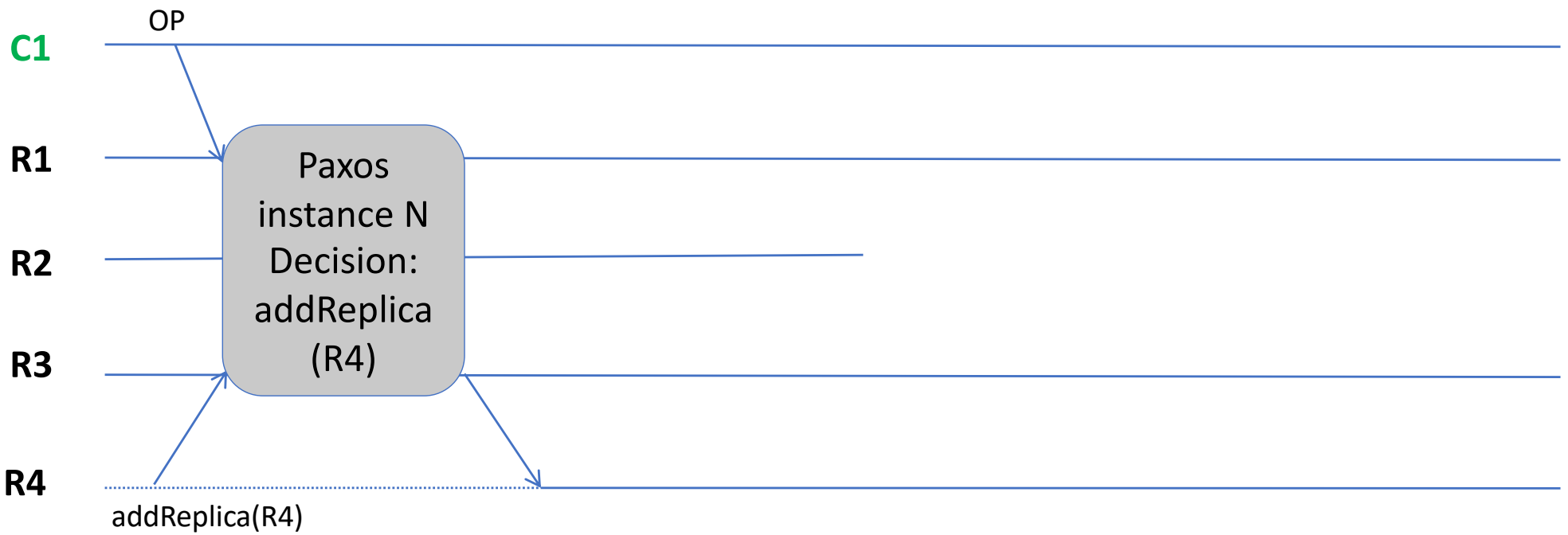
Add/remove replica



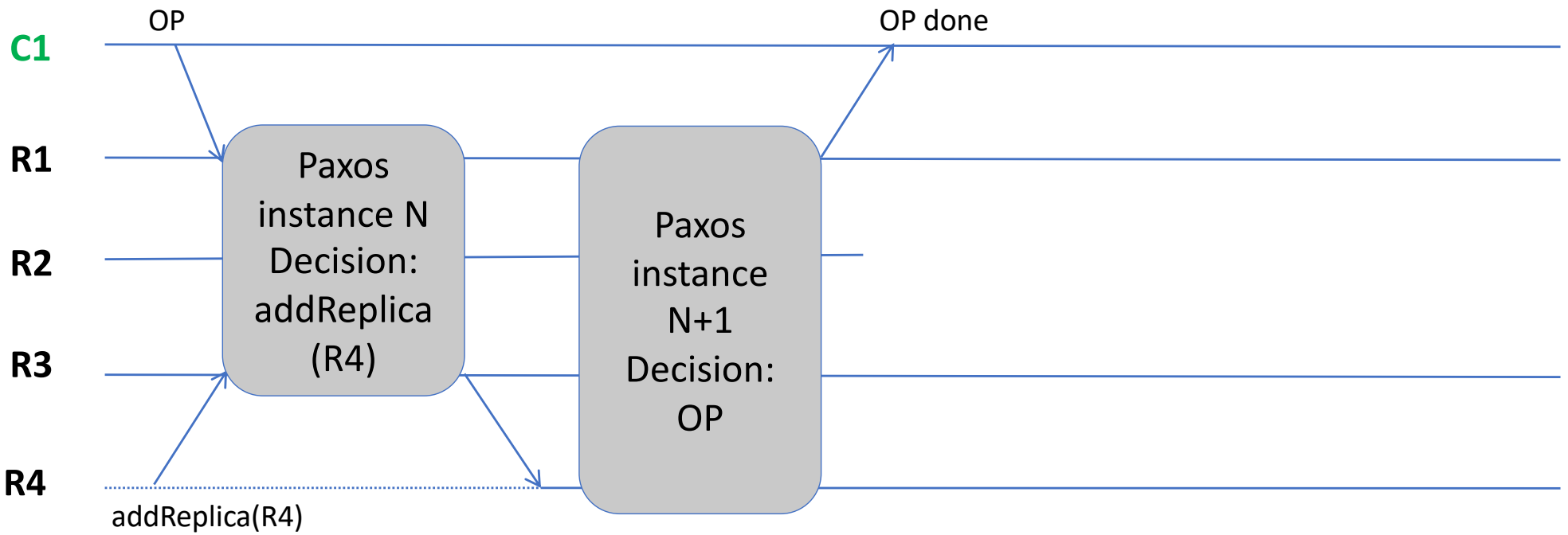
Add/remove replica



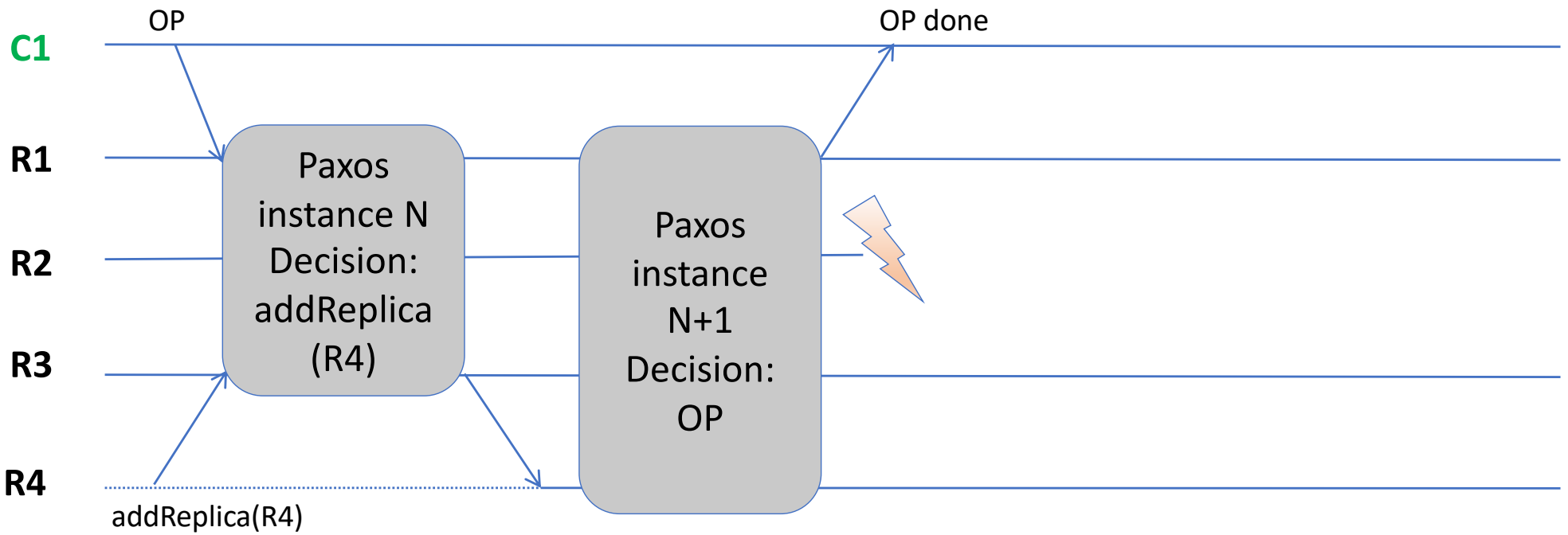
Add/remove replica



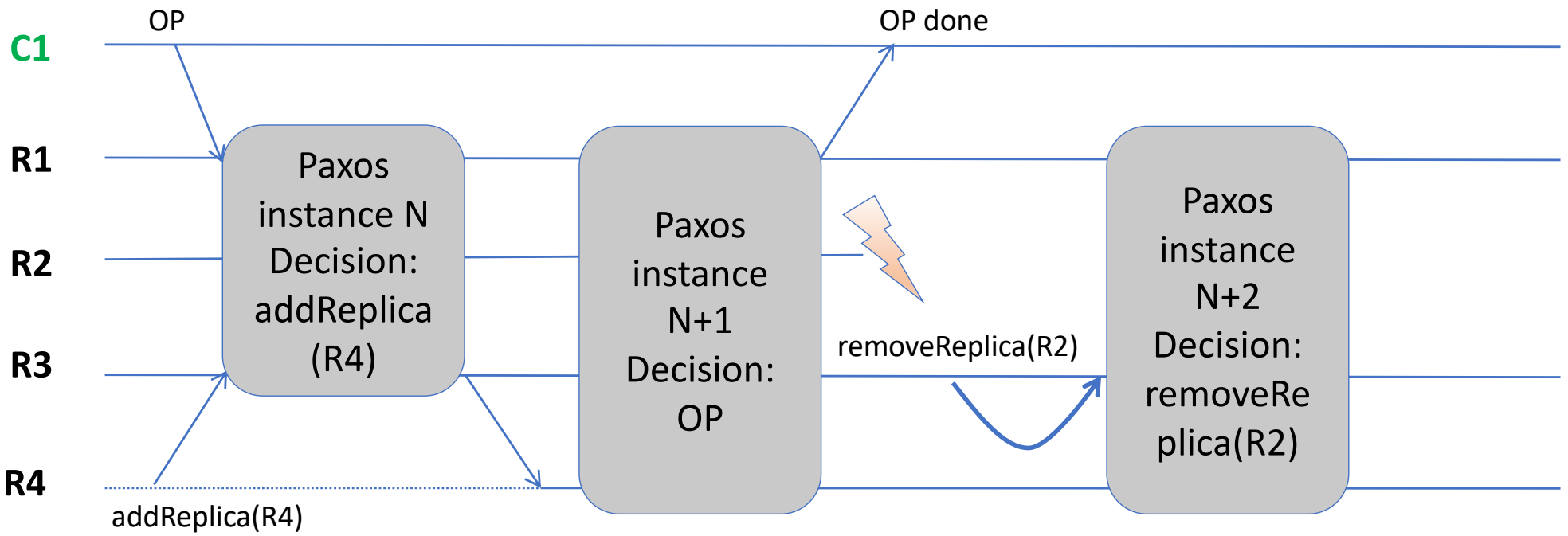
Add/remove replica



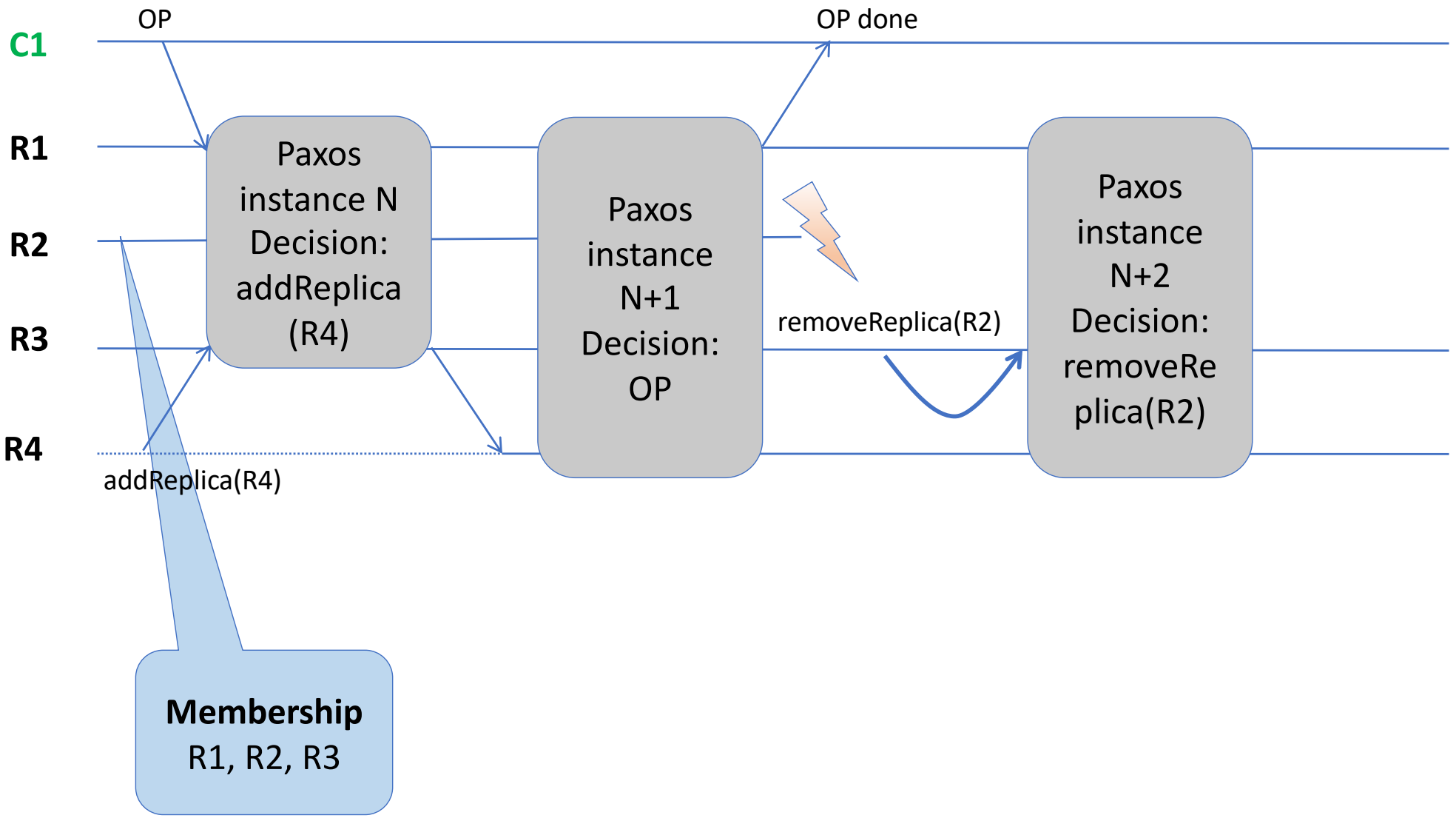
Add/remove replica



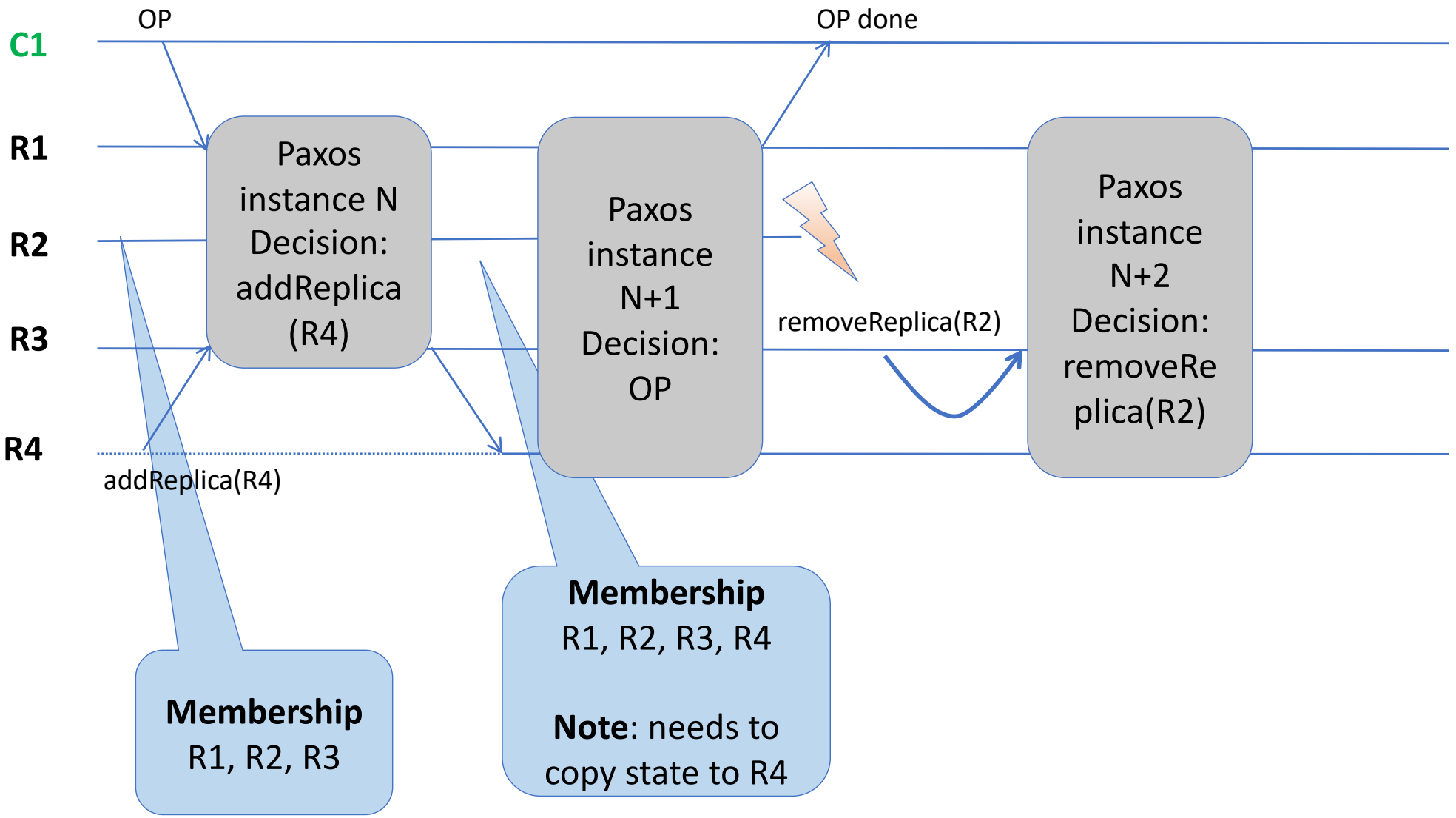
Add/remove replica



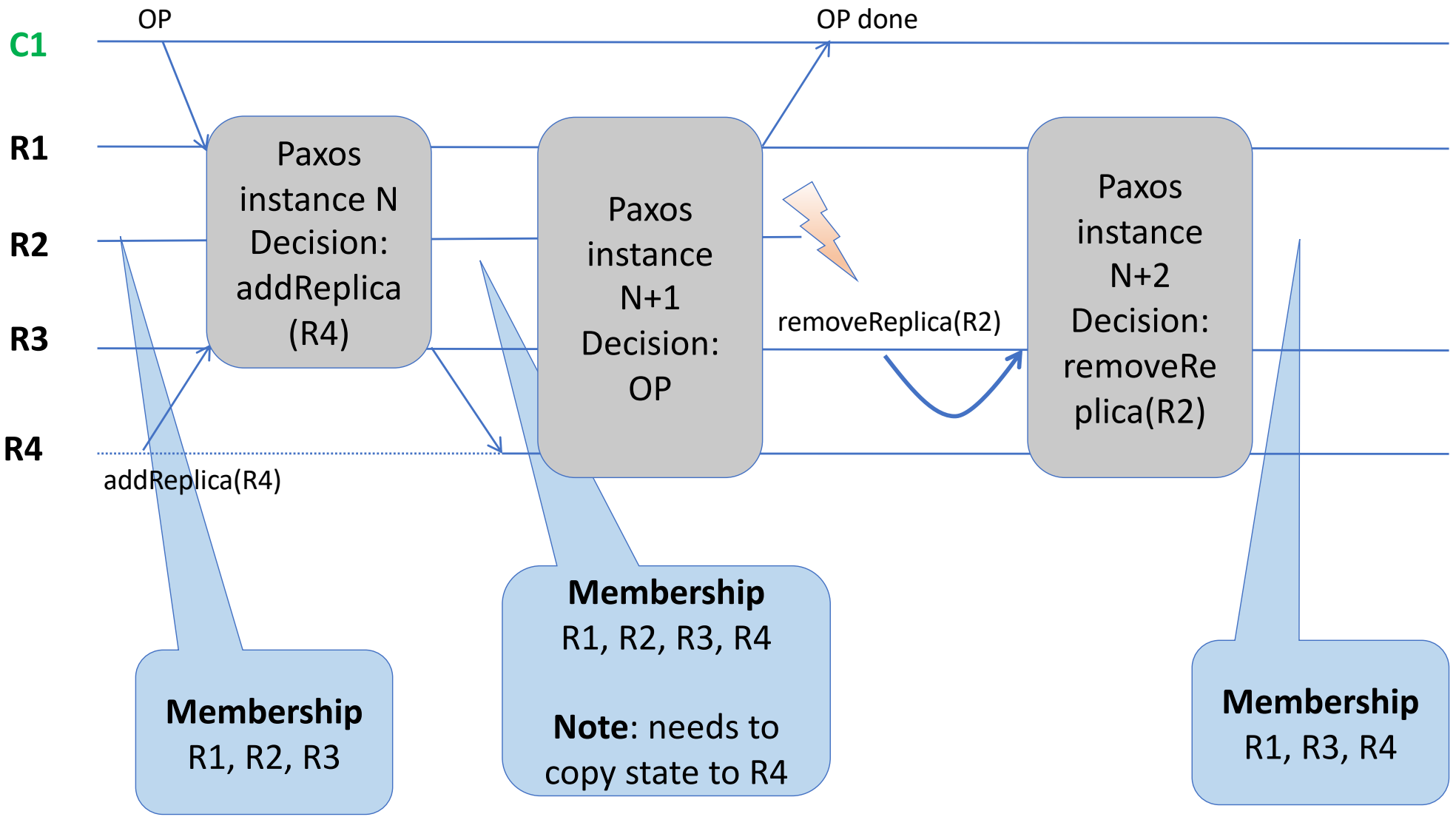
