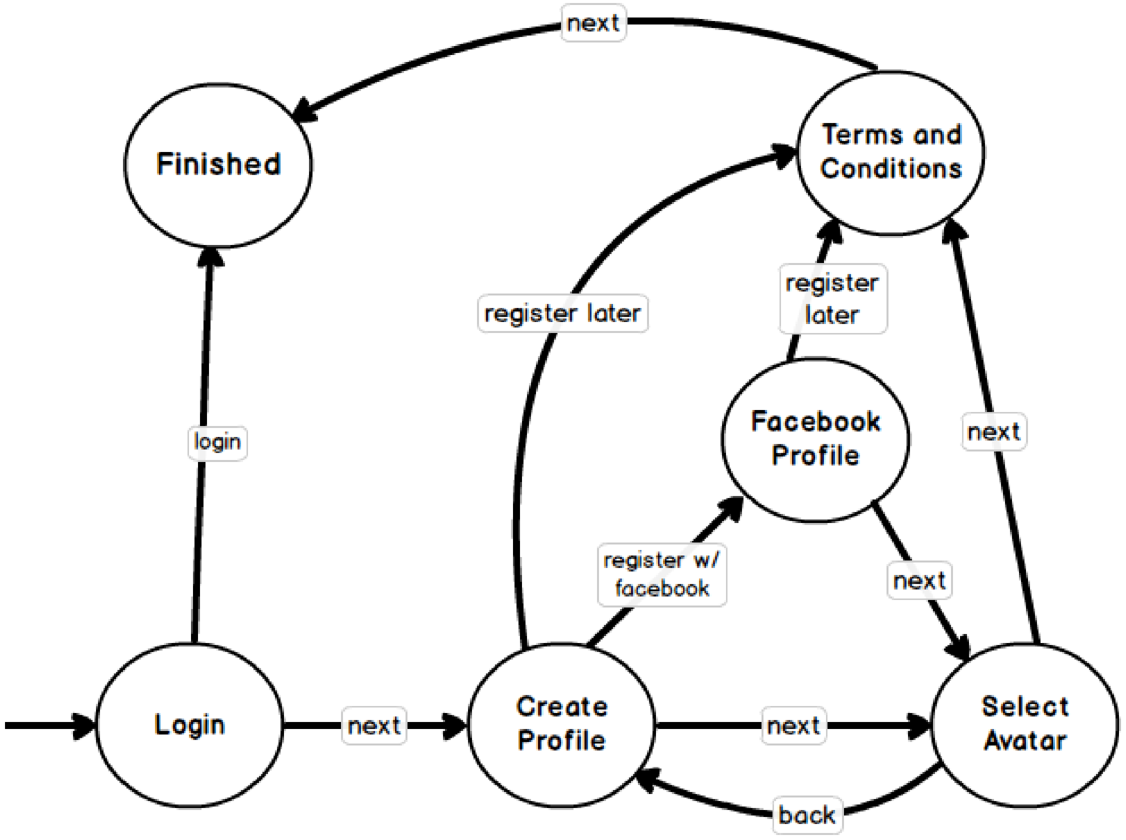
SCC.311: Paxos

Onur Ascigil

Week 5, Lecture 2

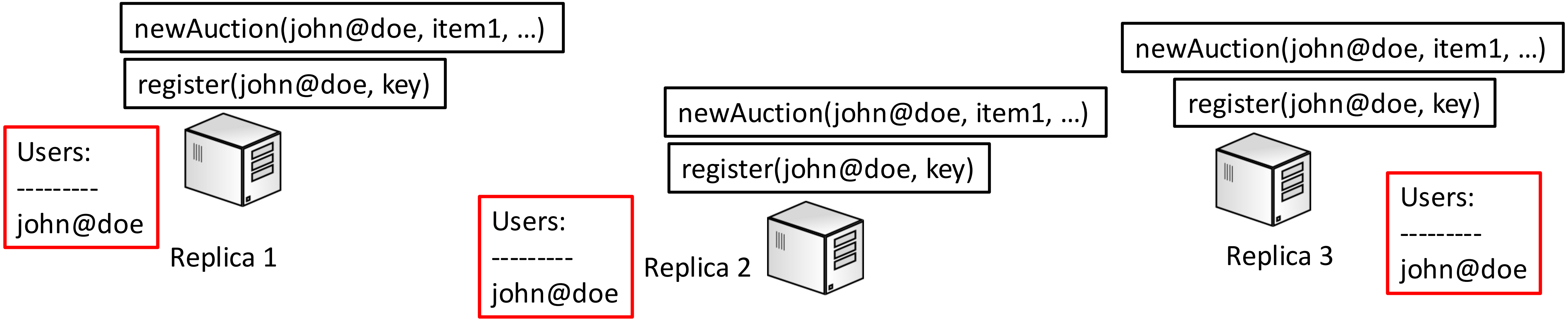
Replication Revisited

- As a system gets larger and needs to tolerate failures:
  - **Replication** is the solution
  - Each replica:
    - runs the same copy of the service (i.e., same code)
    - serves operations from the clients



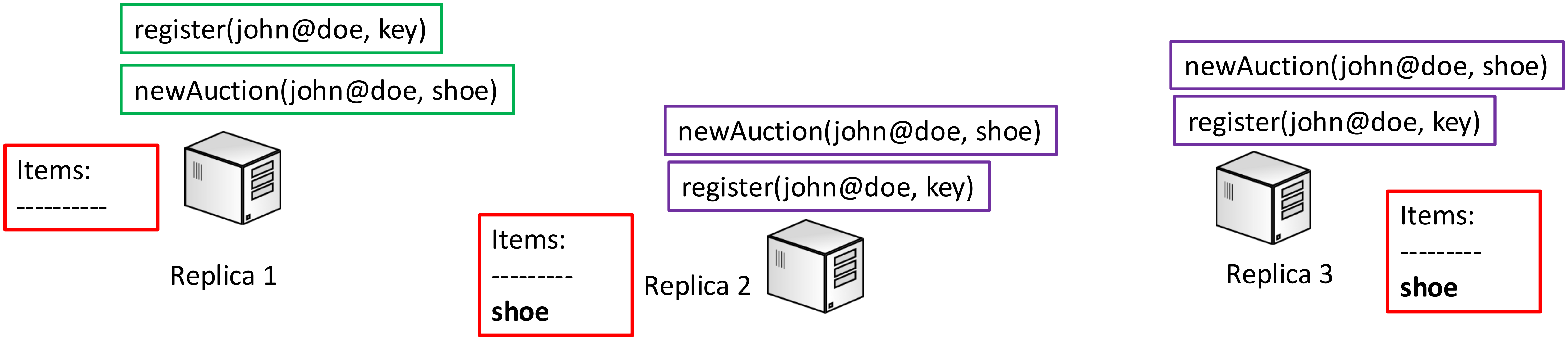
State Machine Replication

- When all the replicas start with the same state and each processes the client requests (i.e., operations) in the same order, replicas reach a consistent state
- This is referred to as: “**State Machine Replication**”



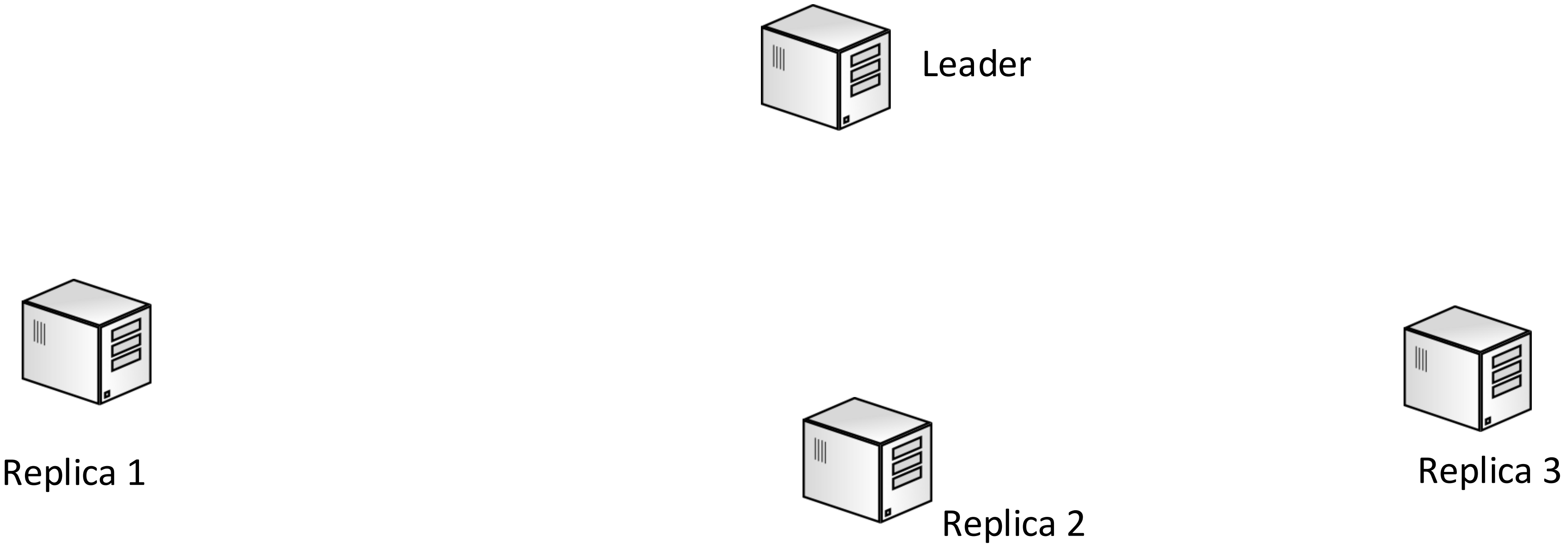
State Machine Replication

- What if the replicas process the operations in different orders?
- Consider the scenario below in a replicated auction service (from your coursework):
  - Replica1 processes *newAuction* RPC first and then the *register* RPC
  - This leads to Replica1 rejecting the newAuction (because the user john@doe does is not registered) and diverging from others



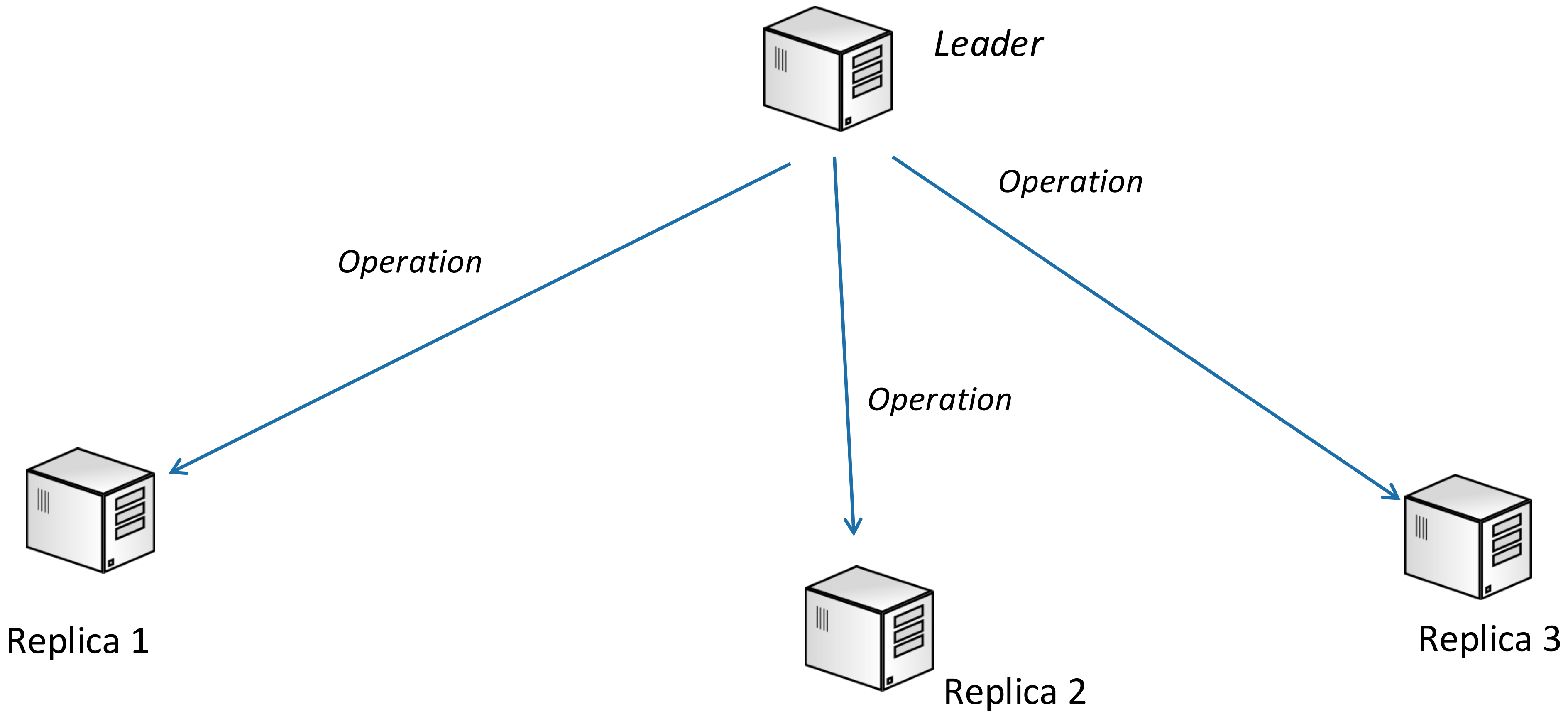
State Machine Replication

- How should we implement State Machine Replication?
  - A strawman solution: **atomic multicast**