Add/remove replica



Add/remove replica



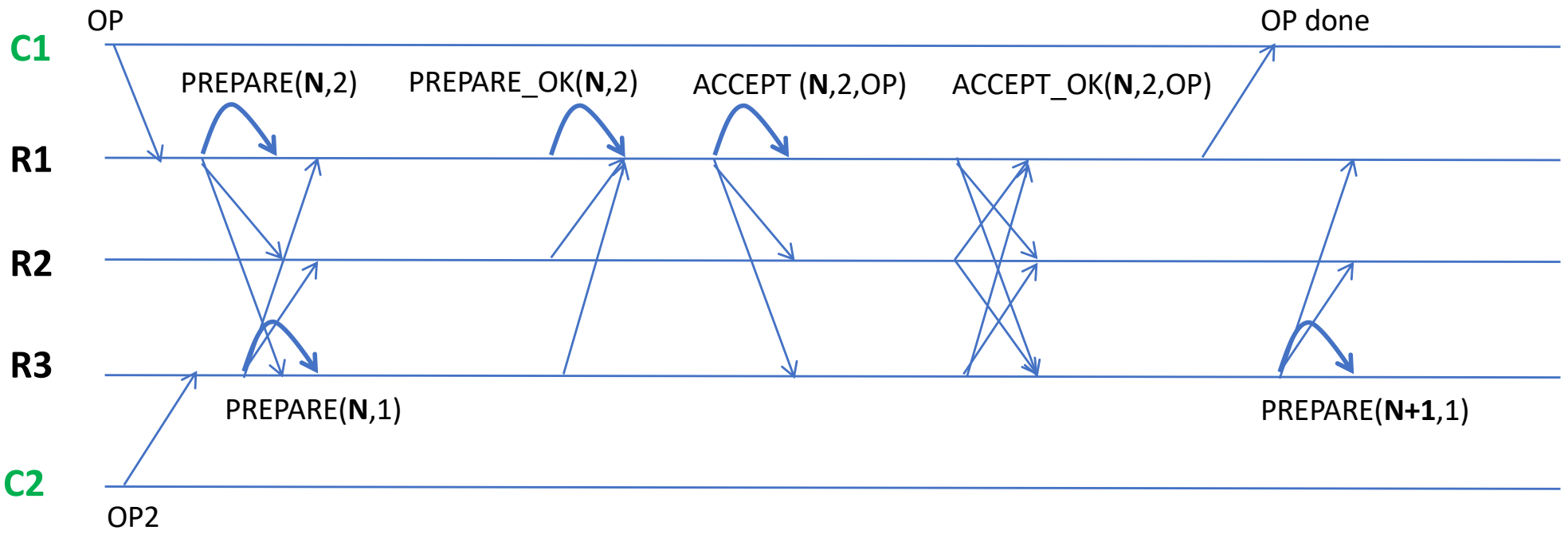
Add/remove replica



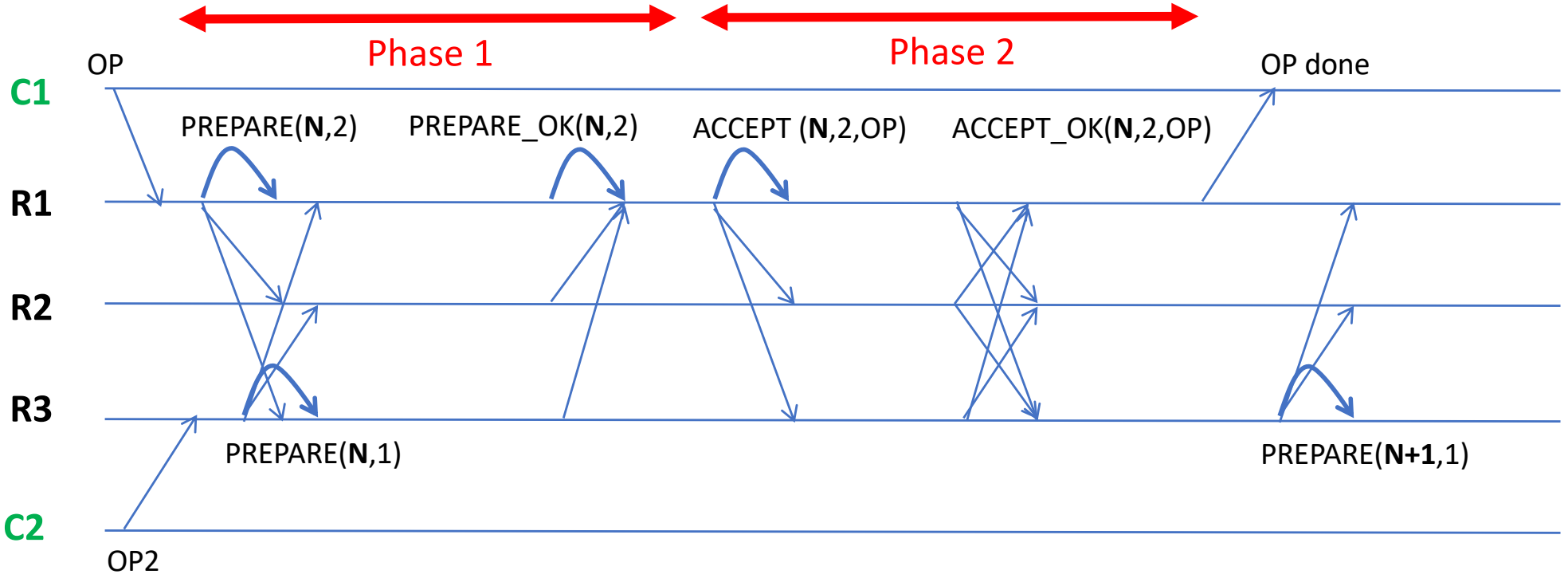
Problems

- Need mechanism to control replica membership (i.e., add / remove replicas)
- Paxos does not guarantee liveness in the presence of concurrent proposals
- Paxos requires two rounds of messages even if a single proposal exists

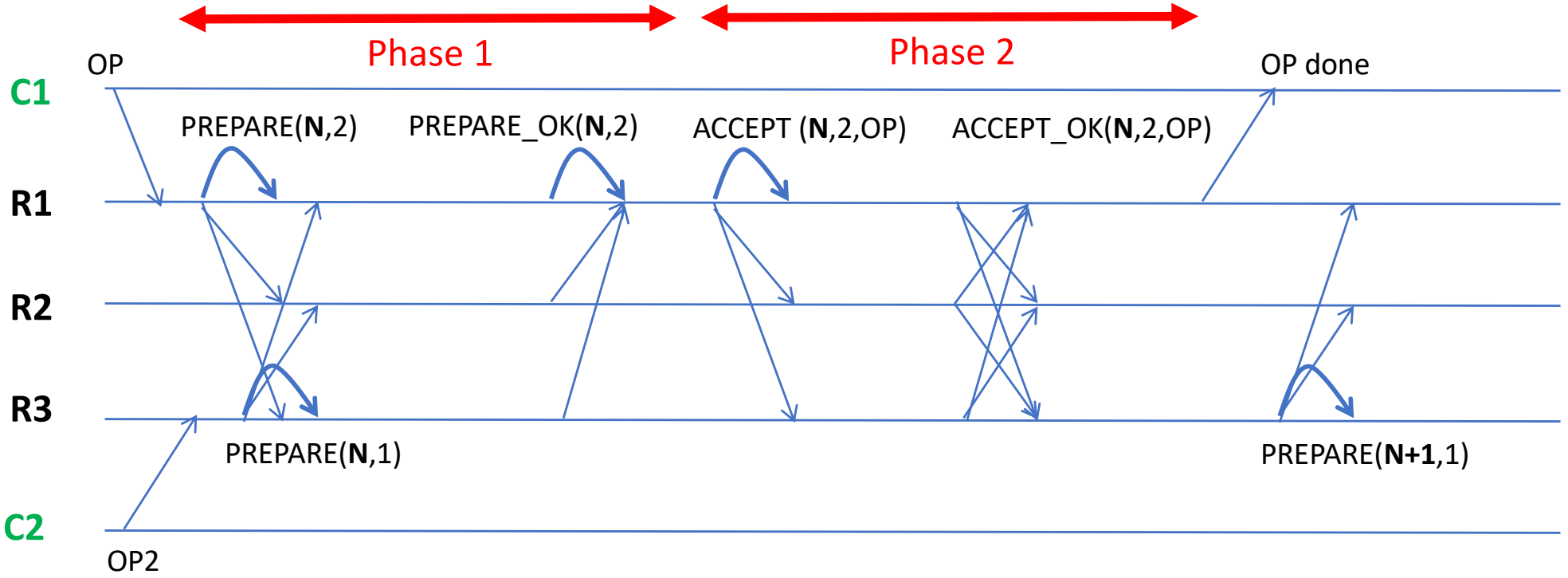
Optimizing execution



Optimizing execution

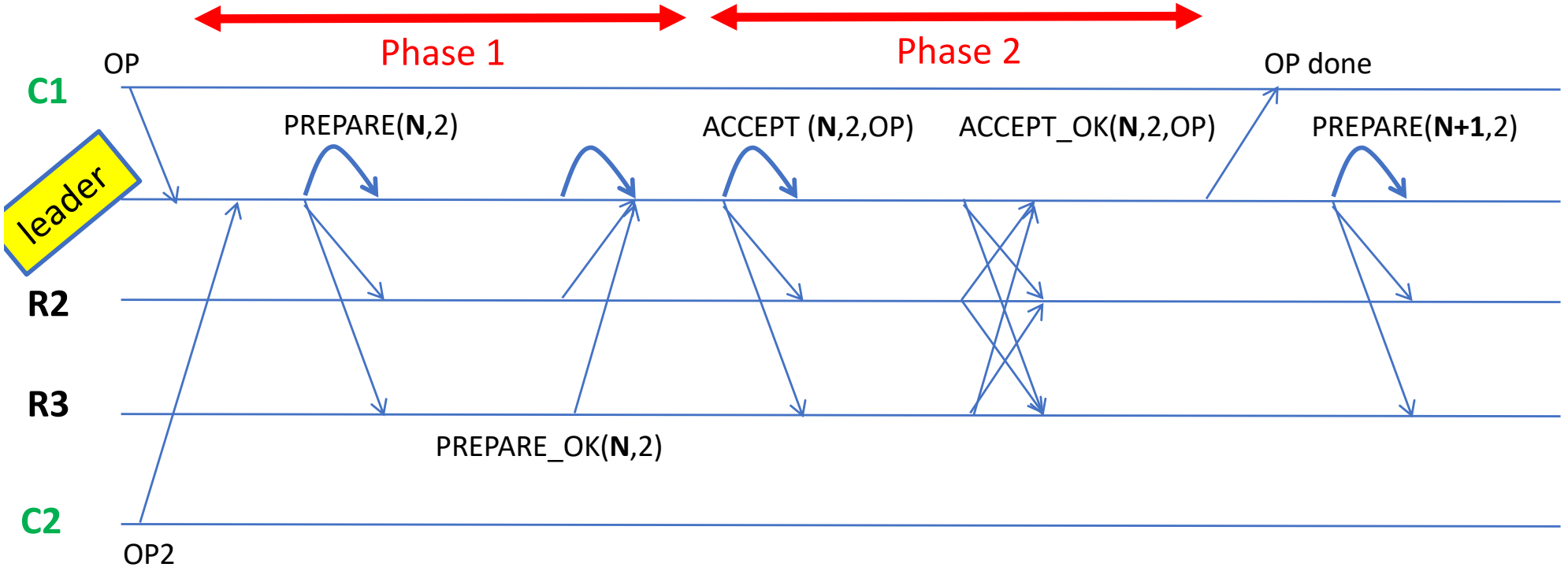


Optimizing execution



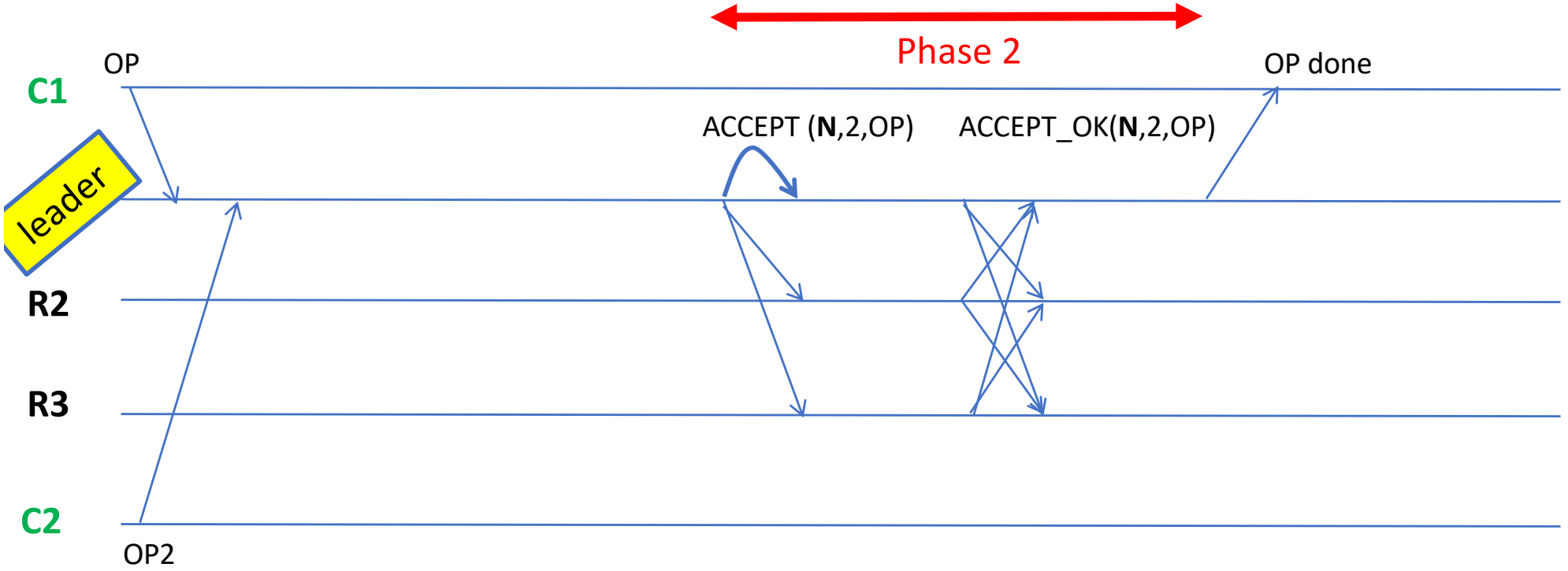
Why do we need two phases?

Optimizing execution



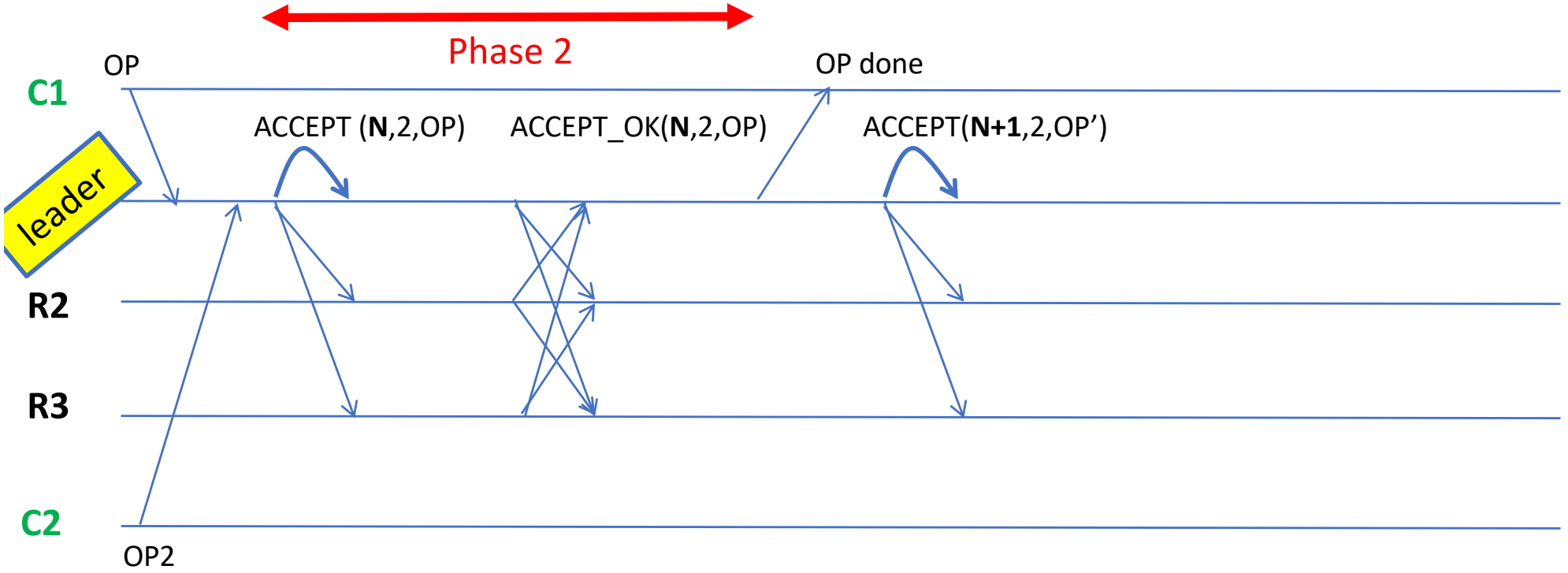
Do we need two phases if a single replica makes proposals?

Optimizing execution



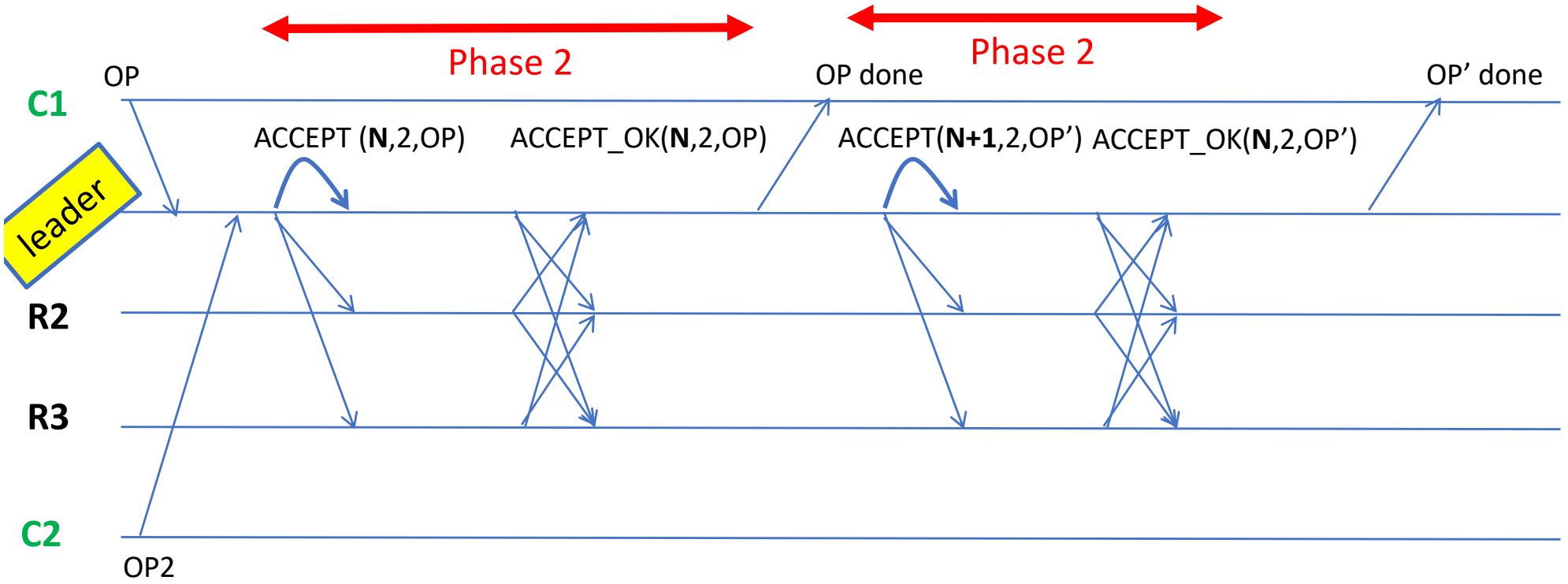
If there is a single proposer phase 1 is not required…

Optimizing execution



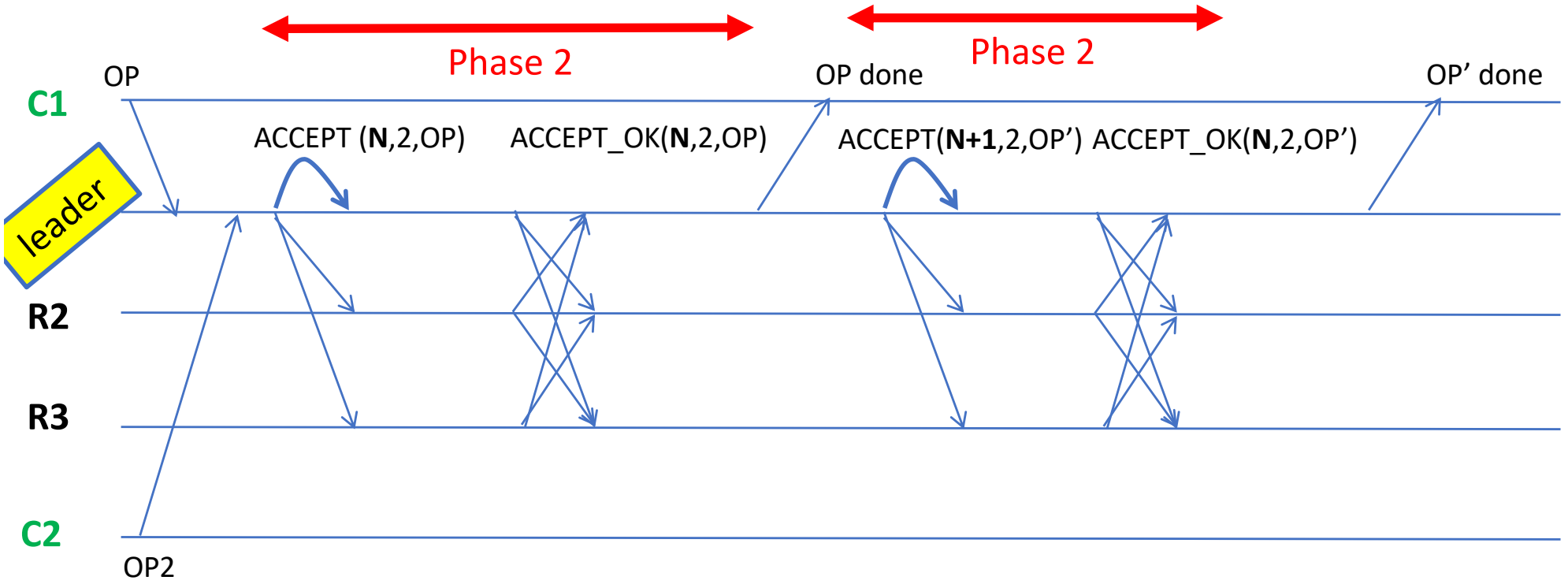
If there is a single proposer phase 1 is not required…

Optimizing execution



If there is a single proposer phase 1 is not required…

Optimizing execution



The existence of a leader effectively reduces the cost of a consensus instance from 2 RTT to 1 RTT

Paxos with a Leader

- Avoids the necessity of executing the prepare phase (more efficient in terms of communication steps).