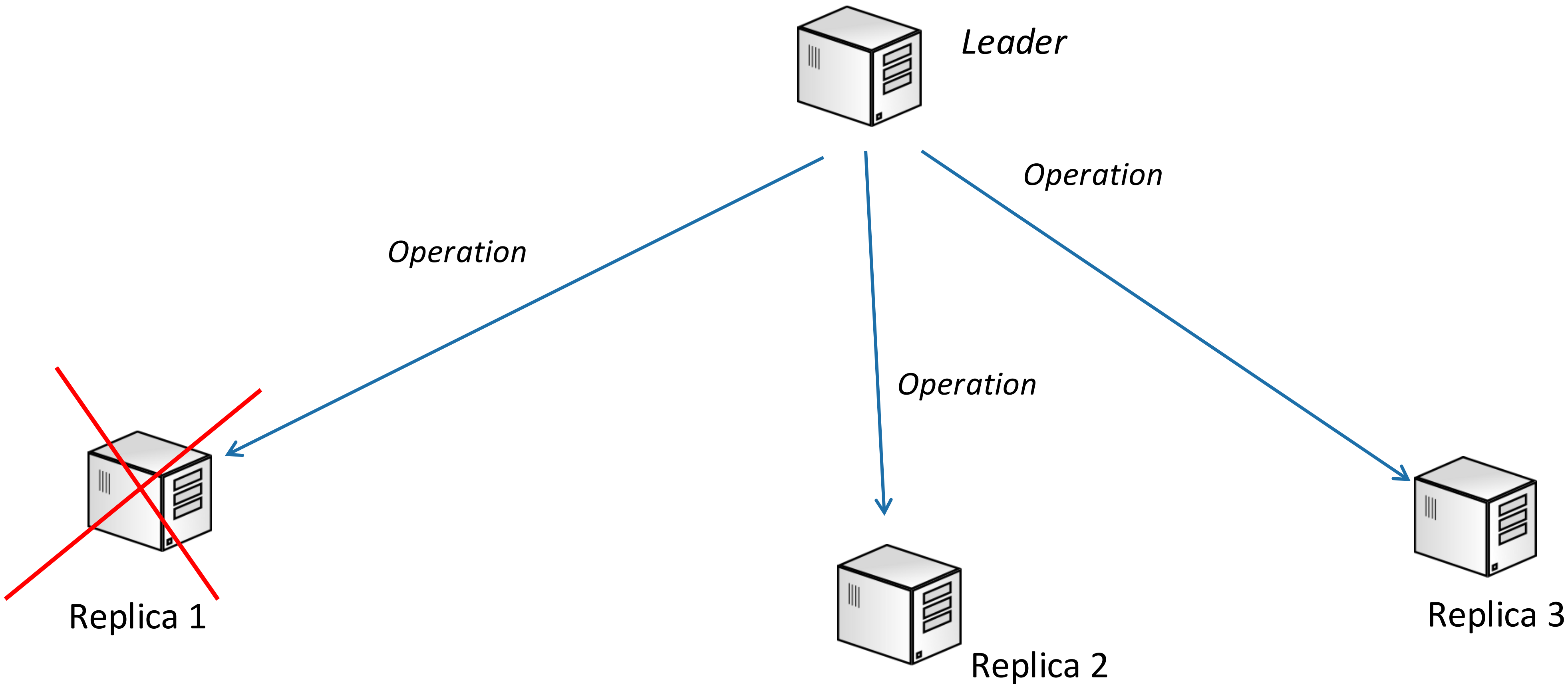
A Strawman Solution: Atomic Multicast

- How should we implement State Machine Replication?



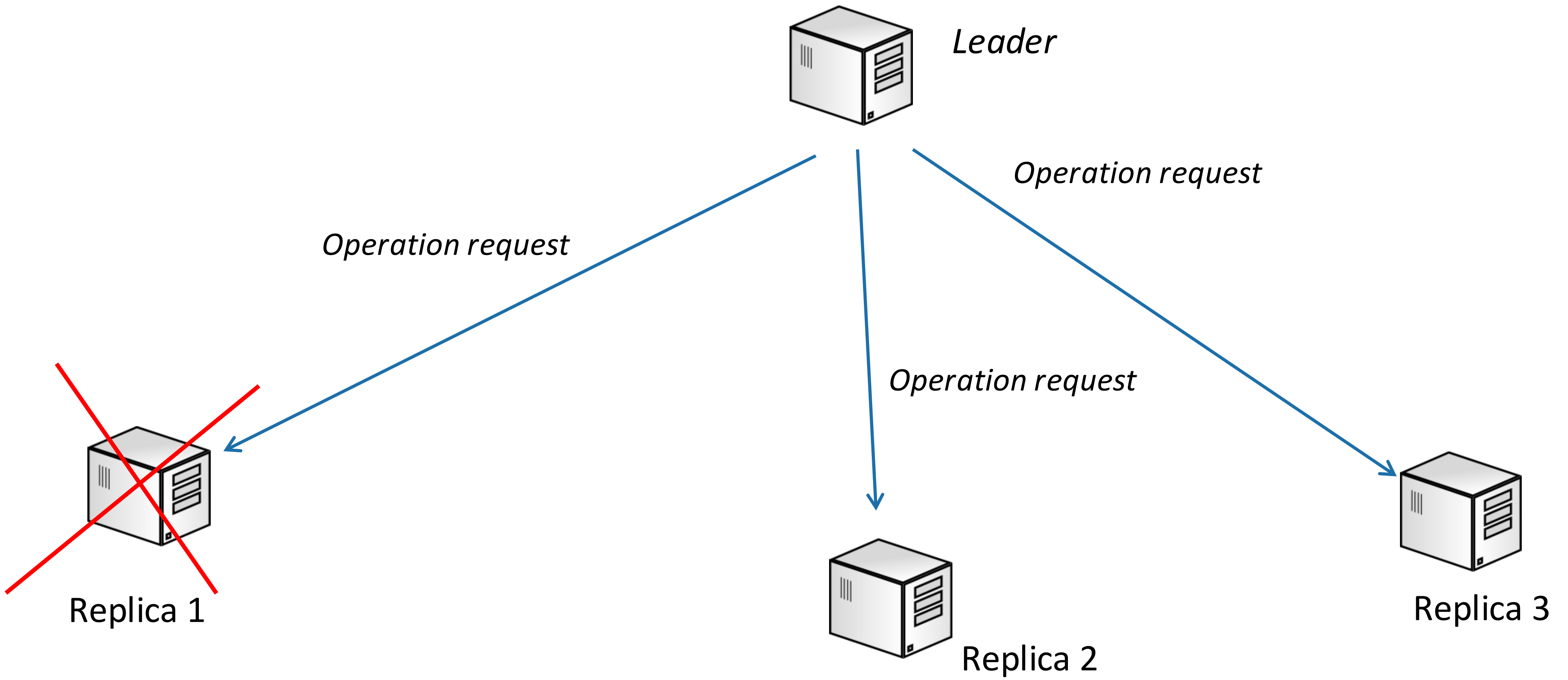
A Strawman Solution: Atomic Multicast

- What if a replica fails?



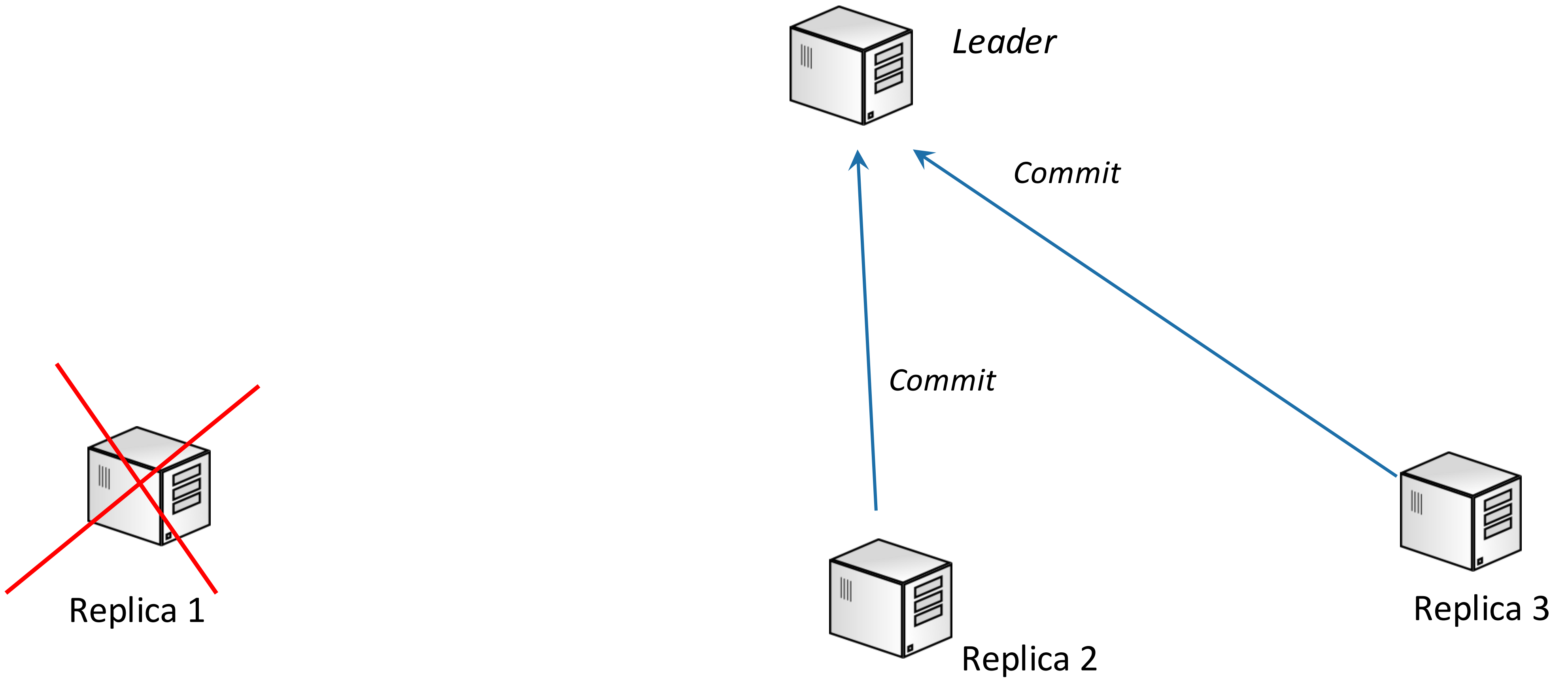
A Strawman Solution: Atomic Multicast

- What if a replica fails?
  - In that case, the leader must abort the operation
    - Remember, atomic multicast is “all or nothing” *(See Week 2, lecture 1)*



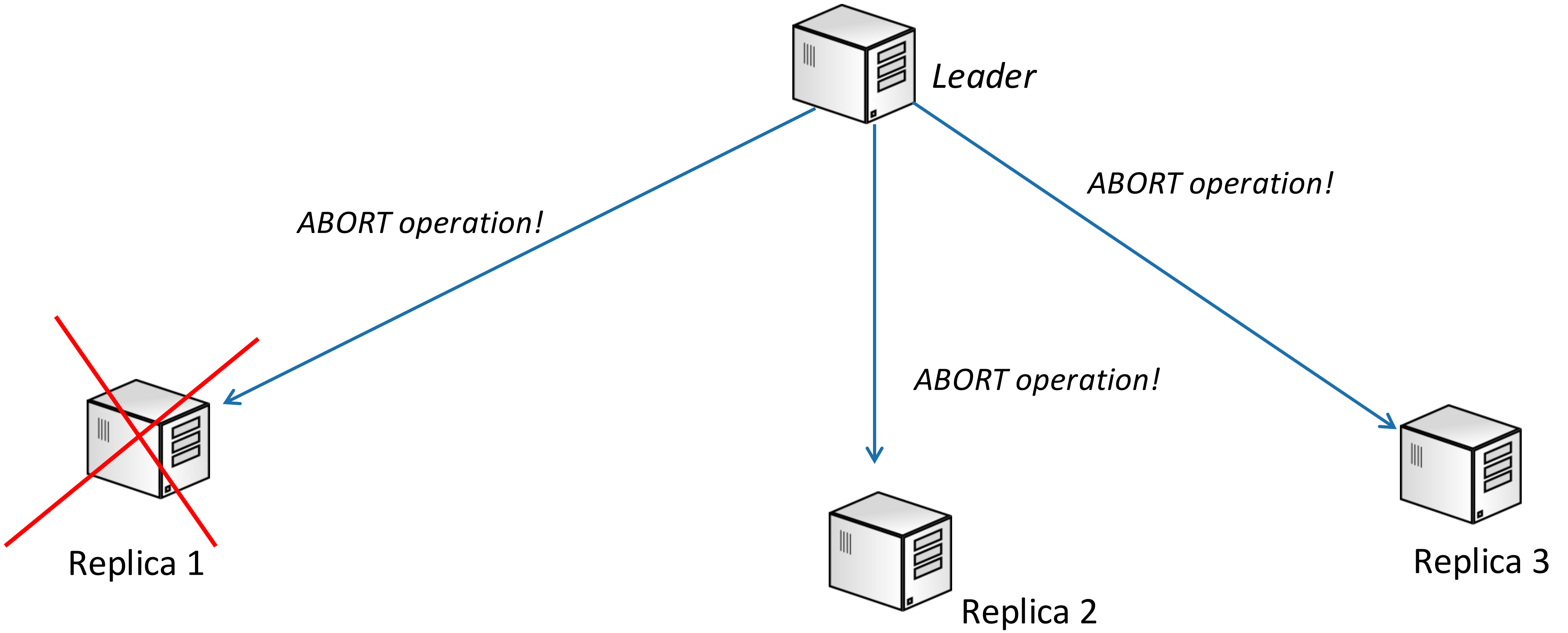
A Strawman Solution: Atomic Multicast

- What if a replica fails and does not repond with a commit?
  - In that case, the leader must abort the operation



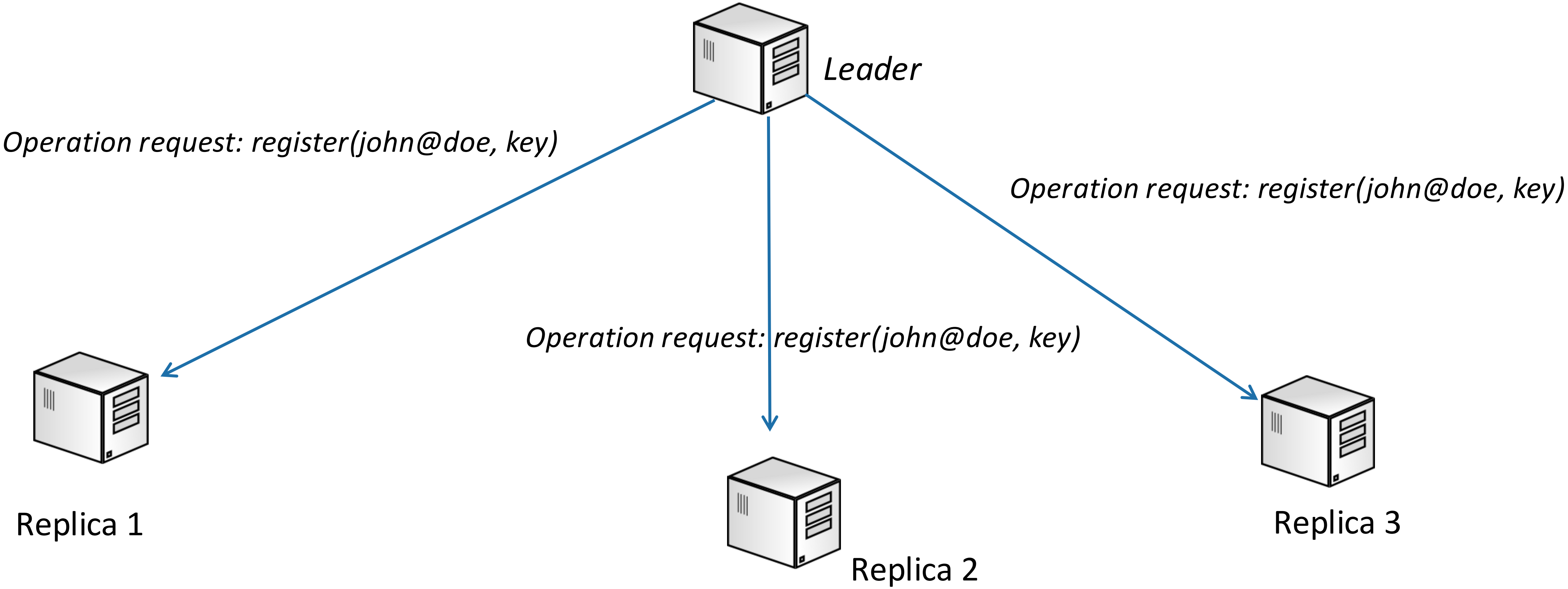
A Strawman Solution: Atomic Multicast

- Leader sends each replica an “operation request”
- If all the replicas respond with a “commit”, then the leader sends an “ACK” to everyone
- In case one or more replicas do not respond with a commit (after resending operation request), the leader gives up and sends ABORT



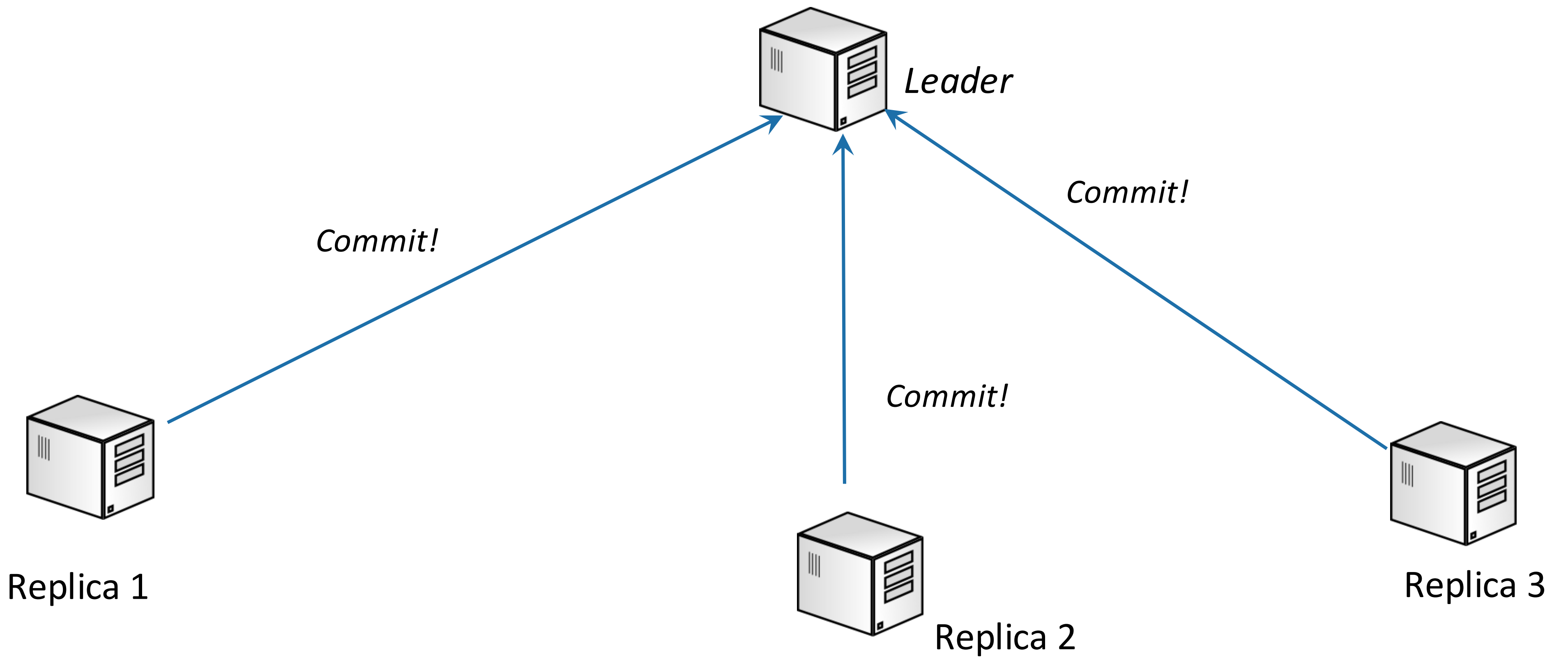
A Strawman Solution: Atomic Multicast

- In the absence of failures, the leader submits operations one-by-one and the replicas commit them in the submitted order



A Strawman Solution: Atomic Multicast

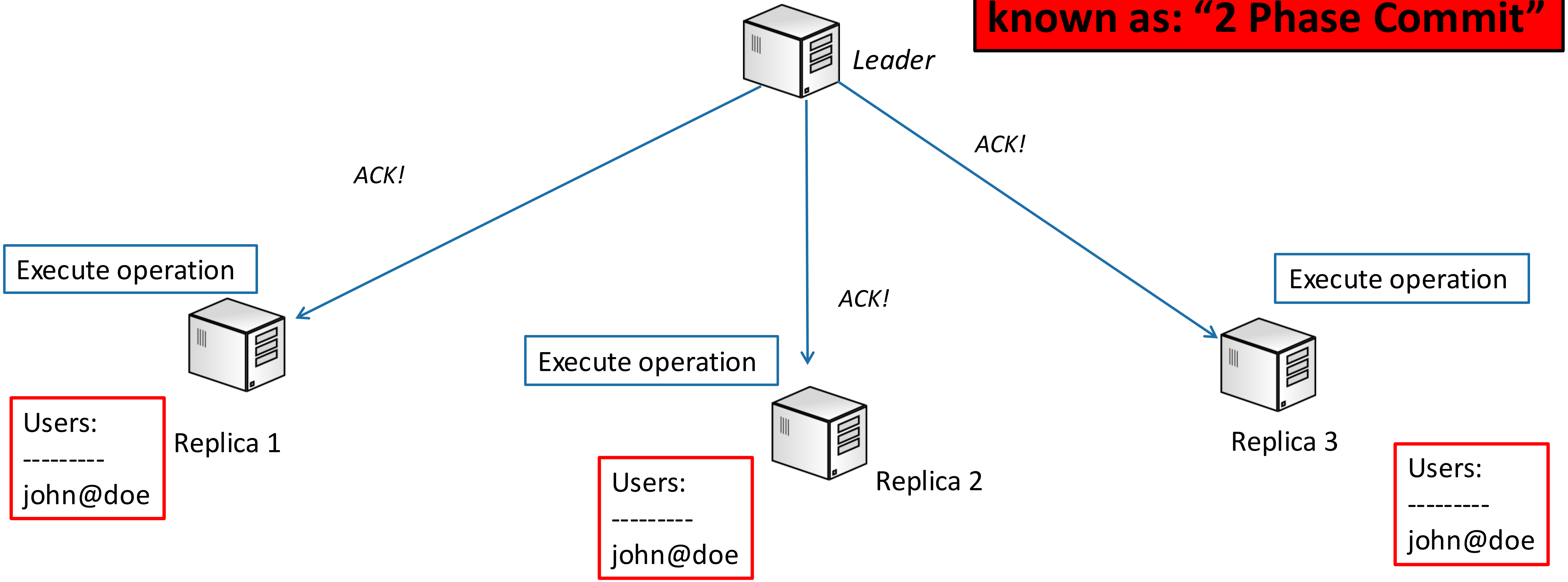
- In the absence of failures, the leader submits operations one-by-one and the replicas commit them in the submitted order



A Strawman Solution: Atomic Multicast

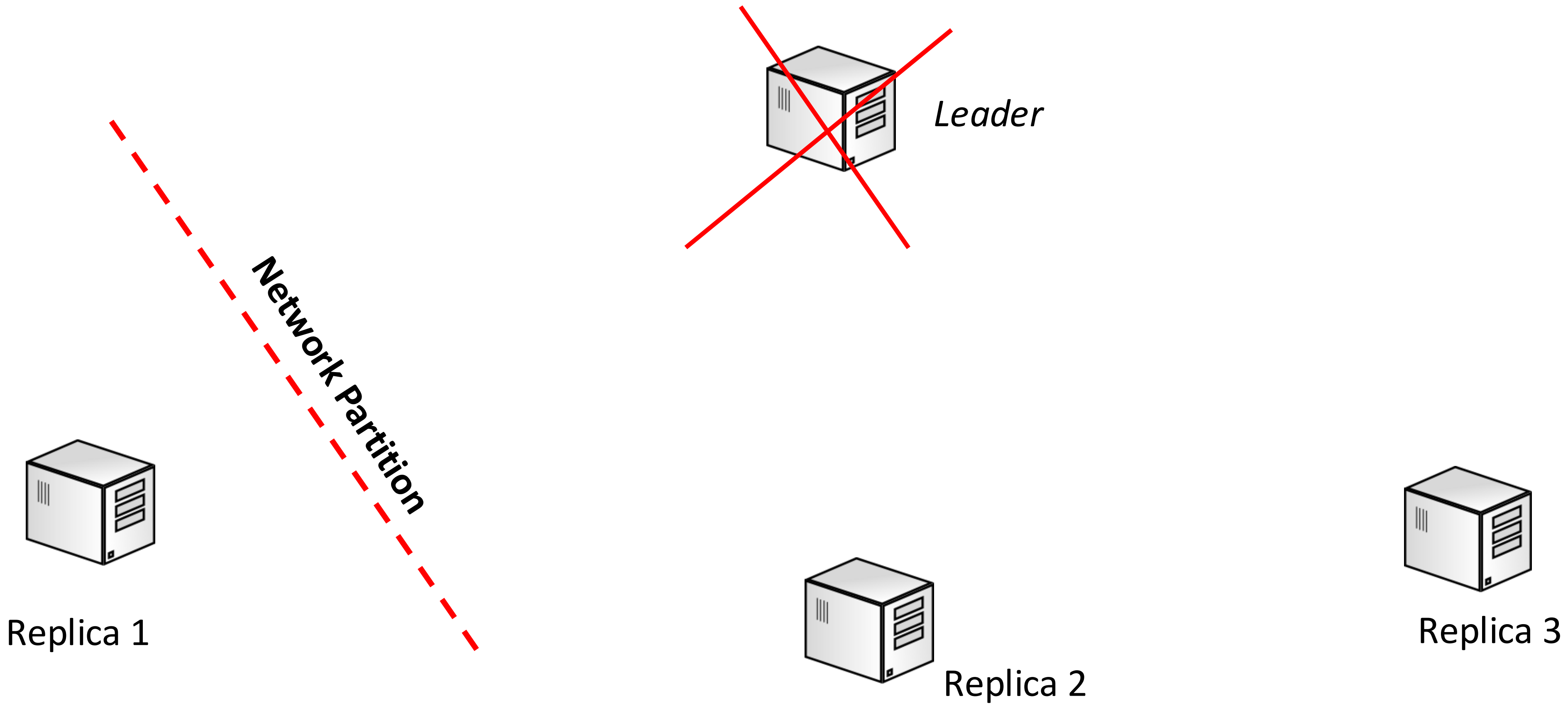
- In the absence of failures, the leader submits operations one-by-one and the replicas commit them in the submitted order

This two stage protocol is known as: "2 Phase Commit"