Paxos with a Leader

- Avoids the necessity of executing the prepare phase (more efficient in terms of communication steps).

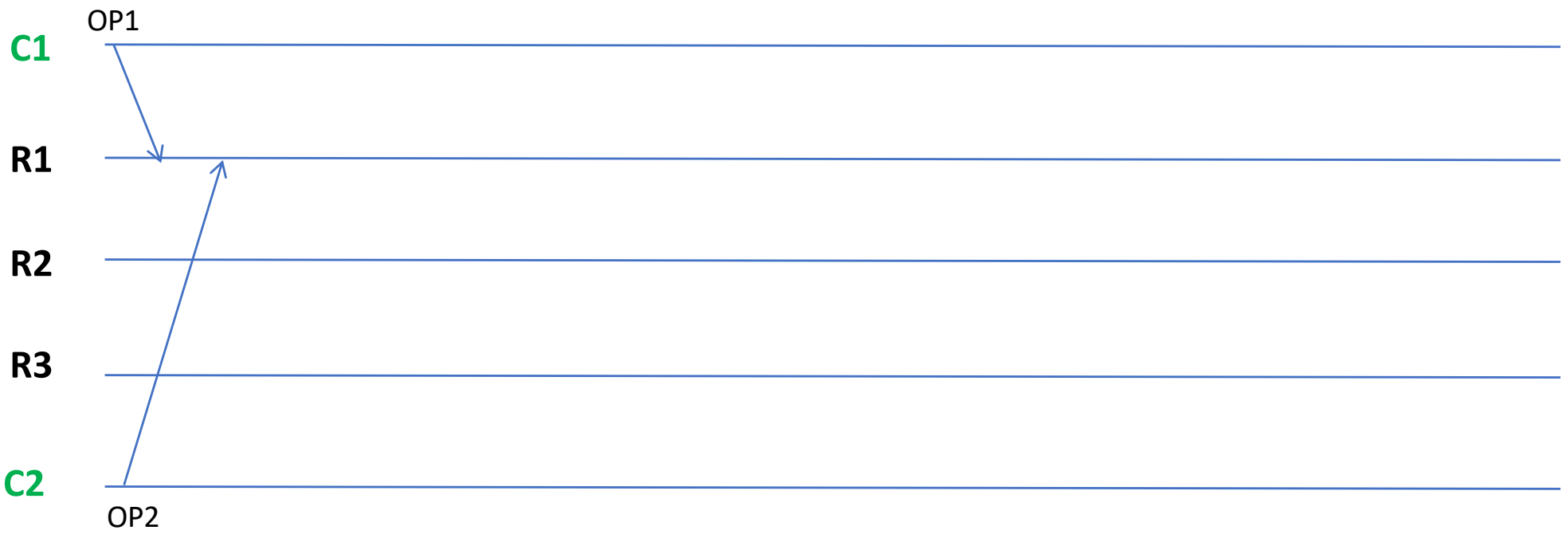
- However:
  - What if the leader fails?
  - How do we select a leader?

Paxos with a Leader

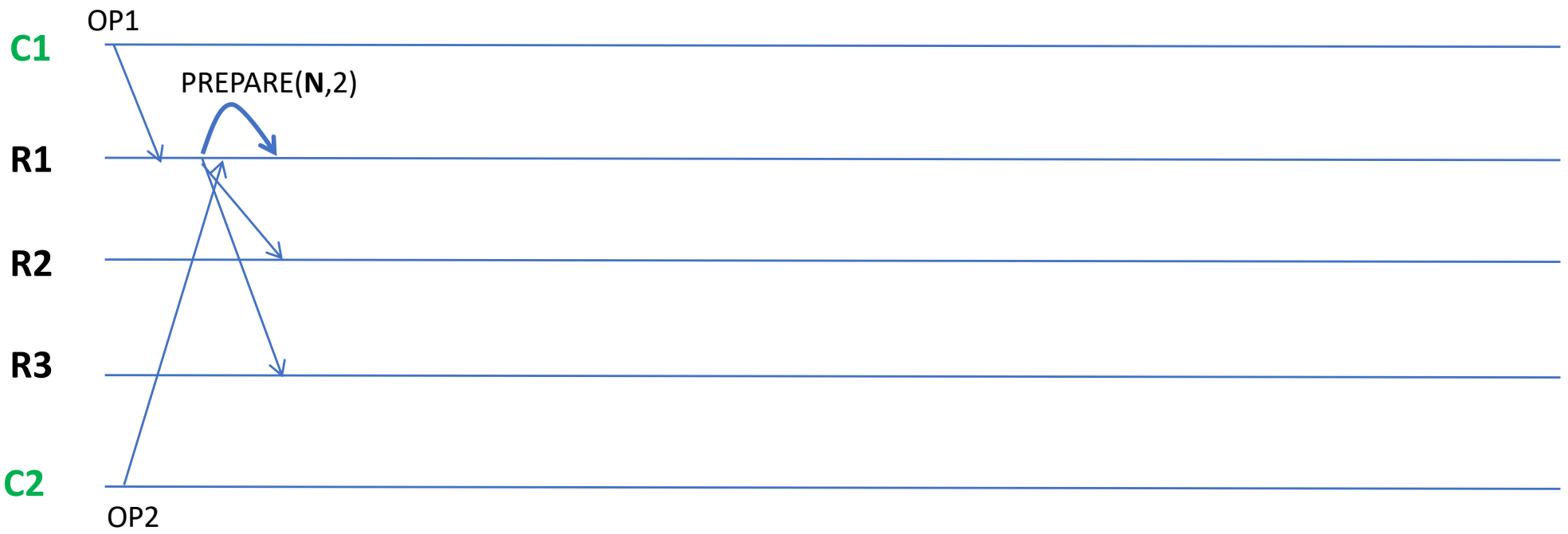
- Avoids the necessity of executing the prepare phase (more efficient in terms of communication steps).

- However:
  - What if the leader fails?
  - How do we select a leader?