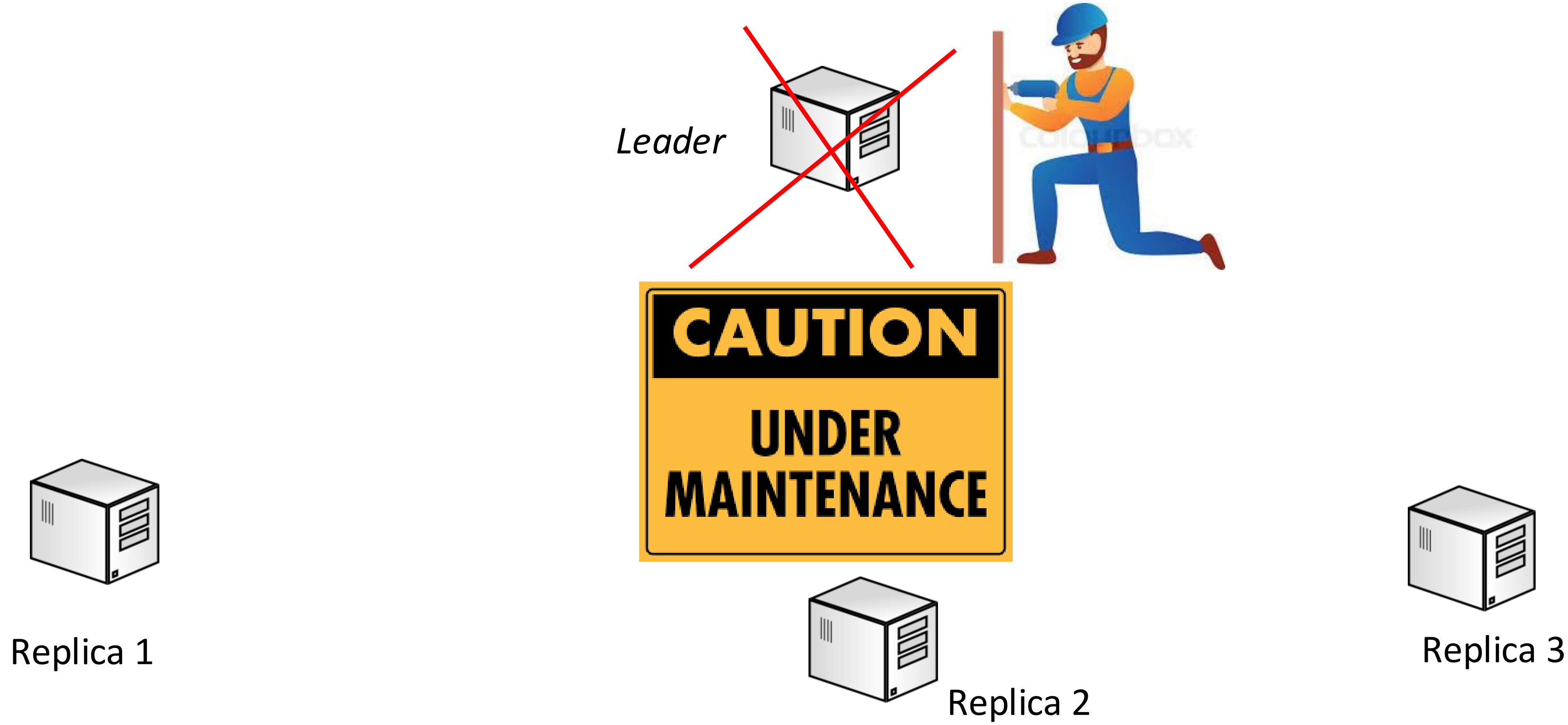
A Strawman Solution: Atomic Multicast

- What if the leader (sender) fails?
- What if the network gets partitioned?
  - One or more replicas are not reachable



A Strawman Solution: Atomic Multicast

- The replicated system comes to a halt when the leader fails!
  - The replicas can no longer safely process operations
  - We must wait until the leader (and the network) is repaired



State Machine Replication

- The strawman solution is blocked whenever the leader or the network fails
- We need a more **reliable** way for replicas to reach a consistent state **even under failures**

Consensus is what enables replicas to agree on the same sequence of operations across replicas, despite failures and without blocking indefinitely.

Team Paxos

Amazon ECS

cassandra

PaxosStore:
High-availability Storage Made Practical in WeChat

Jianjun Zheng[1] Qian Lin[1,2,*] Jiatao Xu[1] Cheng Wei[1]
Chuwei Zeng[1] Pingan Yang[1] Yunfan Zhang[1]
[1]Tencent Inc. [2]National University of Singapore
[1]{rockzheng, sunnyxu, dengoswei, eddyzeng, ypaapyyang, ifanzhang}@tencent.com
[2]linqian@comp.nus.edu.sg

Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency

Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, Leonidas Rigas

Microsoft

Clustrix

ceph

infinit

Megastore: Providing Scalable, Highly Available Storage for Interactive Services

Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, Vadim Yushprakh
Google, Inc.

Doozer

Large-scale cluster management at Google with Borg

Abhishek Verma[†] Luis Pedrosa[‡] Madhukar Korupolu
David Oppenheimer Eric Tune John Wilkes
Google Inc.

LogDevice

The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc.*

neo4j

Team Raft

LogCabin

TiKV

HashiCorp Consul

neo4j

RethinkDB

hazelcast

Atomix

etcd

CockroachDB

PAXOS

- When studying a formal protocol, we start by setting out our *assumptions* about the **system model** **(Week 1, Lecture 1)** in which the protocol operates

- This allows one to analyse whether the protocol's properties are correct if/when the stated assumptions are true

PAXOS – System Model

- **Timing, Synchrony and Networking:** Partially synchronous and fair loss links
  - Messages may be lost, re-ordered, and may take arbitrarily long to deliver but are eventually delivered (after re-trying enough times)
  - Nodes operate at **arbitrary speeds (can be very slow to respond)**
- **Only crash failures:** crash-recovery
  - Nodes can fail and may sebsequently recover
  - Nodes **do not attempt to subvert** the protocol, and messages are never corrupted
    - This means Paxos does NOT consider Byzantine failures
    - We will look at other consensus protocols that consider Byzantine failures (See the upcoming PBFT and Blockchain lectures)

PAXOS – System Model

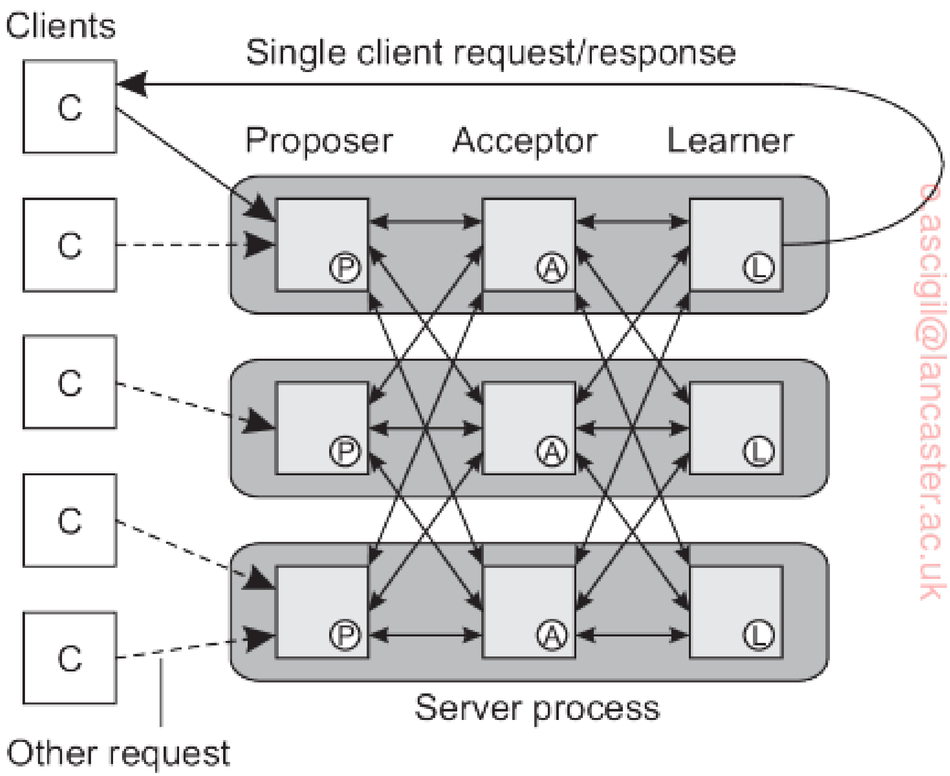
- One of the key problems with the resulting system model based on these assumptions is that ***failure* is indistinguishable from *latency***
  - This is true in all real distributed systems operating under asynchronous or partial synchronous conditions
  - The protocol therefore needs to deal with the “failure detector" making a mistake

PAXOS Protocol

- Paxos uses three different roles
  - **Proposers**: initiate an agreement phase
  - **Acceptors**: participate in agreement
  - **Learners**: receive the outcome of an agreement
- A single proposer, acceptor, and learner form

a single physical node

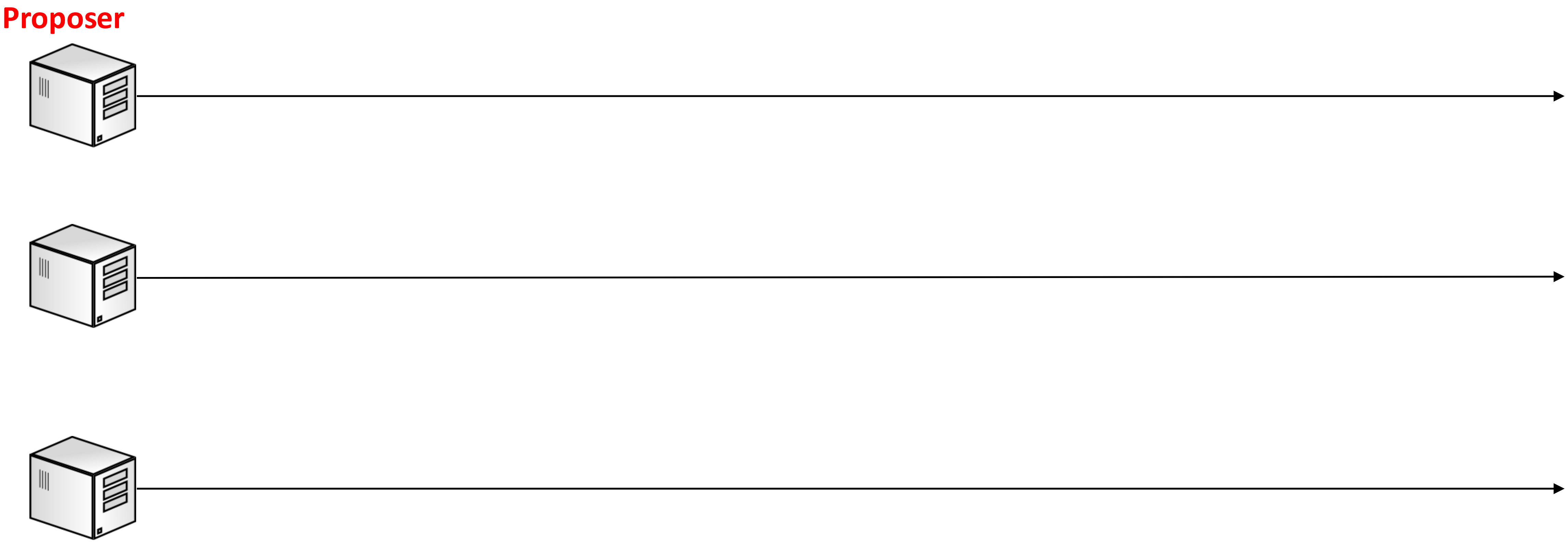
- At every node, each state transition is written to persistent storage before taking the next step



PAXOS Protocol

- Paxos uses two different phases:
  - Prepare & promise phase
    - This stage *selects a proposal number (a logical timestamp)* for the consensus round (this number is also referred to as round number)
    - Uses **prepare** and **promise** messages
  - Accept & Learn phase
    - This stage agrees upon an operation for a chosen proposal number
    - Uses **accept** and **learn** messages

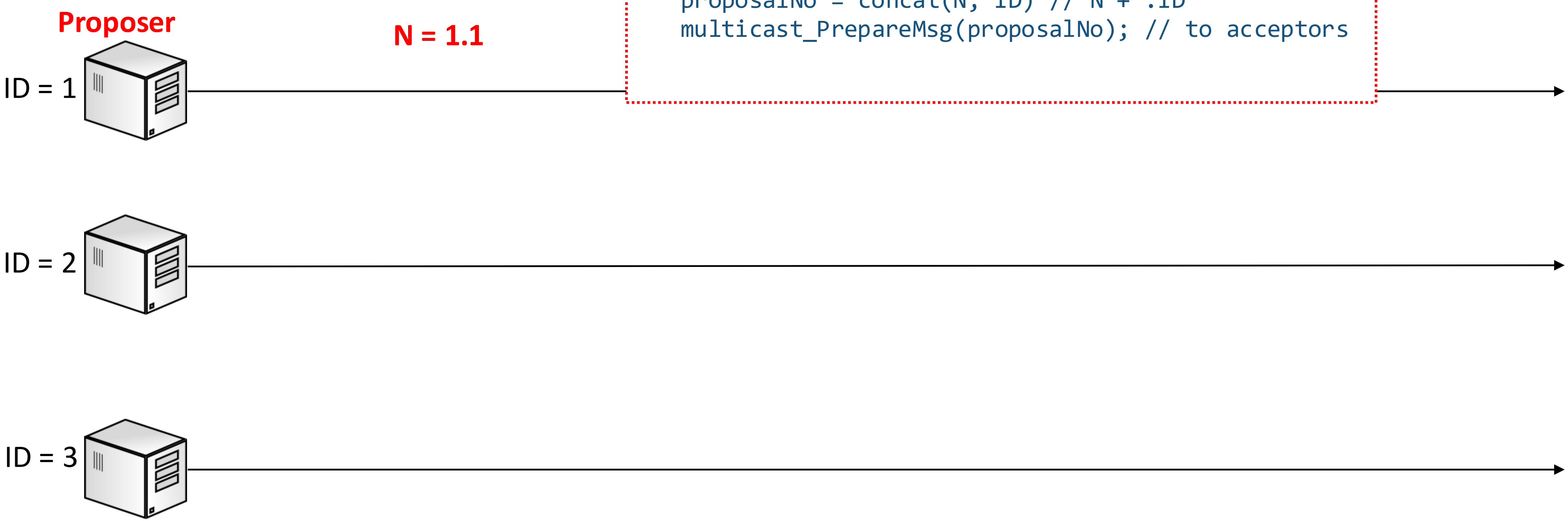
Paxos Protocol



Paxos Protocol

```
int N // largest proposal number seen so far
prepare():
   N = N + 1;
   proposalNo = concat(N, ID) // N + .ID
   multicast_PrepareMsg(proposalNo); // to acceptors
```

Create a proposal number!