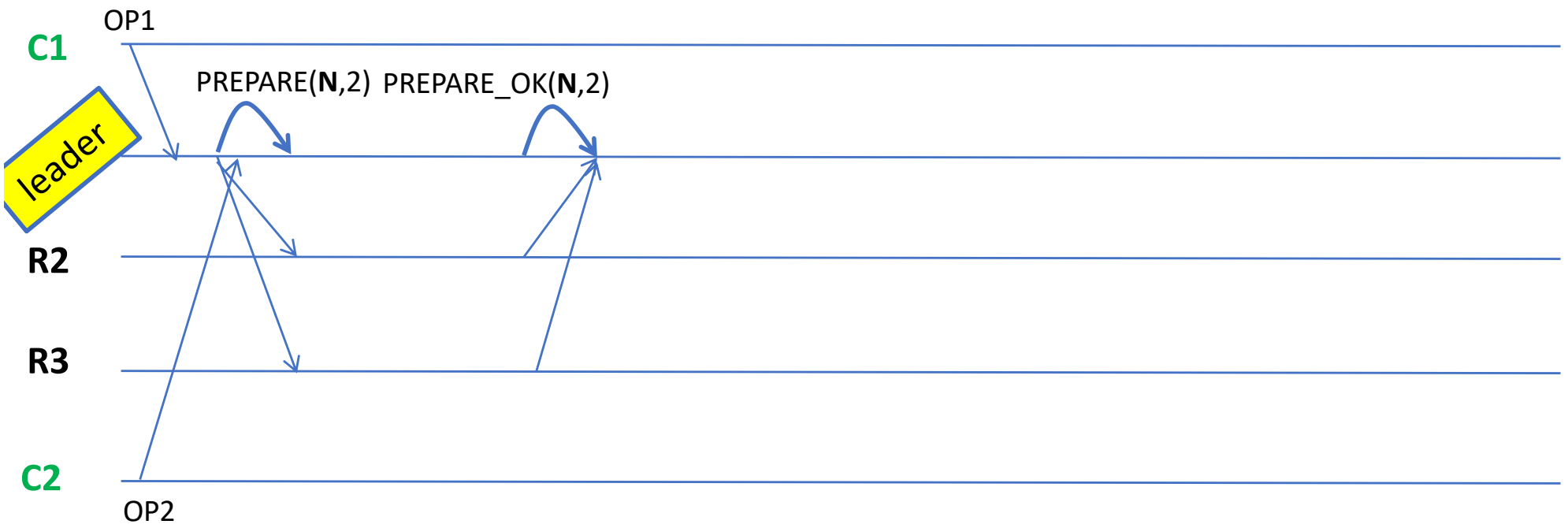
Multi-Paxos: Optimizing execution



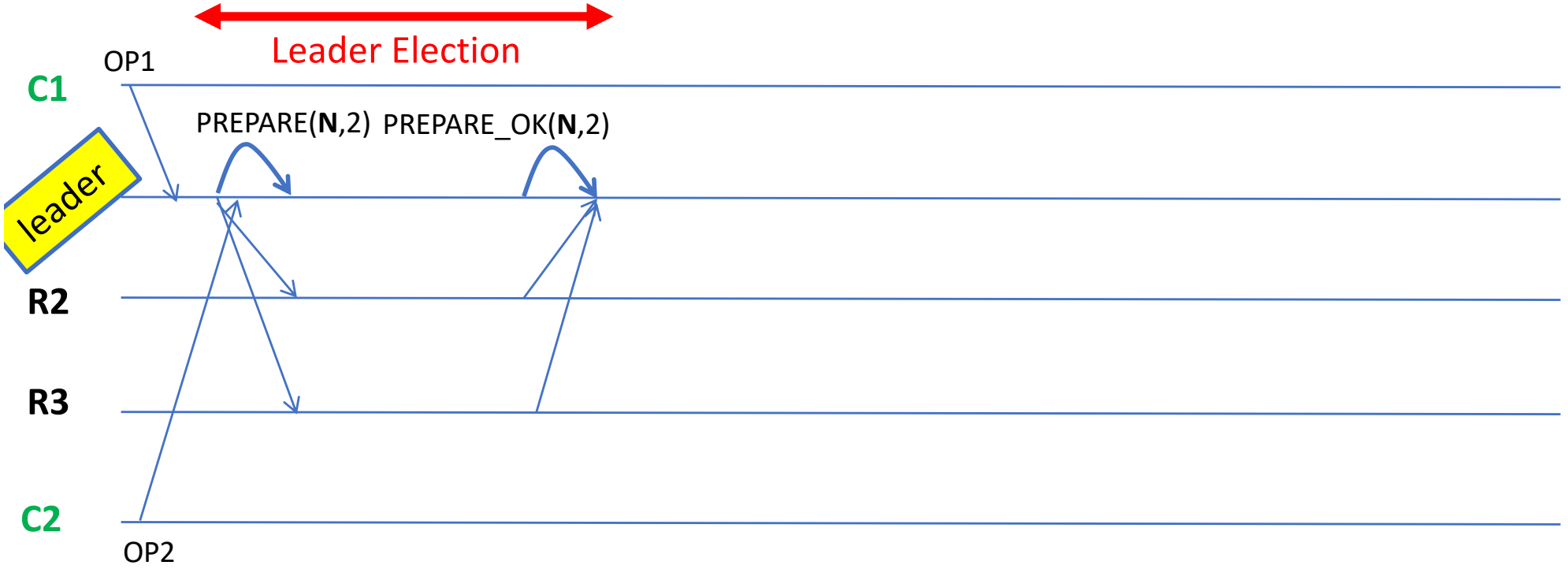
Multi-Paxos: Optimizing execution



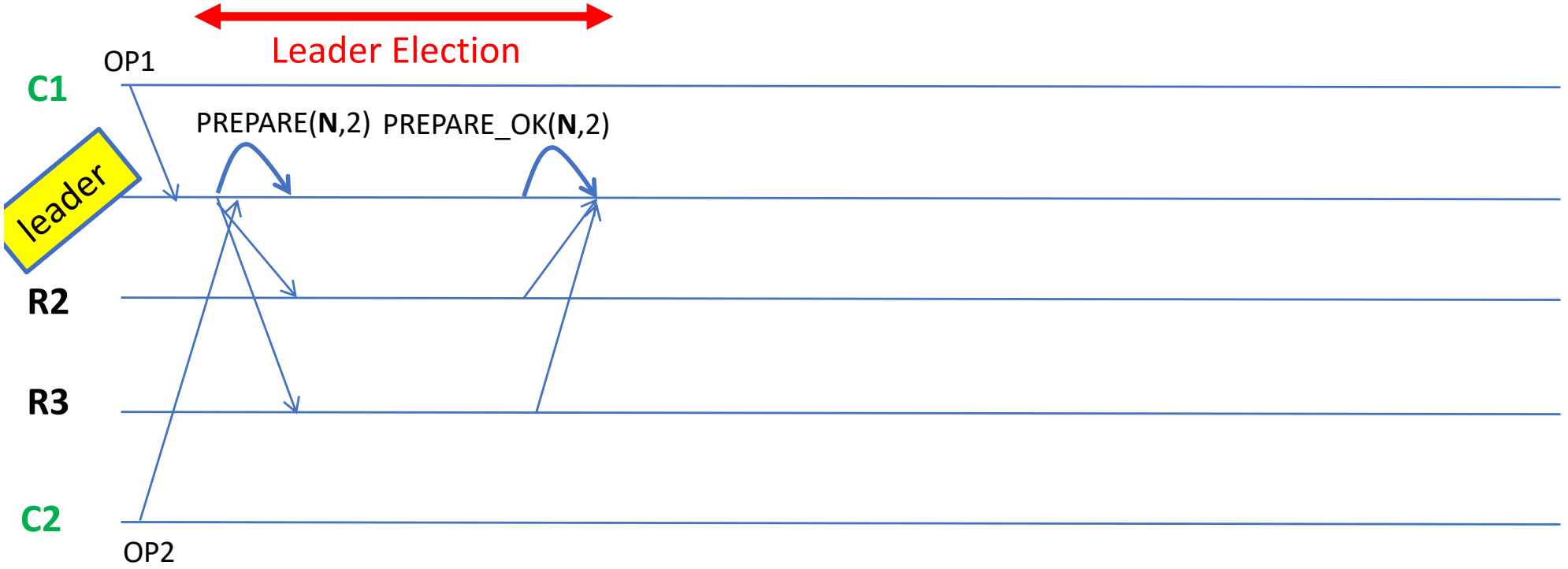
Multi-Paxos: Optimizing execution



Multi-Paxos: Optimizing execution

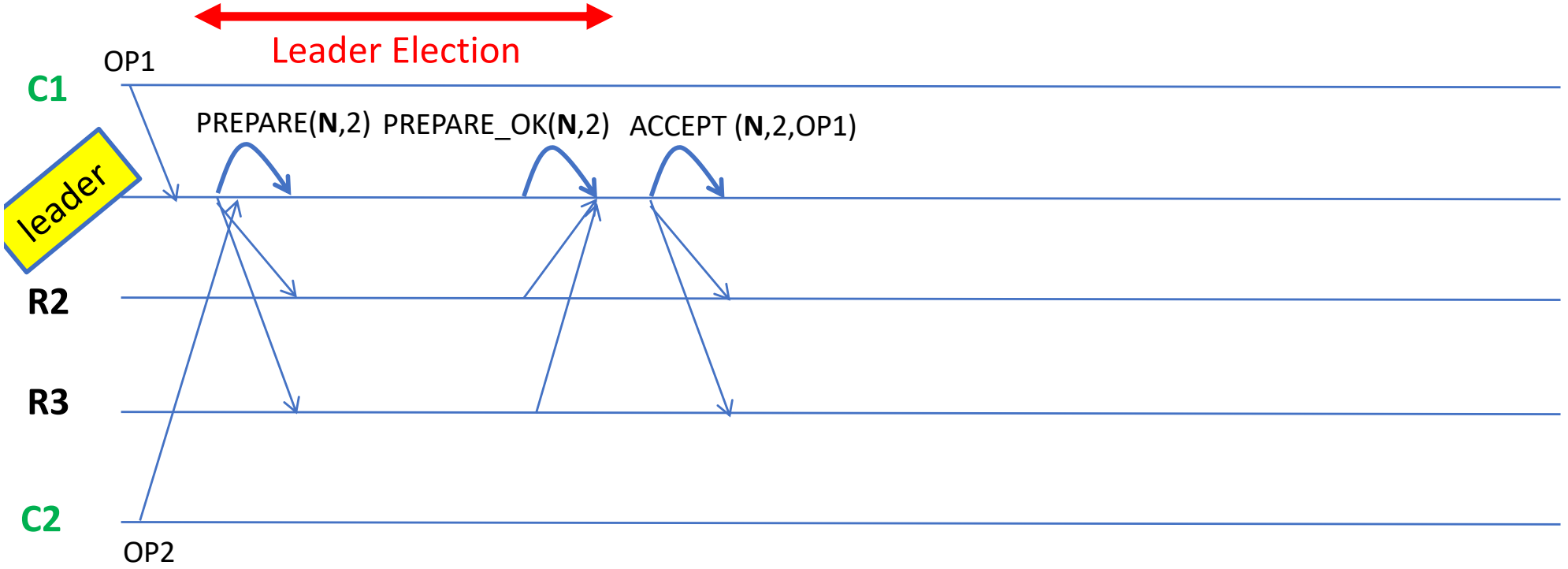


Multi-Paxos: Optimizing execution

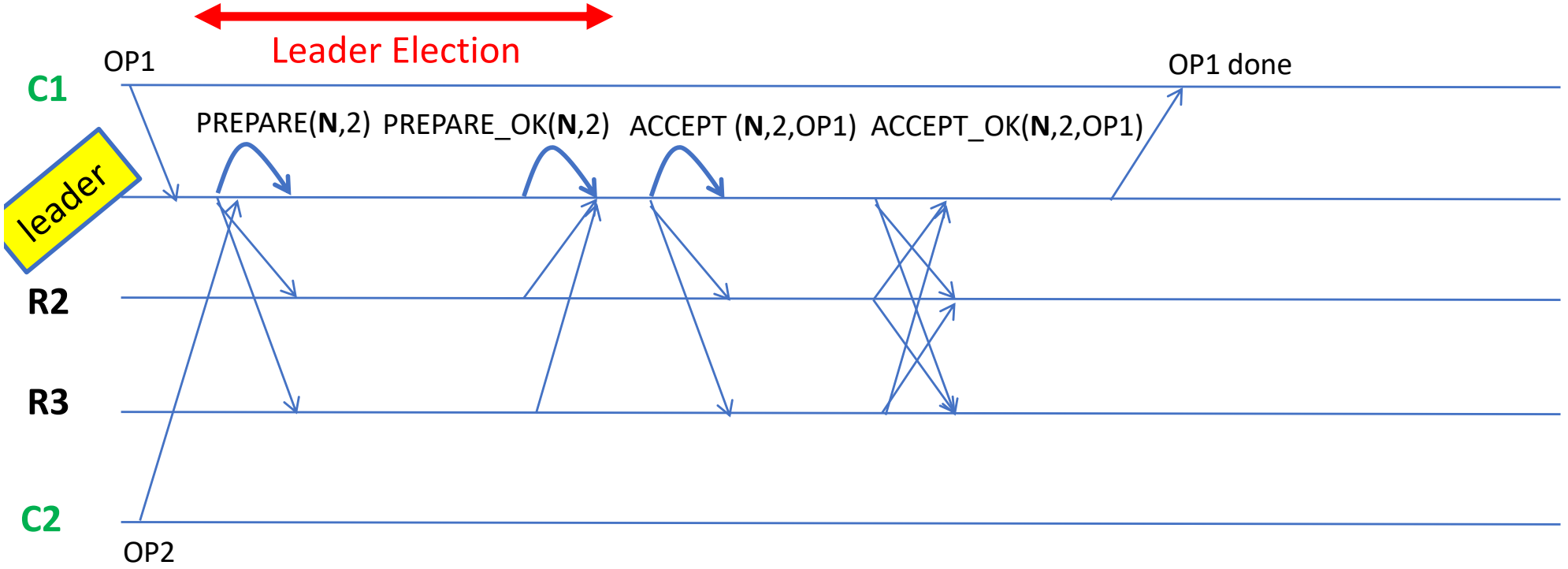


Effectively the prepara phase of paxos is selecting a leader for an instance...
Obviously, if the system is too asynchronous it might be impossible to select a leader (i.e, complete the prepare phase)

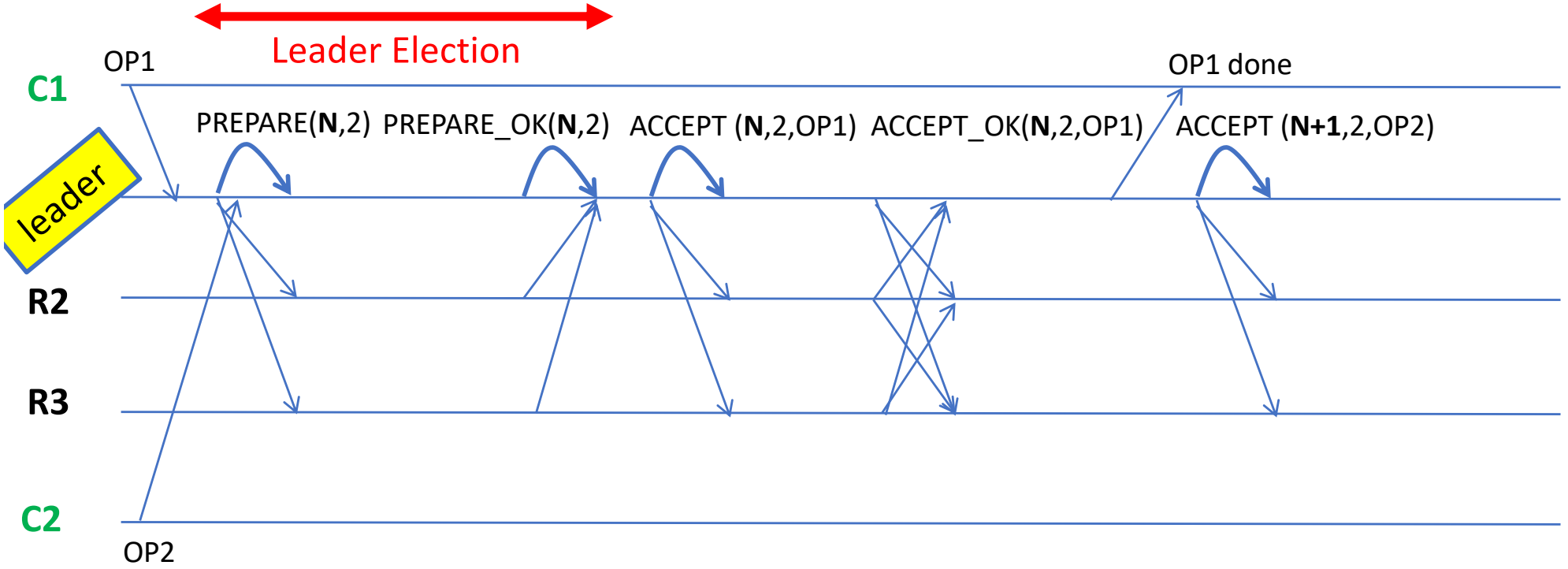
Multi-Paxos: Optimizing execution



Multi-Paxos: Optimizing execution

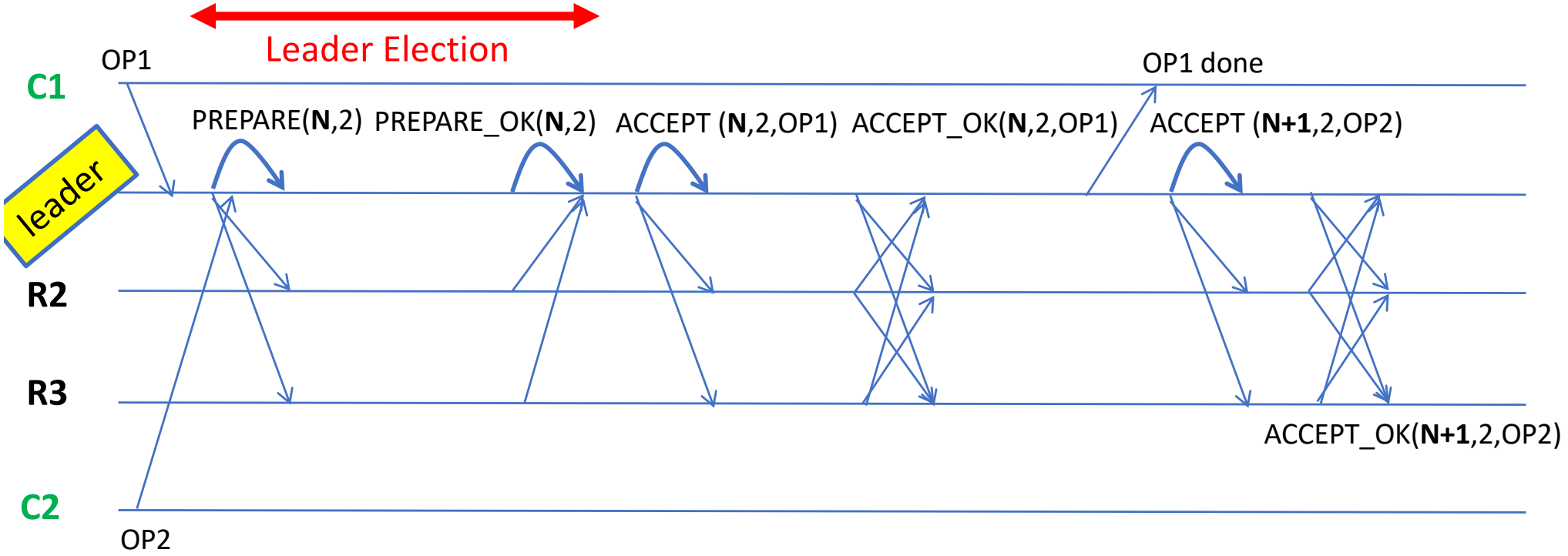


Multi-Paxos: Optimizing execution



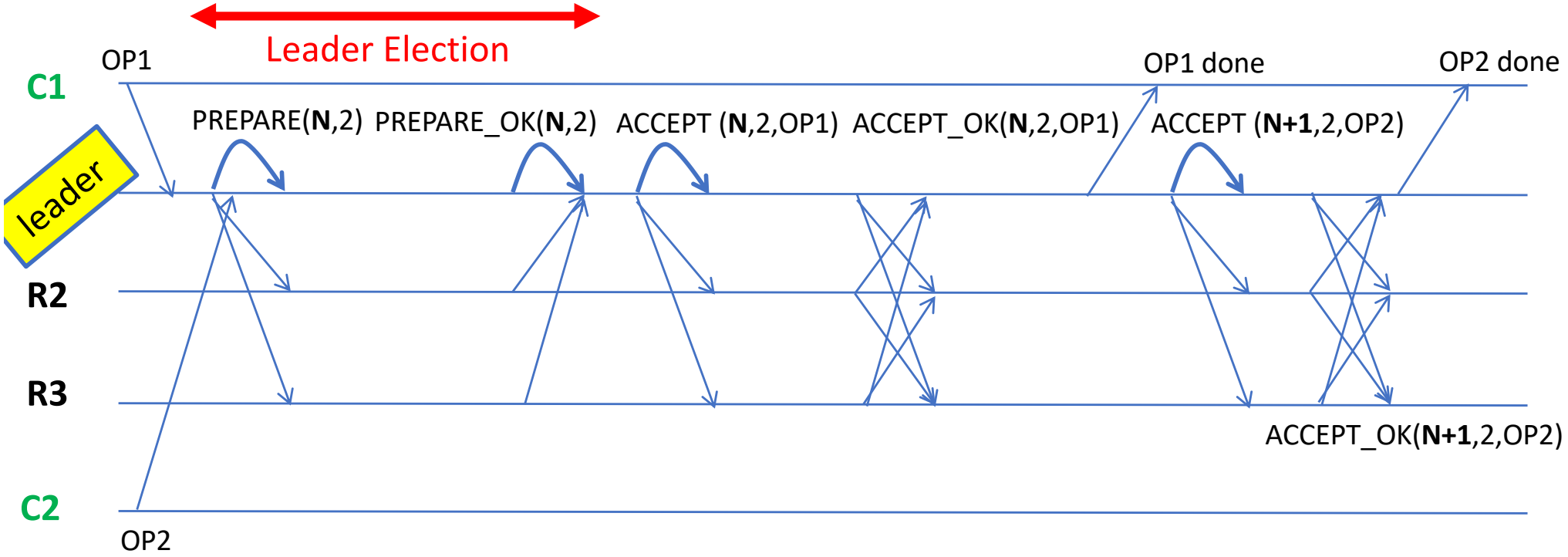
Leader can send ACCEPT immediately; it runs as if it's a prepare had already been accepted by all replicas

Multi-Paxos: Optimizing execution



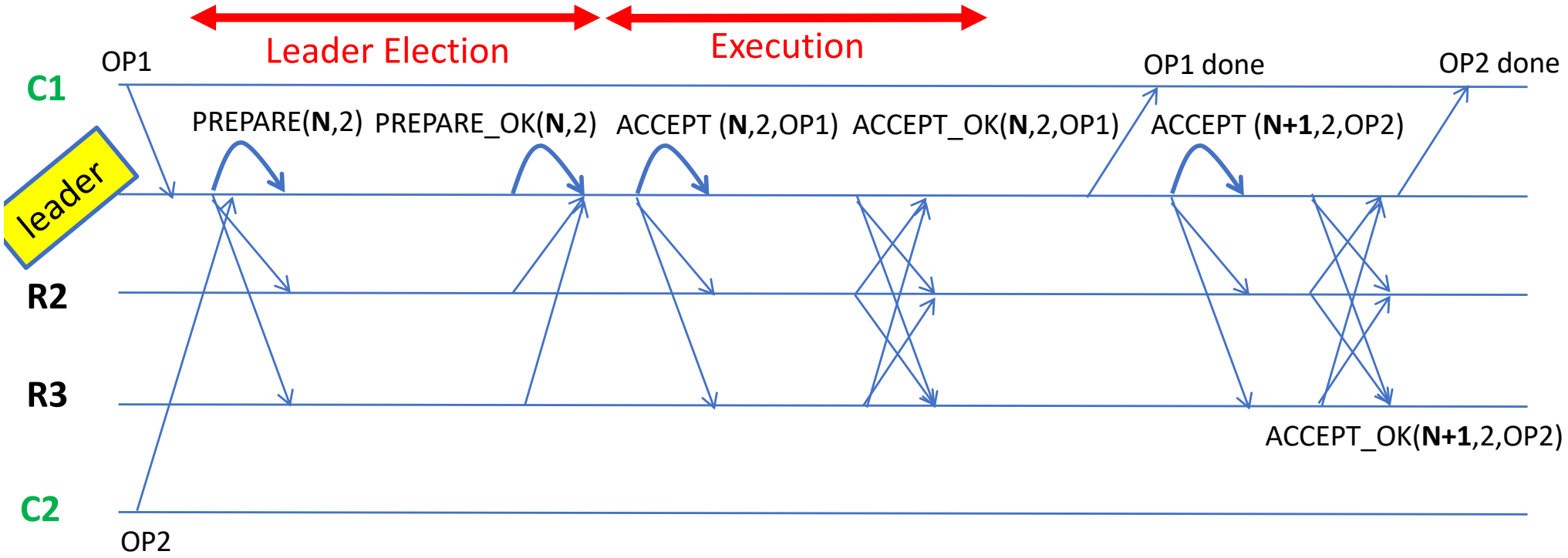
Leader can send ACCEPT immediately; it runs as if it's a prepare had already been accepted by all replicas

Multi-Paxos: Optimizing execution



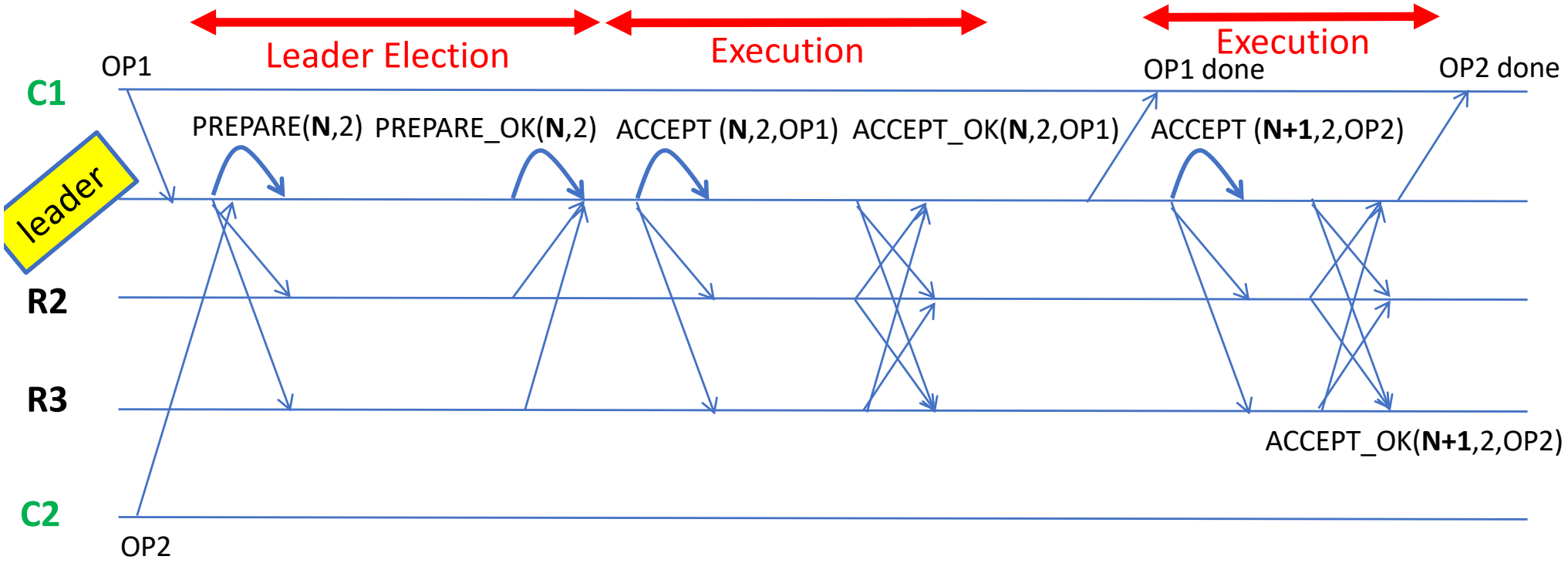
Leader can send ACCEPT immediately; it runs as if it's a prepare had already been accepted by all replicas

Multi-Paxos: Optimizing execution



Leader can send ACCEPT immediately; it runs as if it's a prepare had already been accepted by all replicas

Multi-Paxos: Optimizing execution



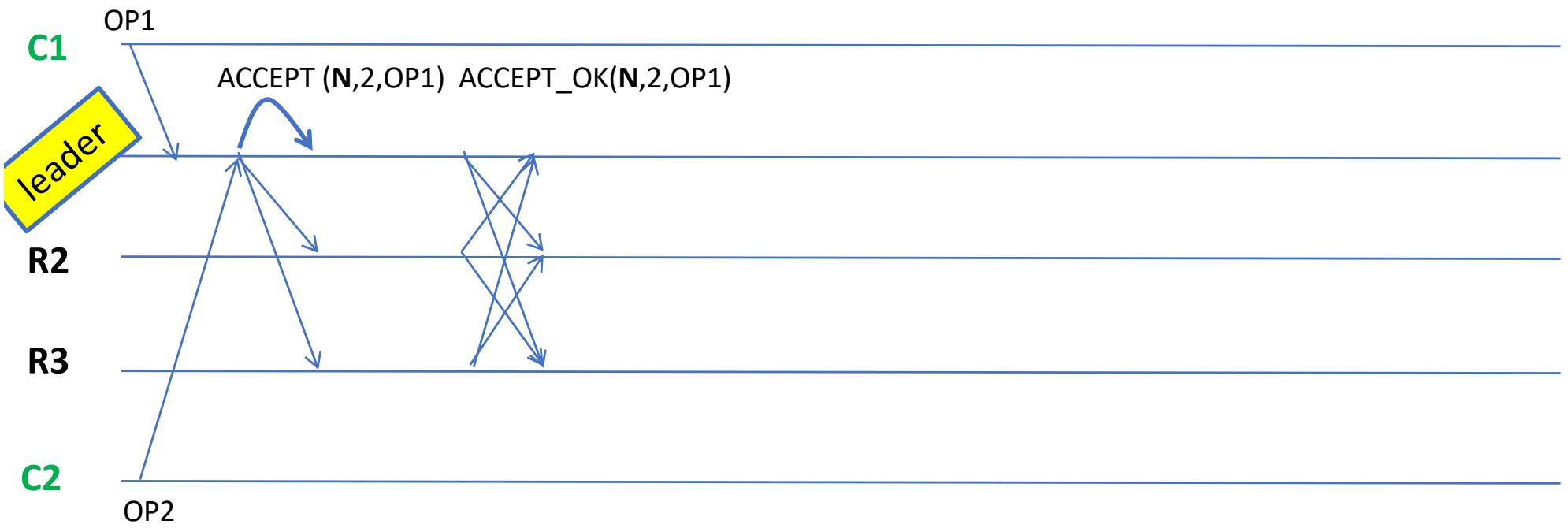
Leader can send ACCEPT immediately; it runs as if it's a prepare had already been accepted by all replicas

Paxos with a Leader

- Avoids the necessity of executing the prepare phase (more efficient in terms of communication steps).

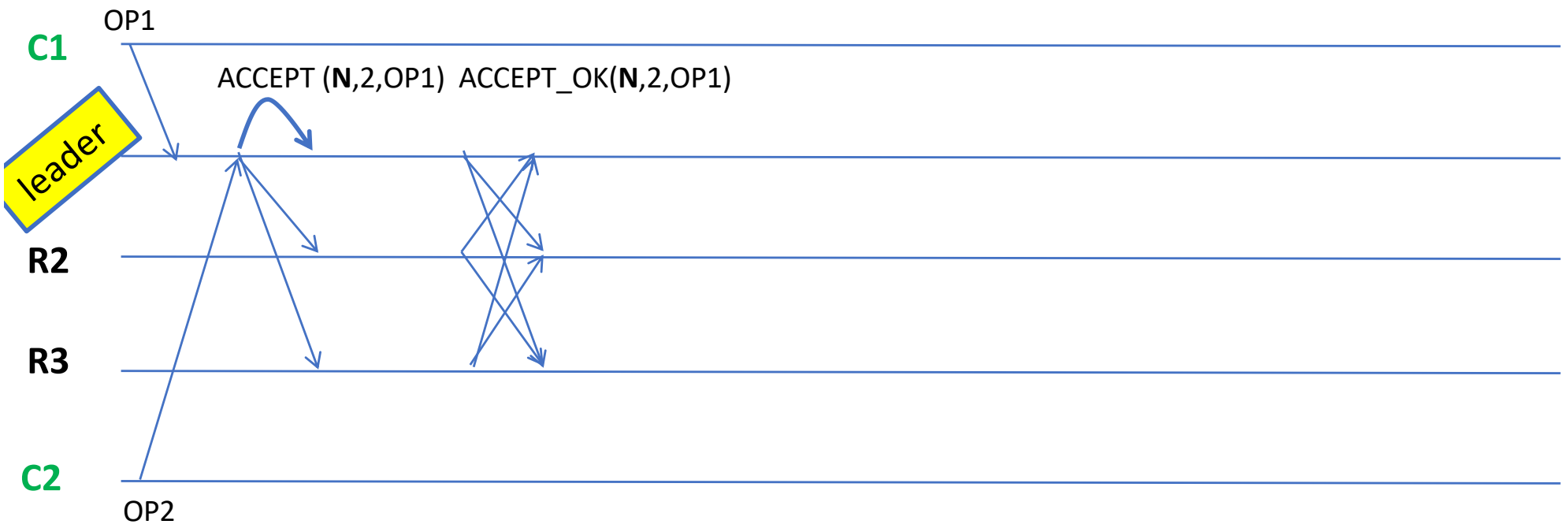
- However:
  - What if the leader fails?
  - How do we select a leader?

Multi-Paxos: Changing the Leader



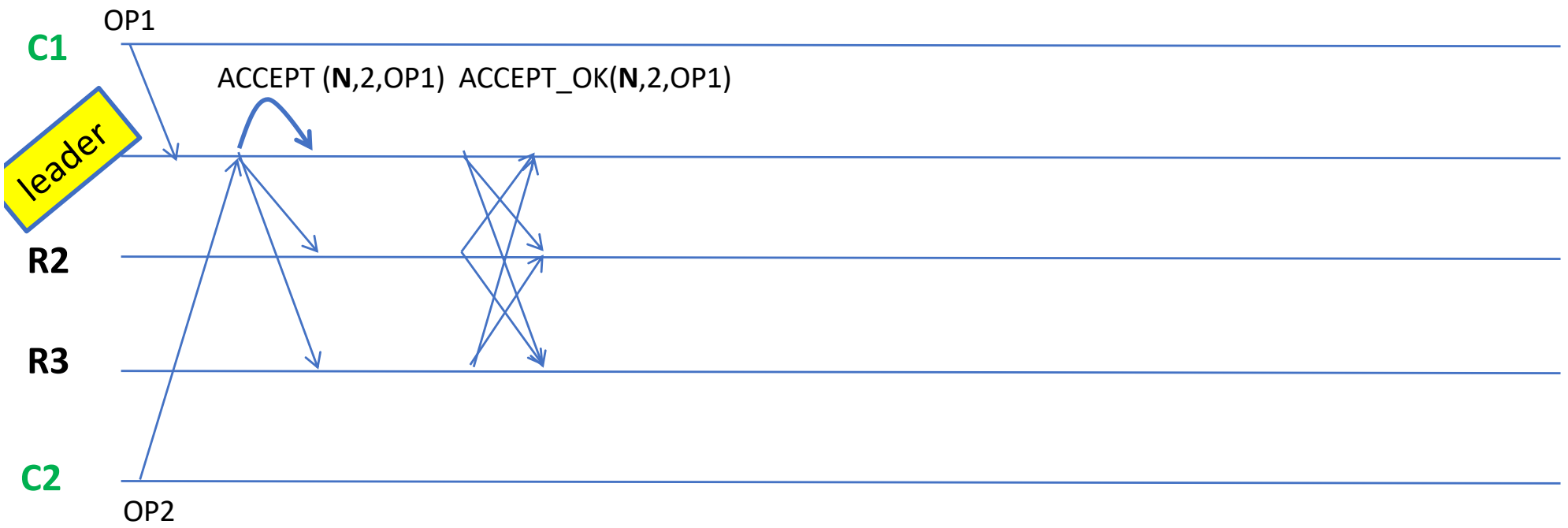
All replicas “monitor” the Leader in the sense that they expect the leader to continously make proposals (i.e., start new paxos instances)

Multi-Paxos: Changing the Leader



All replicas "monitor" the Leader in the sense that they expect the leader to continously make proposals (i.e., start new paxos instances)
What if there are no client operations?