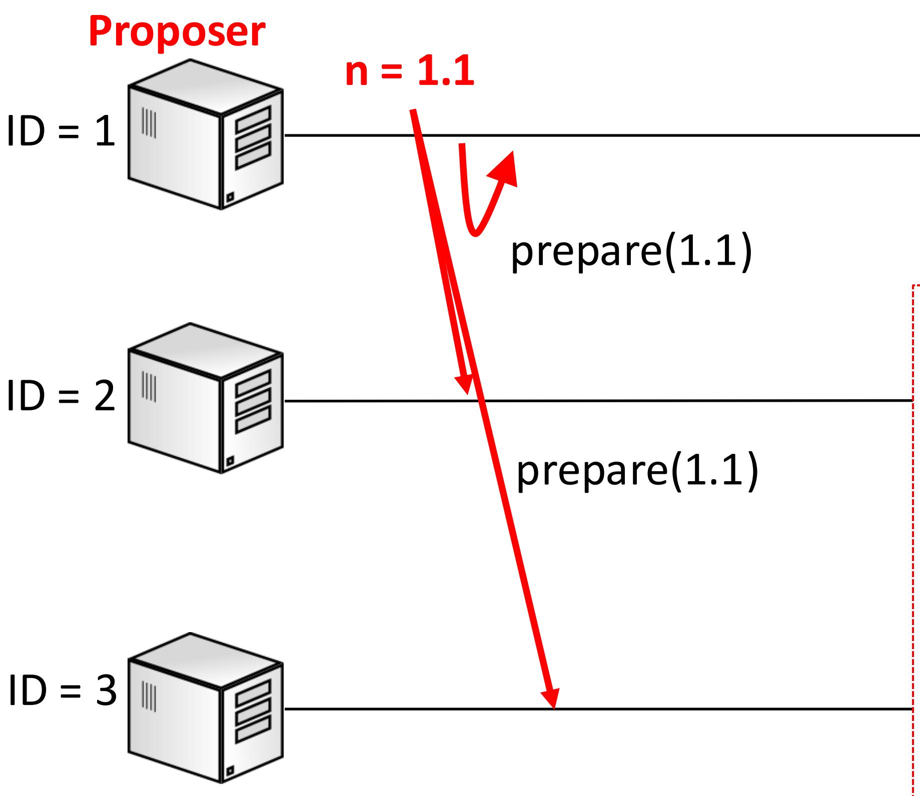


A proposal number has to be:

1) greater than any previous proposal number used in the group
2) unique so that two proposers cannot propose the same number (a greater than relationship must exist between proposal numbers)

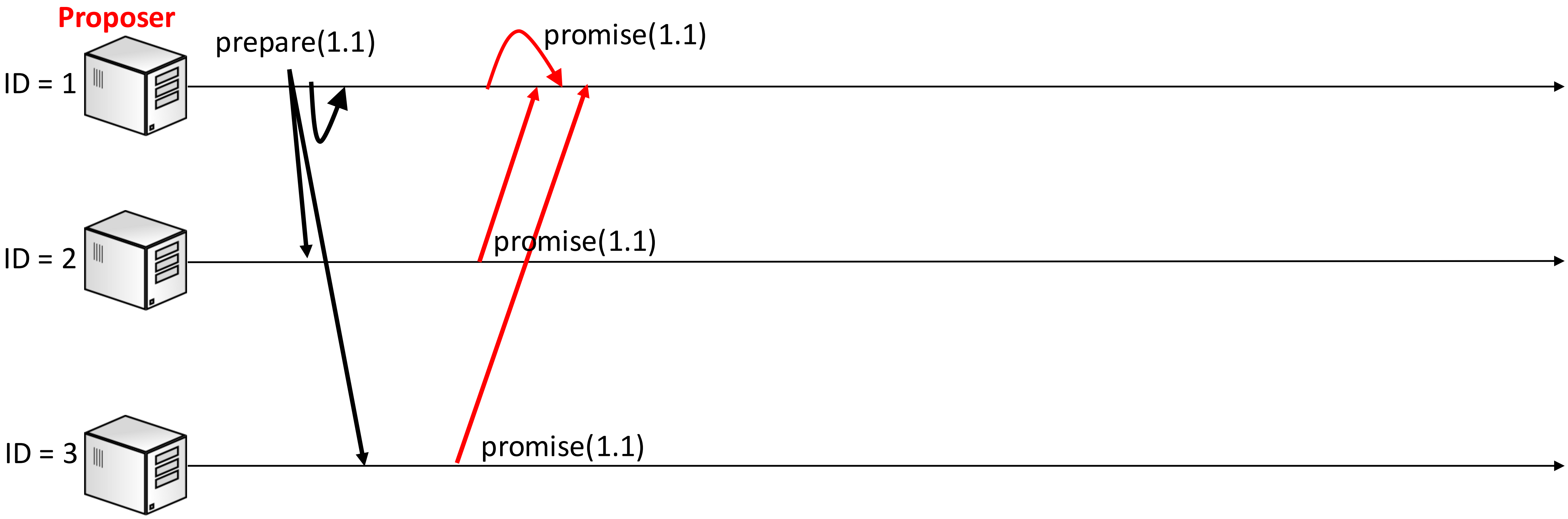
Paxos Protocol

Send prepare messages



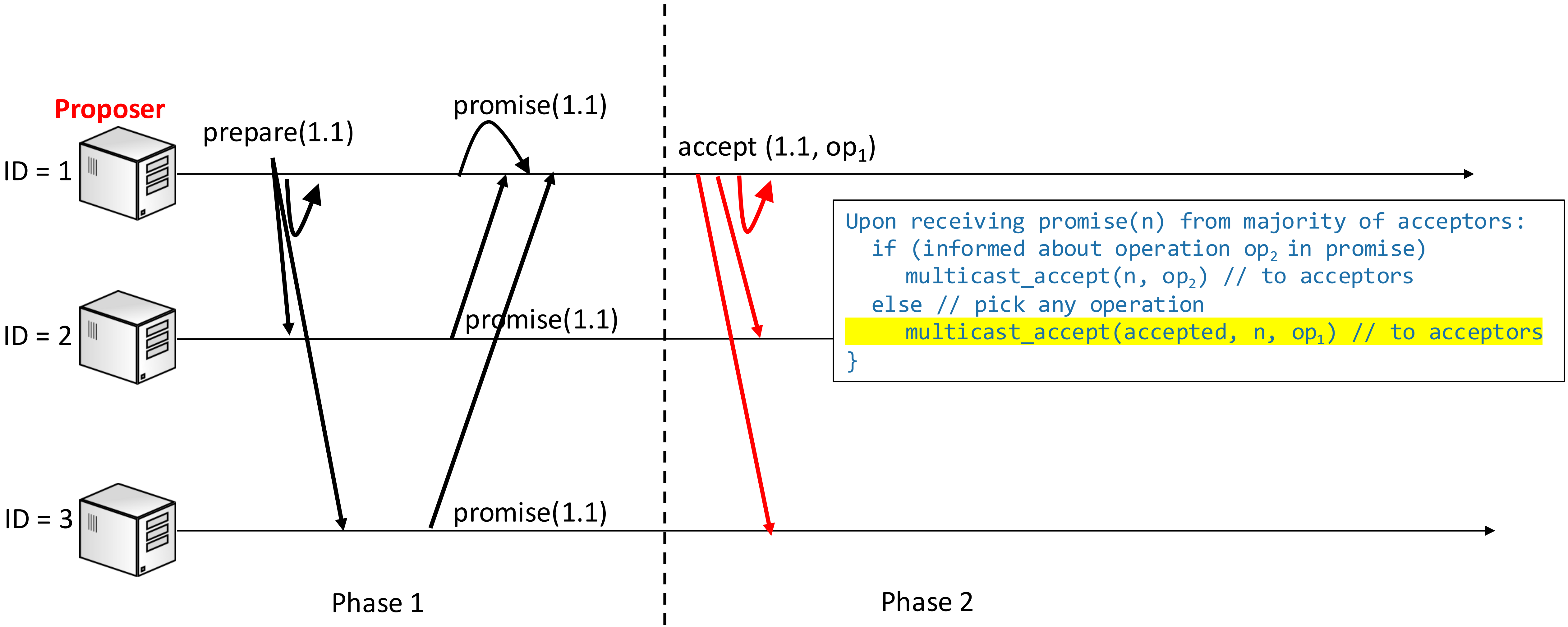

```
promisedN = 0; // the largest proposal number ever promised
uponReceivingPrepare(n):
  if (n > promisedN)
      promisedN=n
      write_state(promised to promisedN) //write to disk
      if (previously accepted operation op₂)
          reply_promise(n, op₂) //to proposer
      else
          reply_promise(n) //to proposer
  else
     do nothing
```

Paxos Protocol



All 3 acceptors send promise() messages

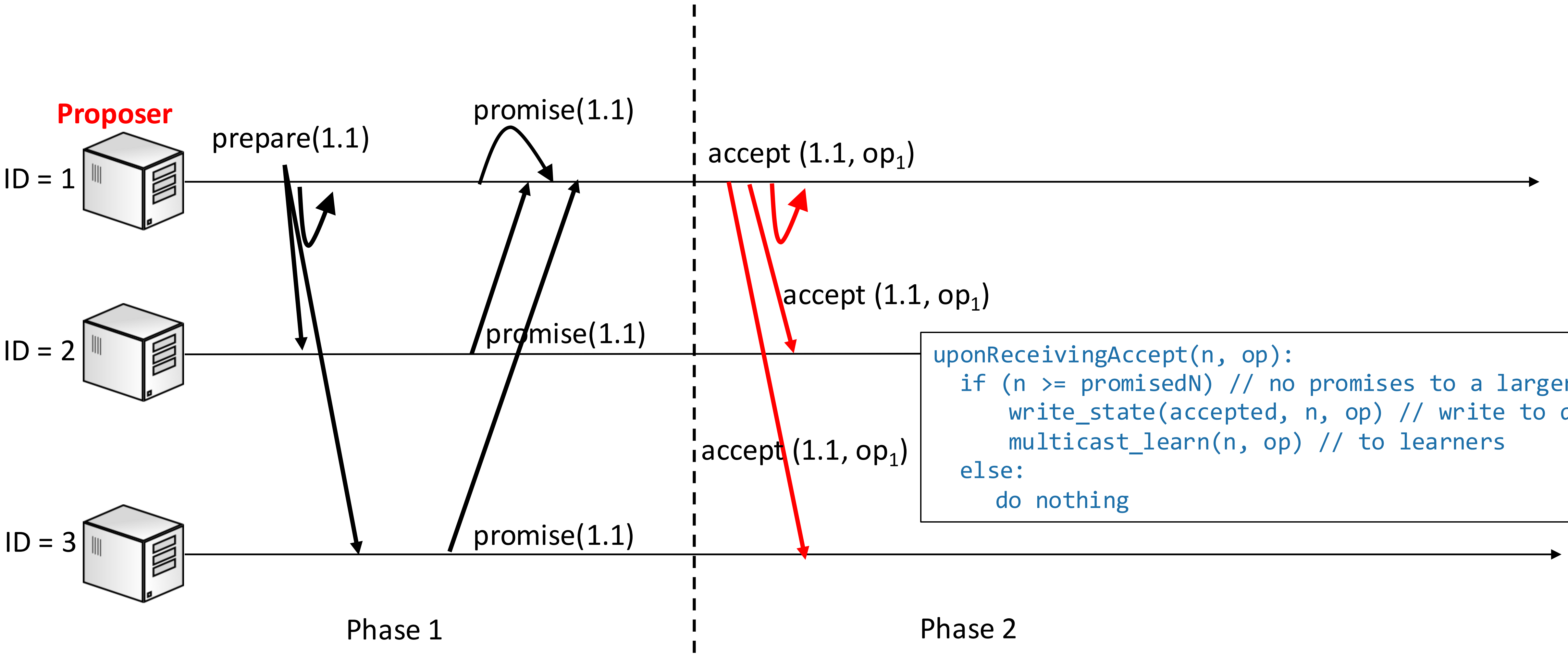
Paxos Protocol




```
Upon receiving promise(n) from majority of acceptors:
  if (informed about operation op2 in promise)
    multicast_accept(n, op2) // to acceptors
  else // pick any operation
    multicast_accept(accepted, n, op1) // to acceptors
}
```