Multi-Paxos: Changing the Leader

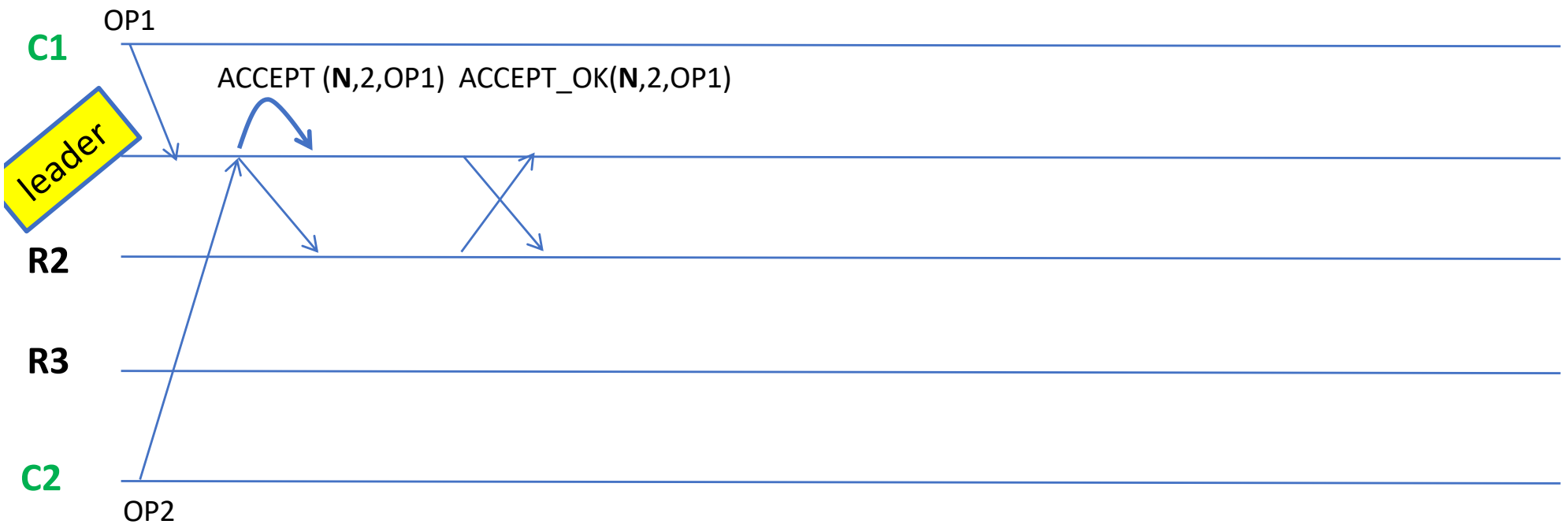


All replicas "monitor" the Leader in the sense that they expect the leader to continously make proposals (i.e., start new paxos instances)

What if there are no client operations?

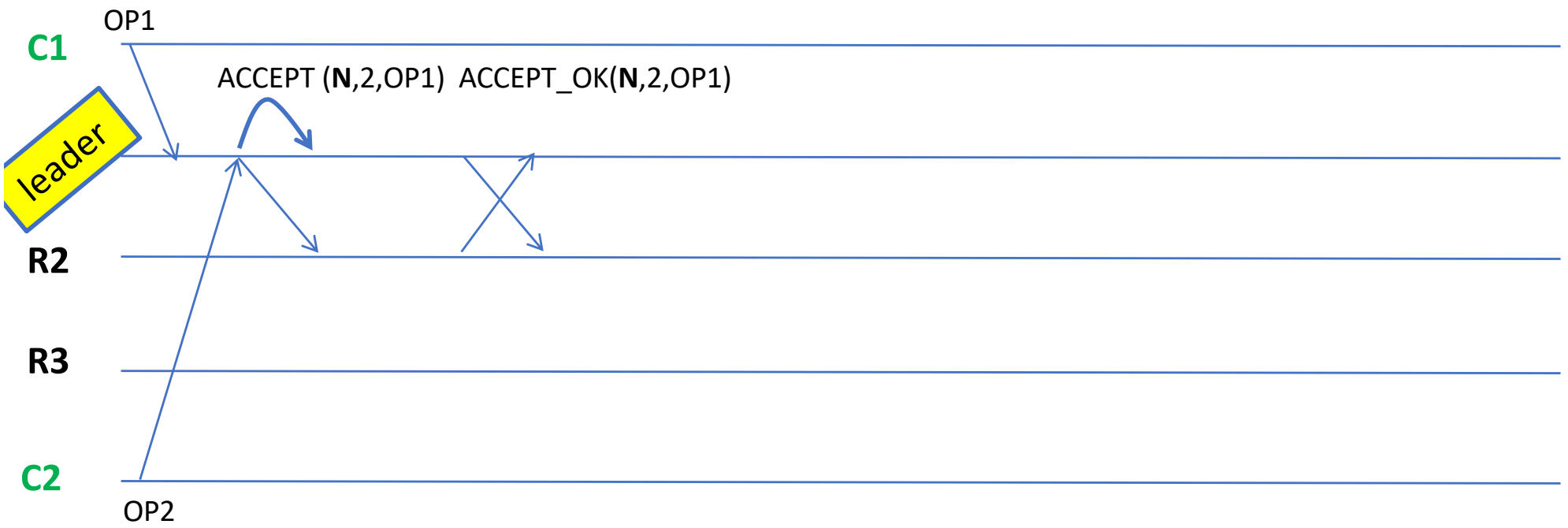
The leader can propose a NO-OP operation.

Multi-Paxos: Changing the Leader



Imagine that Replica 3, does not see the activities of the leader (due to asynchrony)

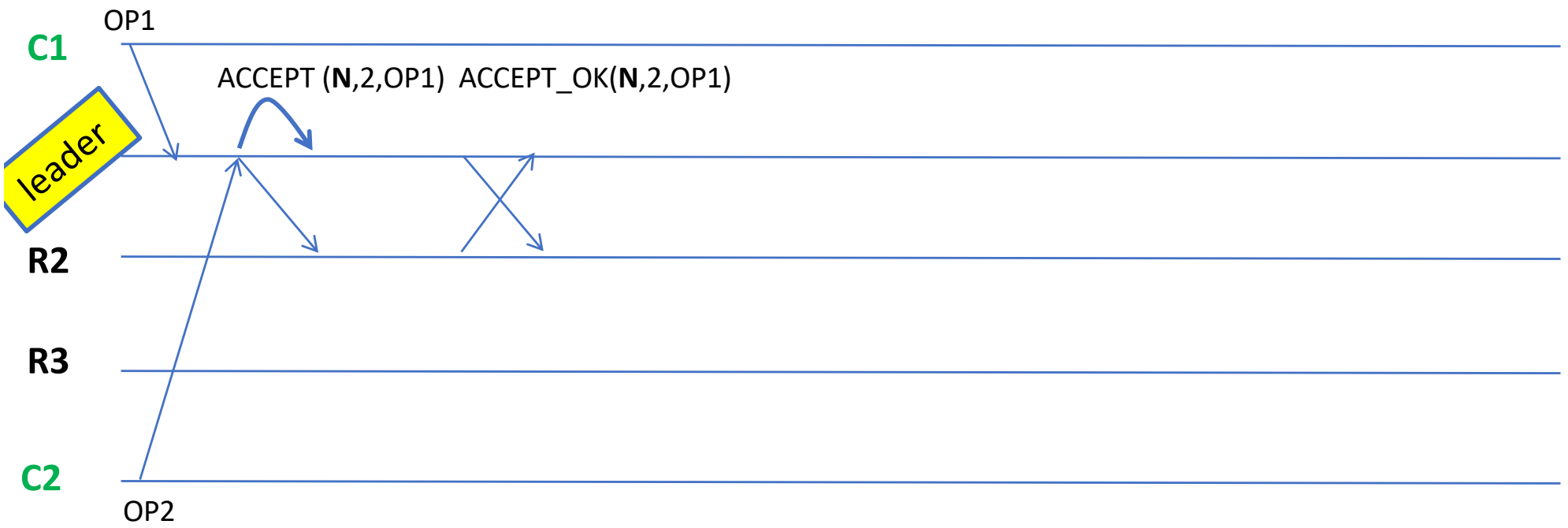
Multi-Paxos: Chaging the Leader



Imagine that Replica 3, does not see the activities of the leader (due to asynchrony)

R3 will suspect that the Leader has failed...

Multi-Paxos: Chaging the Leader

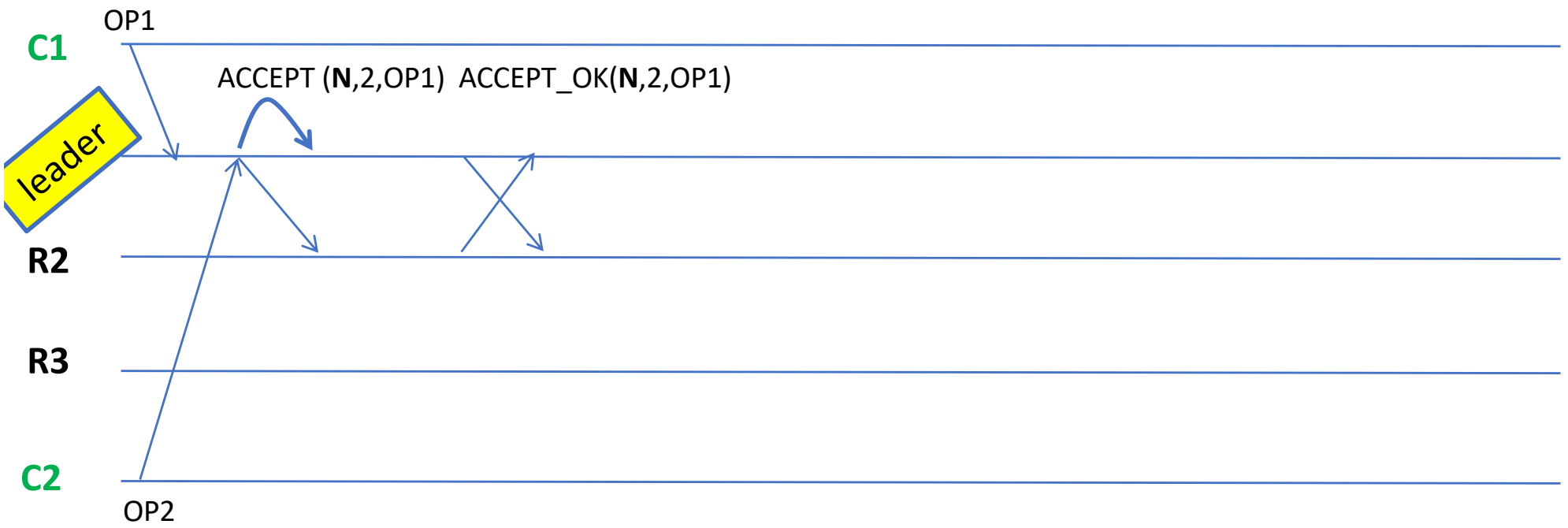


Imagine that Replica 3, does not see the activities of the leader (due to asynchrony)

R3 will suspect that the Leader has failed...

What should R3 do?

Multi-Paxos: Chaging the Leader



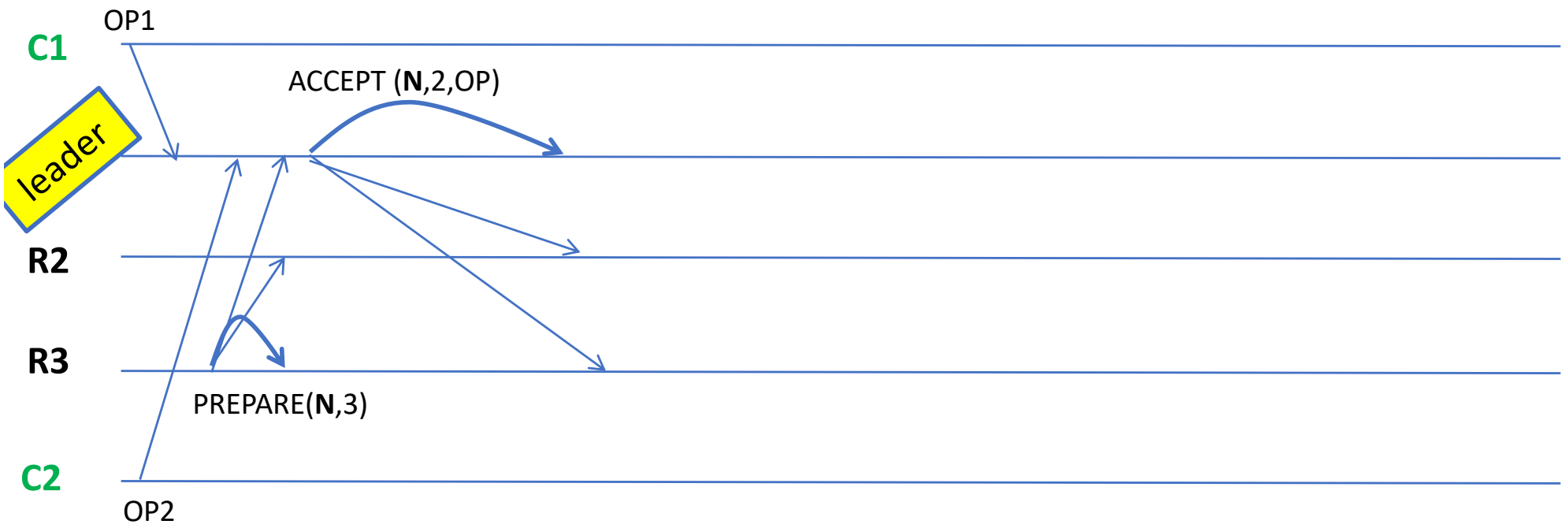
Imagine that Replica 3, does not see the activities of the leader (due to asynchrony)

R3 will suspect that the Leader has failed...

What should R3 do?