Now an operation is proposed in an accept () message

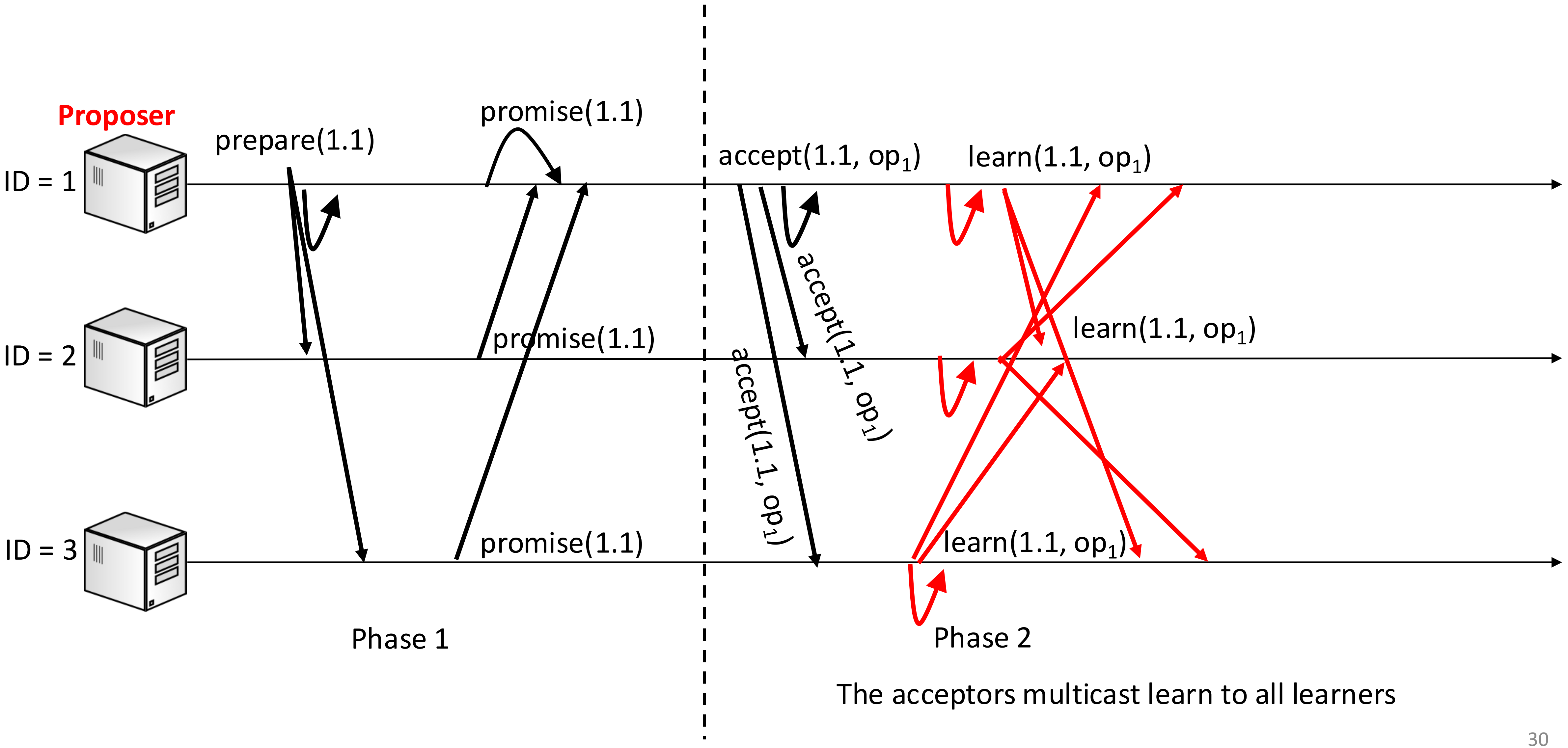
Paxos Protocol




```
uponReceivingAccept(n, op):
  if (n >= promisedN) // no promises to a larger n
     write_state(accepted, n, op) // write to disk
     multicast_learn(n, op) // to learners
  else:
     do nothing
```

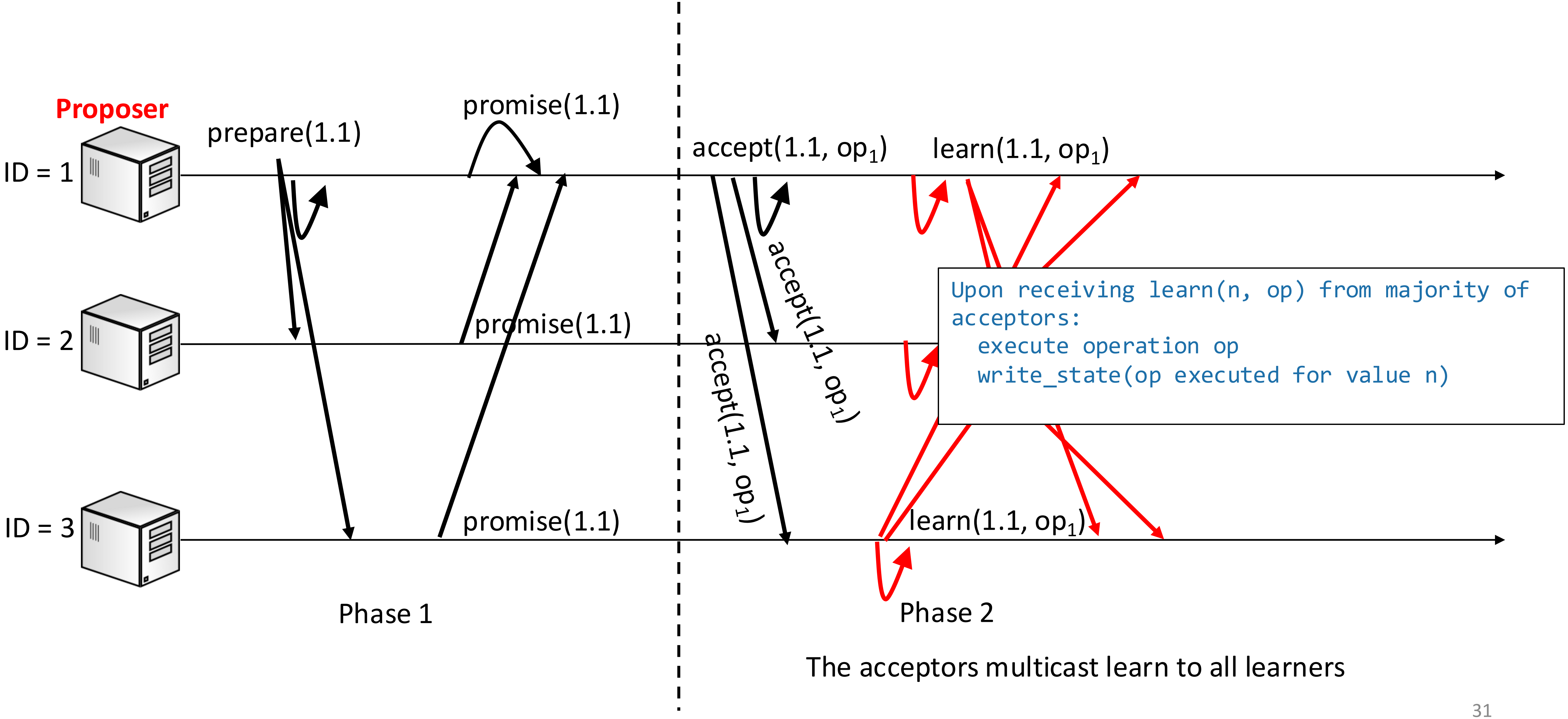
Now the proposer proposes a value *v* in an accept_request()

Paxos Protocol



The acceptors multicast learn to all learners

Paxos Protocol

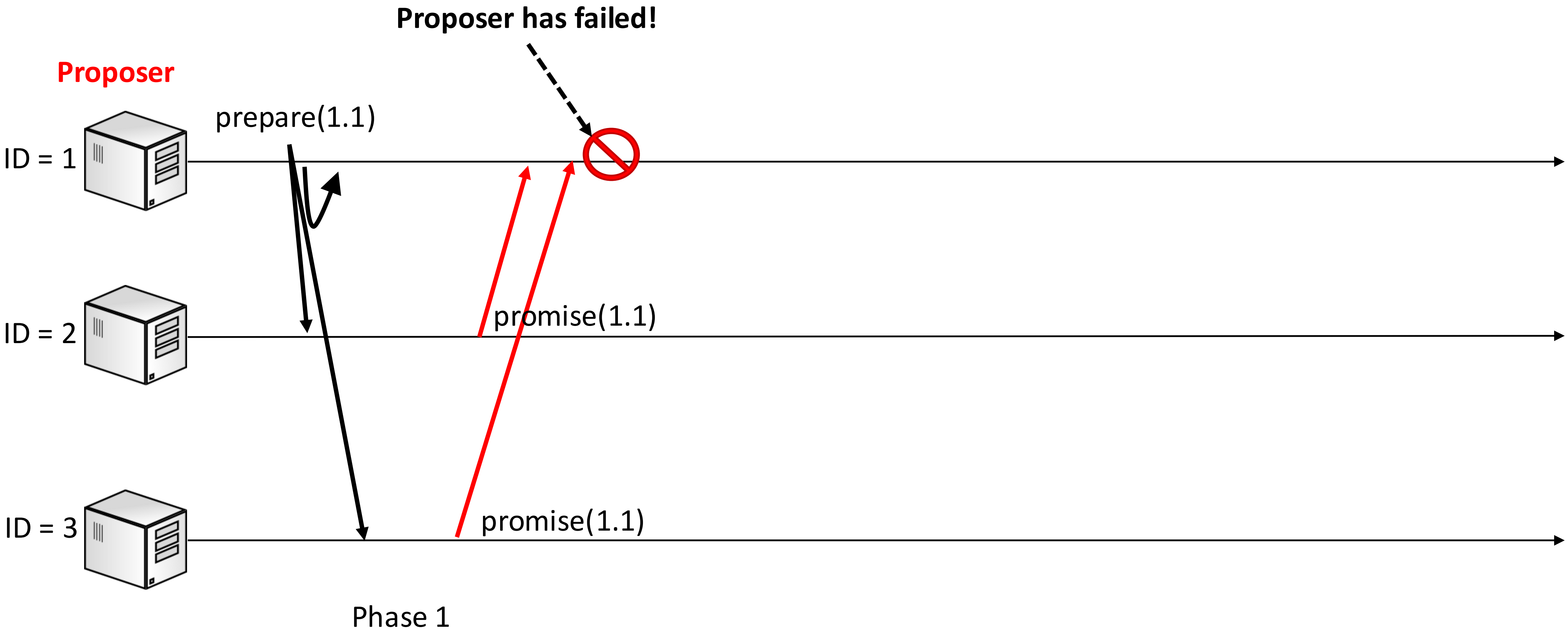


The acceptors multicast learn to all learners

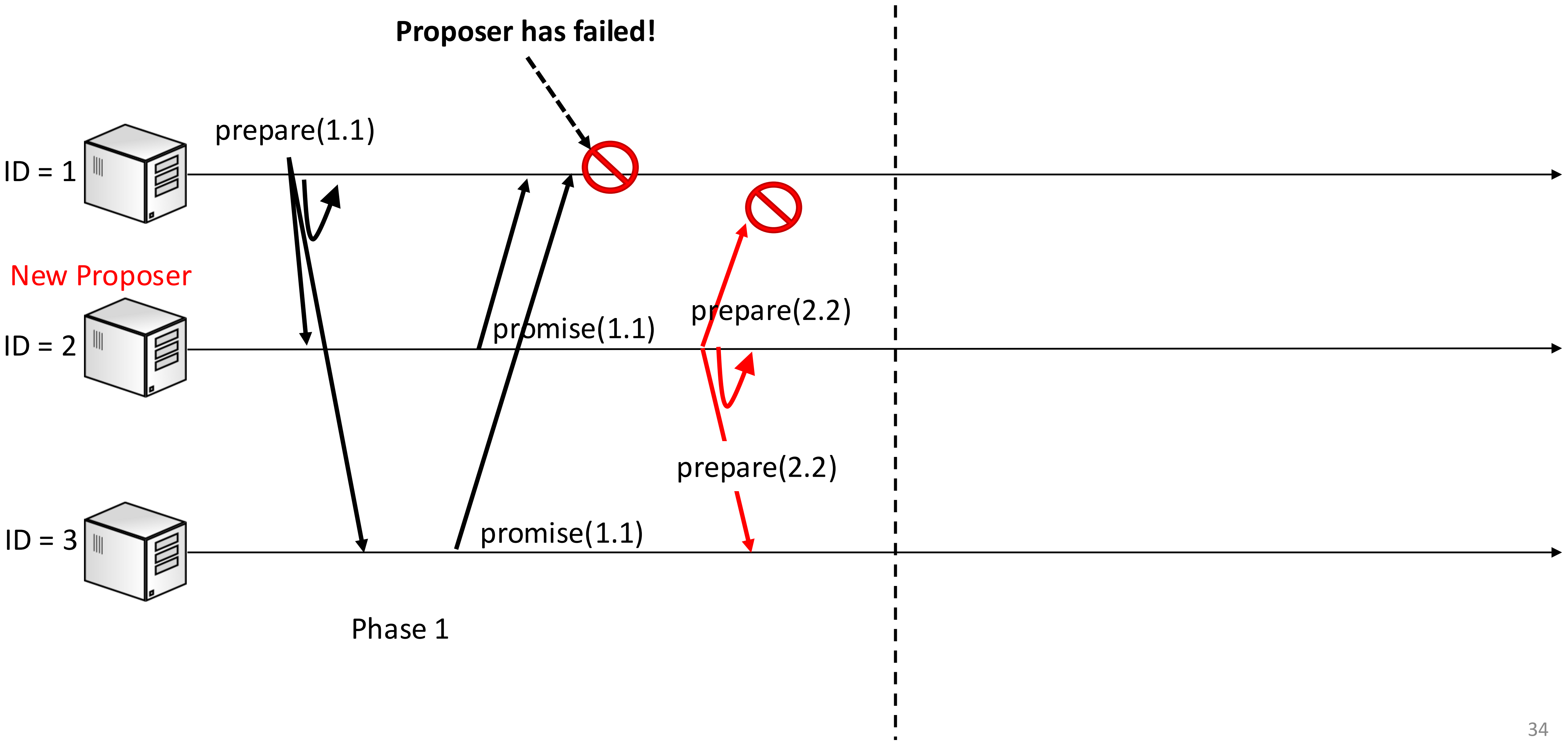
Paxos Protocol

- In the absence of failures, all learners eventually execute the operation
- We will now go through a couple of failure scenarios
  - Scenario 1: Proposer fails after sending prepare
  - Scenario 2: Proposer fails after sending accept
  - Scenario 3: Acceptor fails after receiving accept
  - Scenario 4: two proposers propose at the same time

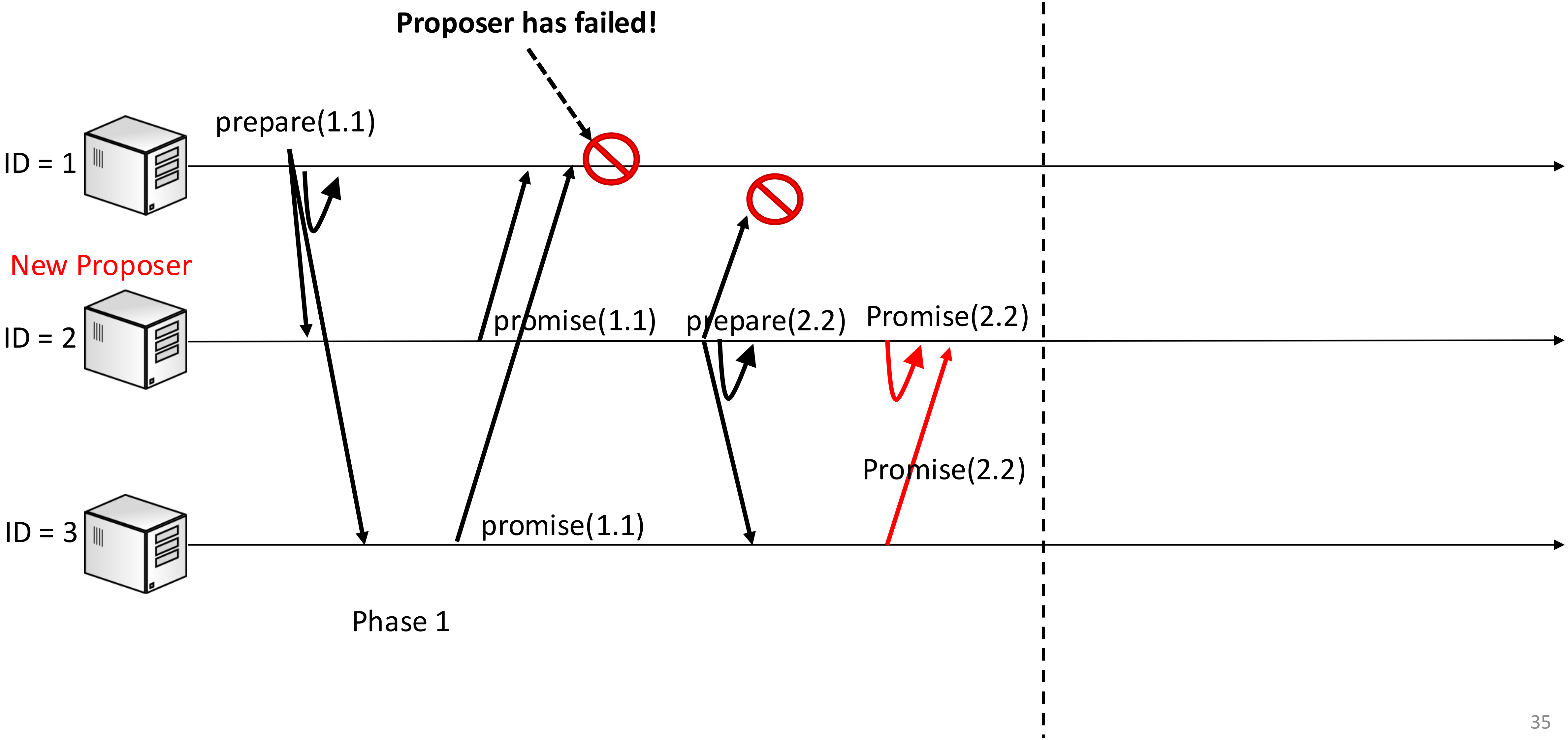
Scenario 1: Proposer fails after prepare



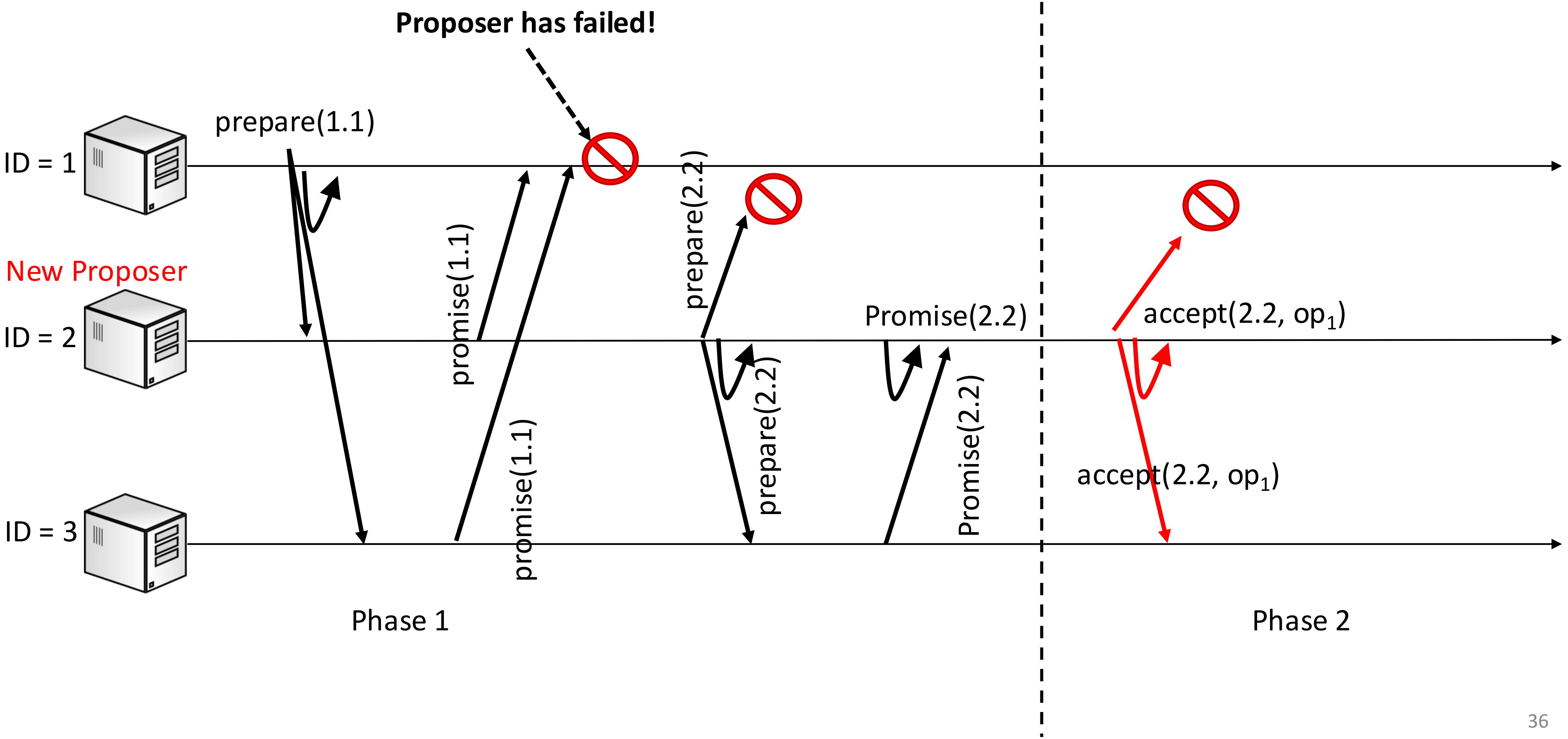
Scenario 1: Proposer fails after prepare



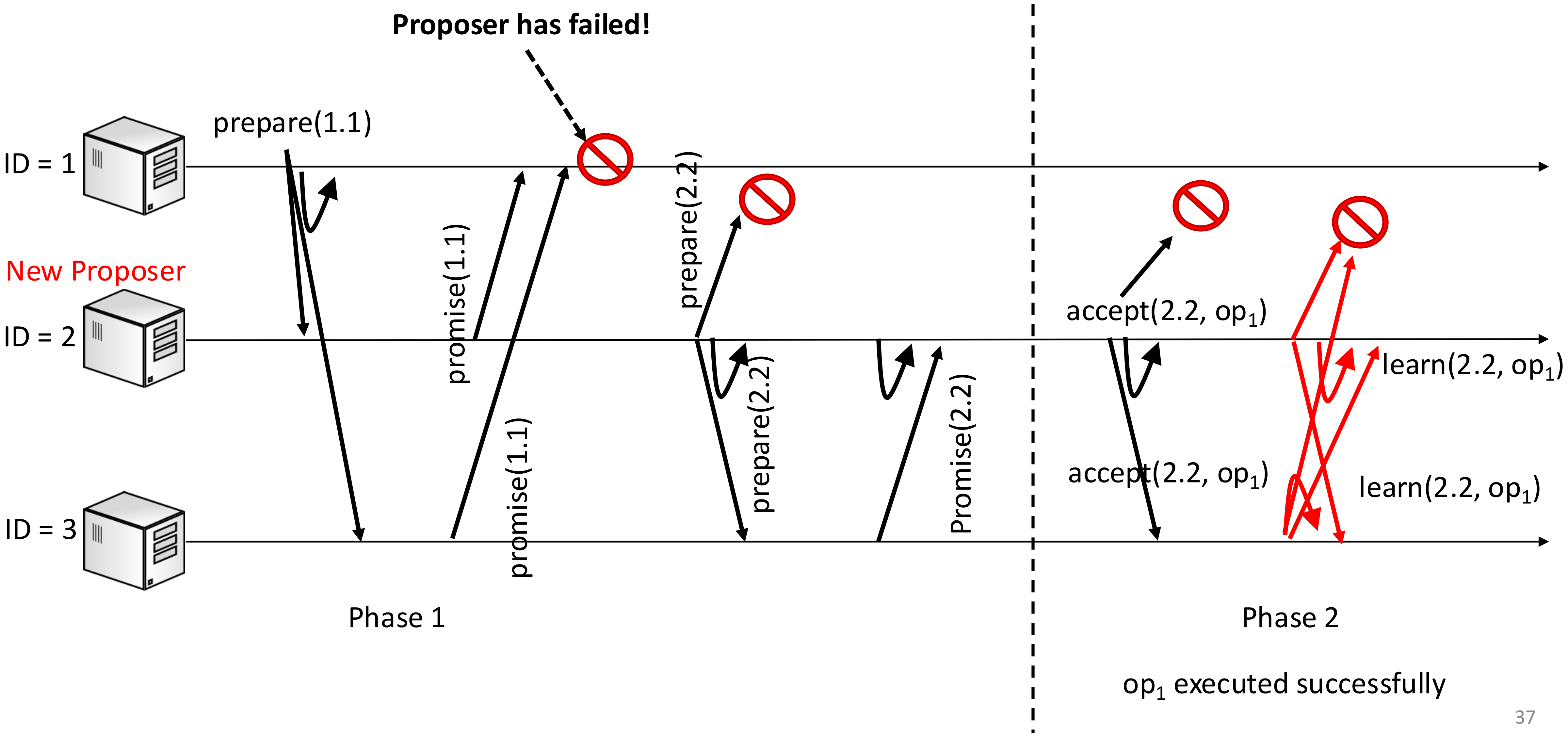
Scenario 1: Proposer fails after prepare



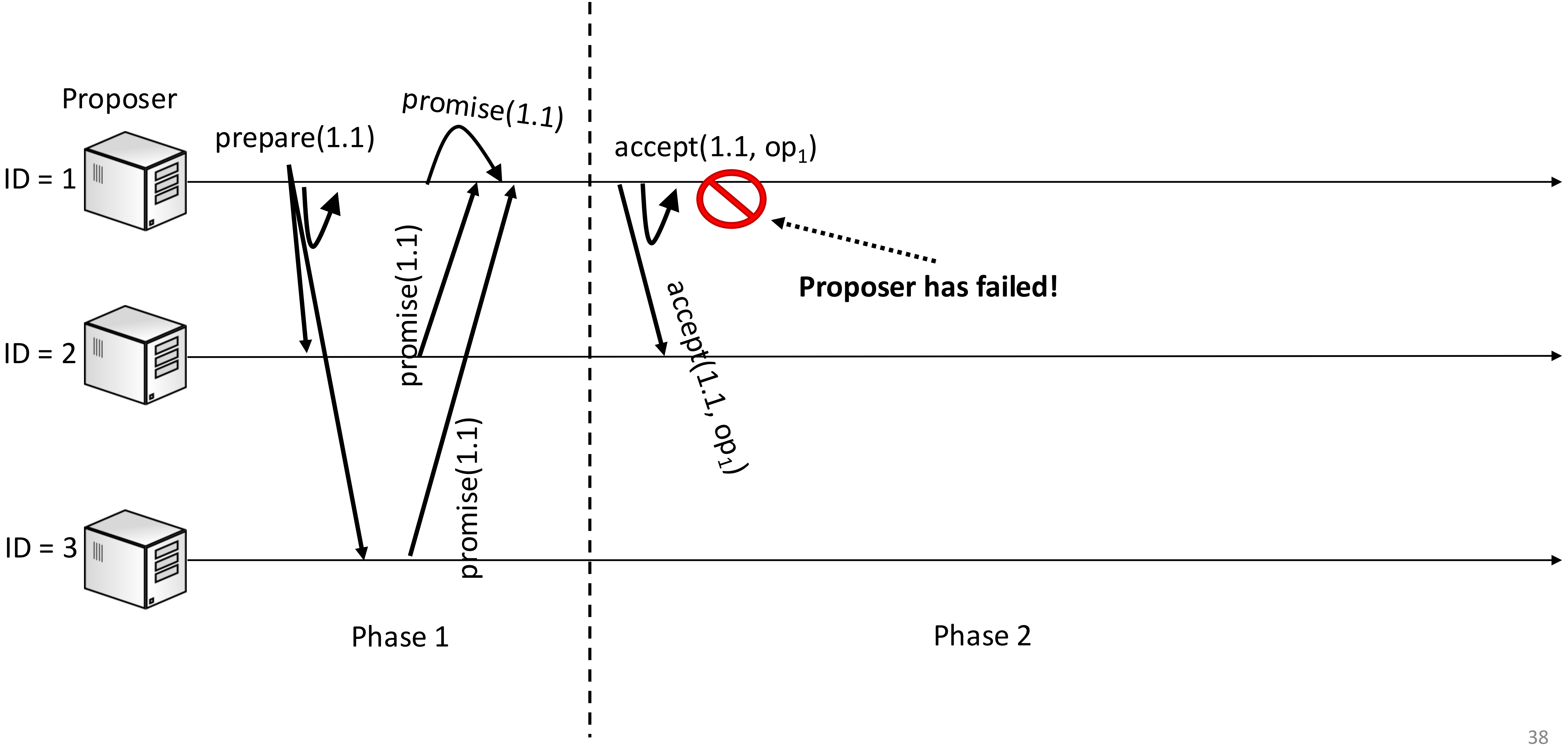
Scenario 1: Proposer fails after prepare