Try to become the new Leader by executing Prepare

Multi-Paxos: Chaging the Leader



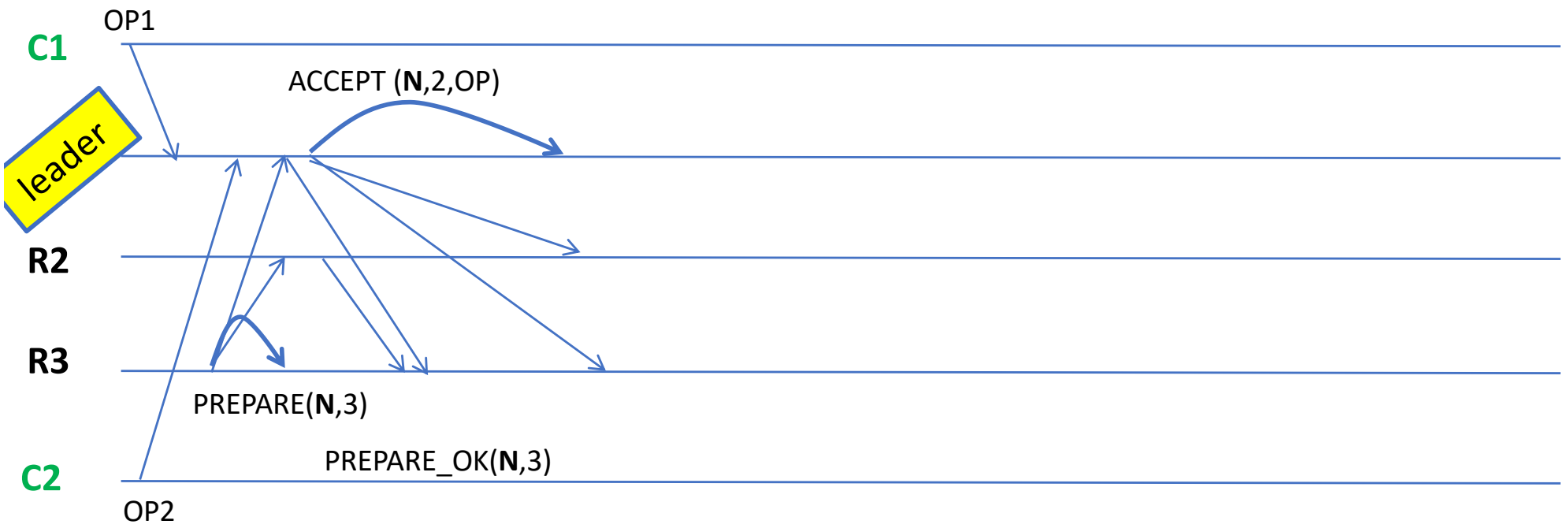
Imagine that Replica 3, does not see the activities of the leader (due to asynchrony)

R3 will suspect that the Leader has failed...

What should R3 do?

Try to become the new Leader by executing Prepare

Multi-Paxos: Chaging the Leader



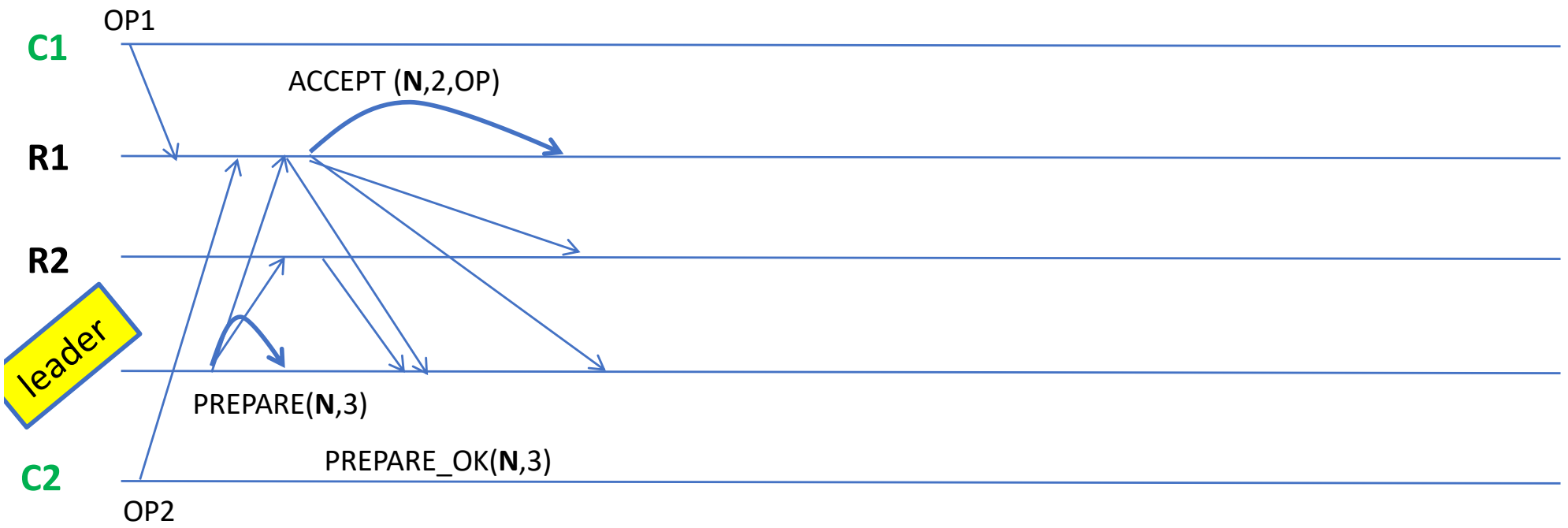
Imagine that Replica 3, does not see the activities of the leader (due to asynchrony)

R3 will suspect that the Leader has failed...

What should R3 do?

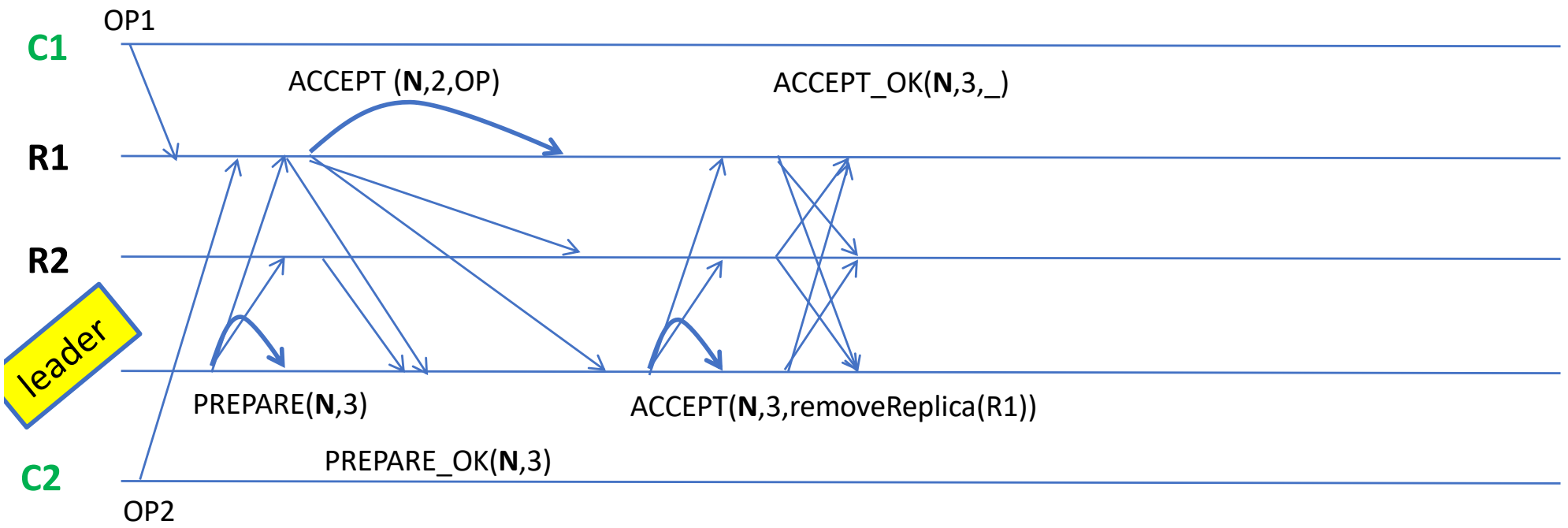
Try to become the new Leader by executing Prepare

Multi-Paxos: Chaging the Leader



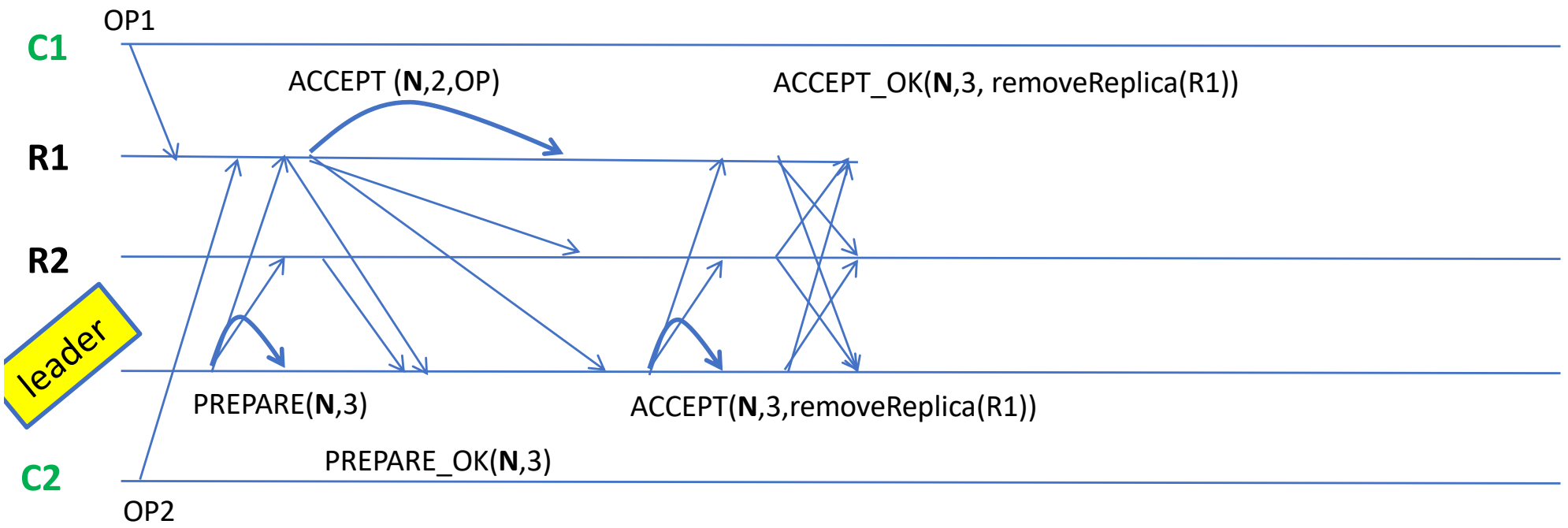
Now R3 becomes the leader, and the Accept issued by R1 is rejected, because there is a prepare with higher sequence number.

Multi-Paxos: Chaging the Leader



R3 can now propose to remove R1.

Multi-Paxos: Chaging the Leader



R1 is no longer part of the system and R3 has taken over as leader...

Removing a suspected leader might not be smart…

- Evidently, the current leader, when receives a prepare from another replica with a higher sequence number, can avoid to reply, and execute a prepare of its own with a higher sequence number (might not be enough since a majority might already have accepted the first prepare).

Removing a suspected leader might not be smart…

- Evidently, the current leader, when receives a prepare from another replica with a higher sequence number, can avoid to reply, and execute a prepare of its own with a higher sequence number (might not be enough since a majority might already have accepted the first prepare).
- A new leader might not immediately remove a previously suspected node, it might delay this to avoid creating more instability in the system.

Attention to the prepare

- What if a replica becomes leader in some instance n, and the previous leader had already locked-in values for instances up to m, where m > n?

Attention to the prepare

- What if a replica becomes leader in some instance n, and the previous leader had already locked-in values for instances up to m, where m > n?

- Prepare_OK messages have to reported values accepted for any instance >= n.
- The new leader will have to re-execute (i.e., issue accept messages) for all those replicas using the values that are reported in Prepare_OK messages (as in the original Paxos).

Summary of Multi-Paxos

- The previous modifications (optimizations) to Paxos are known as Multi-Paxos.
- The intuition is:
  - To have an explicit leader (or a distinguished proposer).
  - Imbue the membership management into the state machine.
  - Have a single prepare phase to be used to execute multiple accept phases in sequence by the leader.

Summary of Multi-Paxos

- Details:
  - Since only the leader proposes, client requests either are all directed at the leader, or should be redirected from replicas that receive them to the leader.
  - The leader can batch multiple operations (in an order defined by him) in a single Paxos instance.
  - The leader can also start multiple Paxos instances concurrently (i.e., instance n, n+1, n+2, n+3) all with different values. Replicas (including the leader) can only execute operations following the strict instance order however.

Warning about Multi-Paxos in the Literature

Paxos Made Simple

Leslie Lamport

ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) | December 2001, pp. 51-58

Download BibTex

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.

View Publication

Research Areas

Algorithms

Multi-Paxos is implicitly mentioned as a possible optimization of Paxos by Leslie Lamport in his 2001 Paper.

Warning about Multi-Paxos in the Literature

Multi-Paxos: An Implementation and Evaluation

*Hao Du** *David J. St. Hilaire**

Abstract

We implemented a fully functional sequential Multi-Paxos system and a prototype parallel Multi-Paxos system. The throughput of the system under various settings of the Paxos cluster size, network latency, and window size (of the parallel Paxos) were evaluated, giving us insight into how and when these factors affect the performance. Additonally, despite existing methods for electing distinguished proposer, we proposed and evaluated a different simple yet effective approach to determine the distinguished proposer.

respond with a rejection or with a Promise. A Promise tells the Proposer that the Acceptor will not accept a Proposal with a lower proposal number. A Promise also contains the last proposal value that the Acceptor accepted. If the proposer receives a Promise from a majority of the Acceptors, it can move to the Propose phase. In this phase, the Proposer sends a Proposal message containing the Paxos iteration number, the proposal number, and the proposal value (this is either the value returned in the Prepare phase or a value of the Proposer's choosing if no value was obtained in the Prepare phase) to the Acceptors. If the Acceptors have not promised to only ac-

It is experimentally evaluated by Hao Du and David Hilaire in this (somewhat obscure) technical report from 2009

Warning about Multi-Paxos in the Literature

- Typically people assume that after a complete execution of the two phases of Paxos by a new leader, in the following instances the leader uses a special sequence number (typically zero) in all other accepts (to denote that he had become a leader in a previous round).
- I have presented an alternative solution where the leader keeps using the same sequence number that he used in his (successful) prepare. This is not common (as far as I know) but not doing it leads to multiple issues on implementations and proofs (particularly with parallel instances).
- There is a paper under submission that explains this aspect:

  ChainPaxos: When Chain Replication Meets Paxos
  Pedro Fouto, Nuno Preguiça, and João Leitão
  Under submission to Eurosys 2020.

Homework 4:

- Not this week…