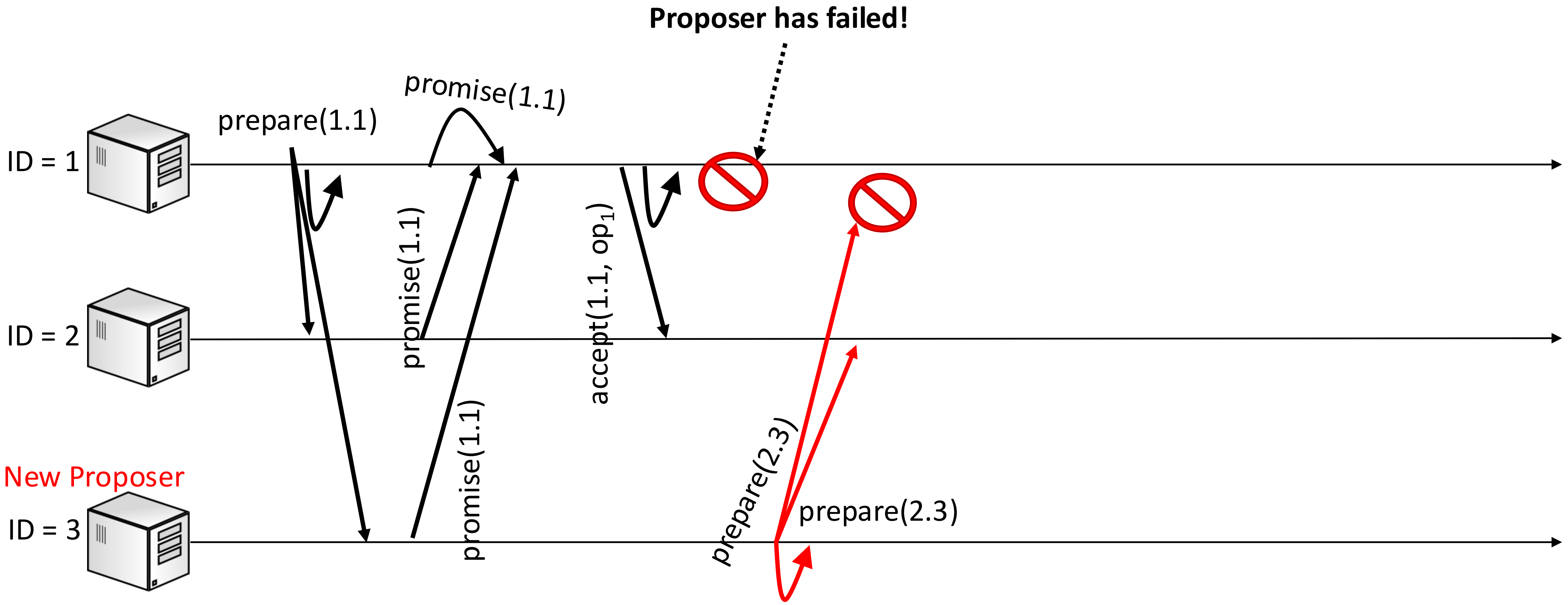
Scenario 1: Proposer fails after prepare



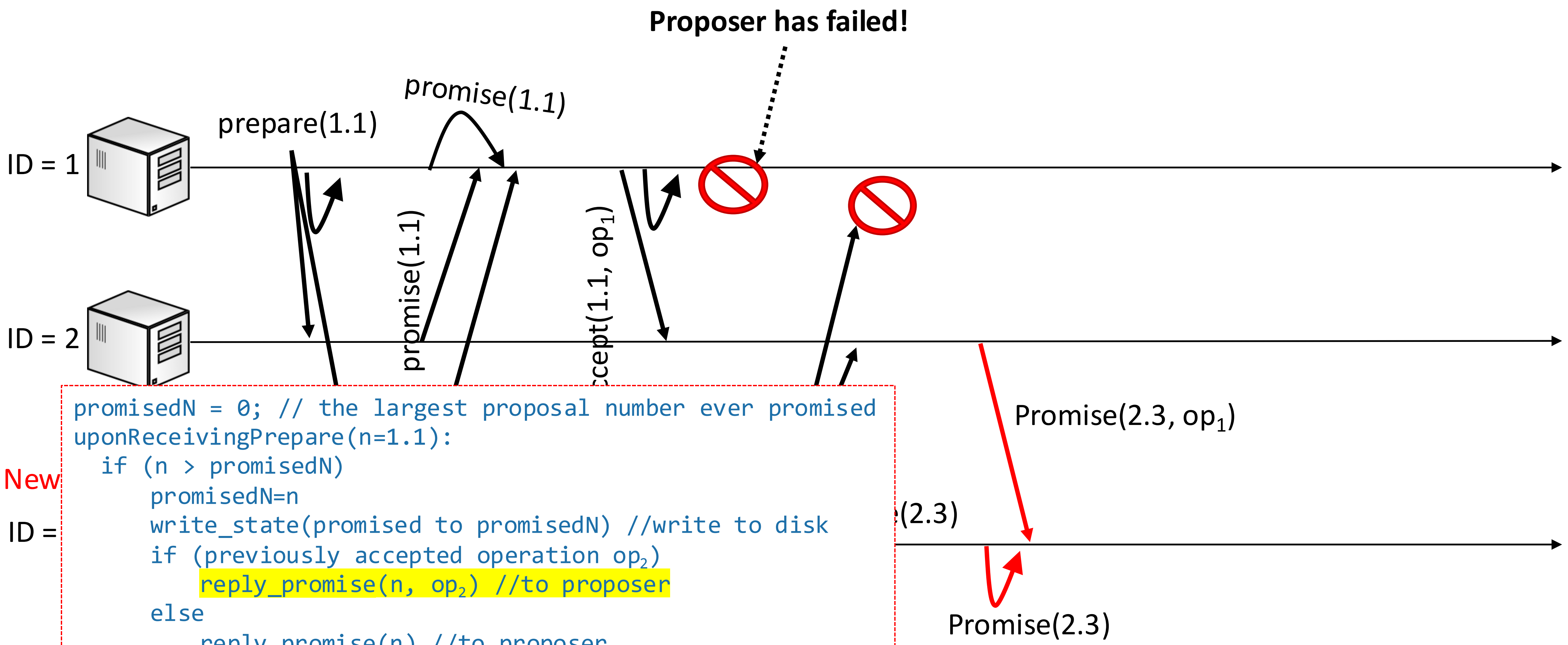
Scenario 2: Proposer fails after accept



Scenario 2: Proposer fails after accept

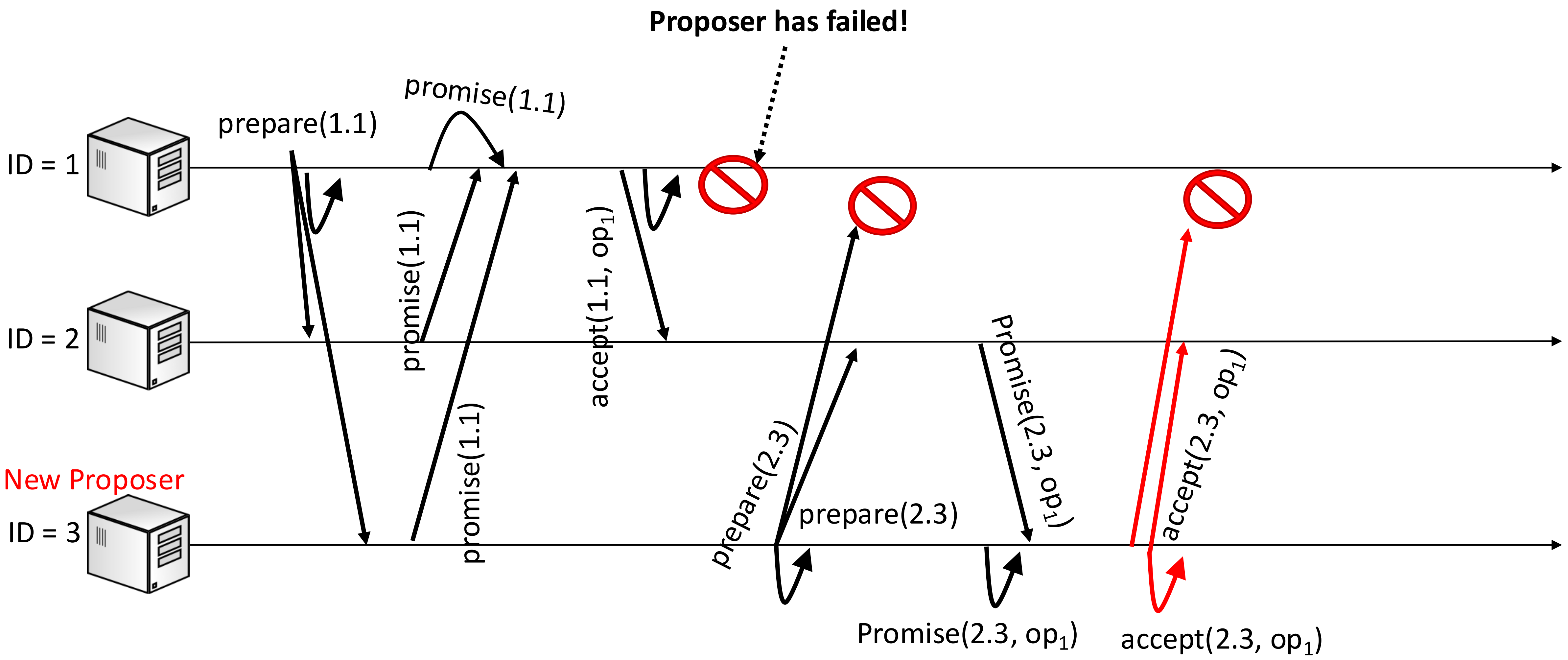


Scenario 2: Proposer fails after accept

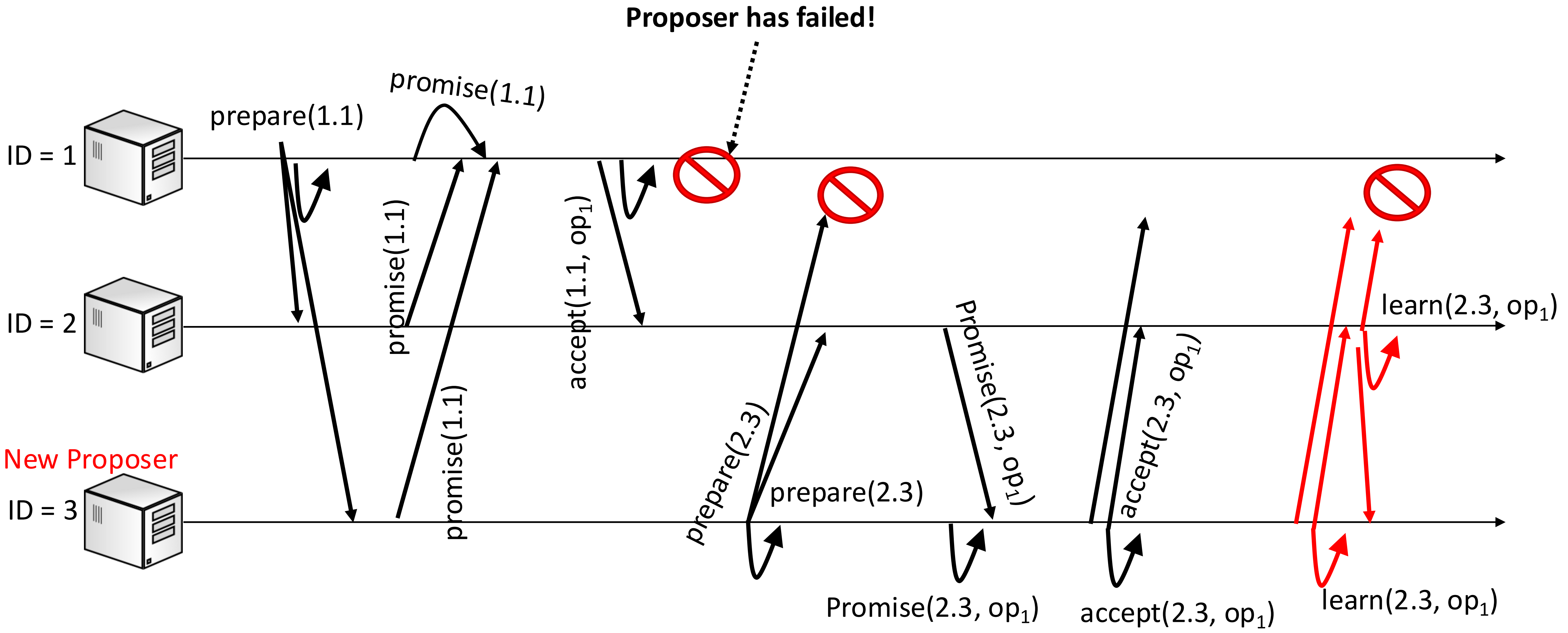



```
promisedN = 0; // the largest proposal number ever promised
uponReceivingPrepare(n=1.1):
  if (n > promisedN)
      promisedN=n
      write_state(promised to promisedN) //write to disk
      if (previously accepted operation op2)
          reply_promise(n, op2) //to proposer
      else
          reply_promise(n) //to proposer
  else
     do nothing
```

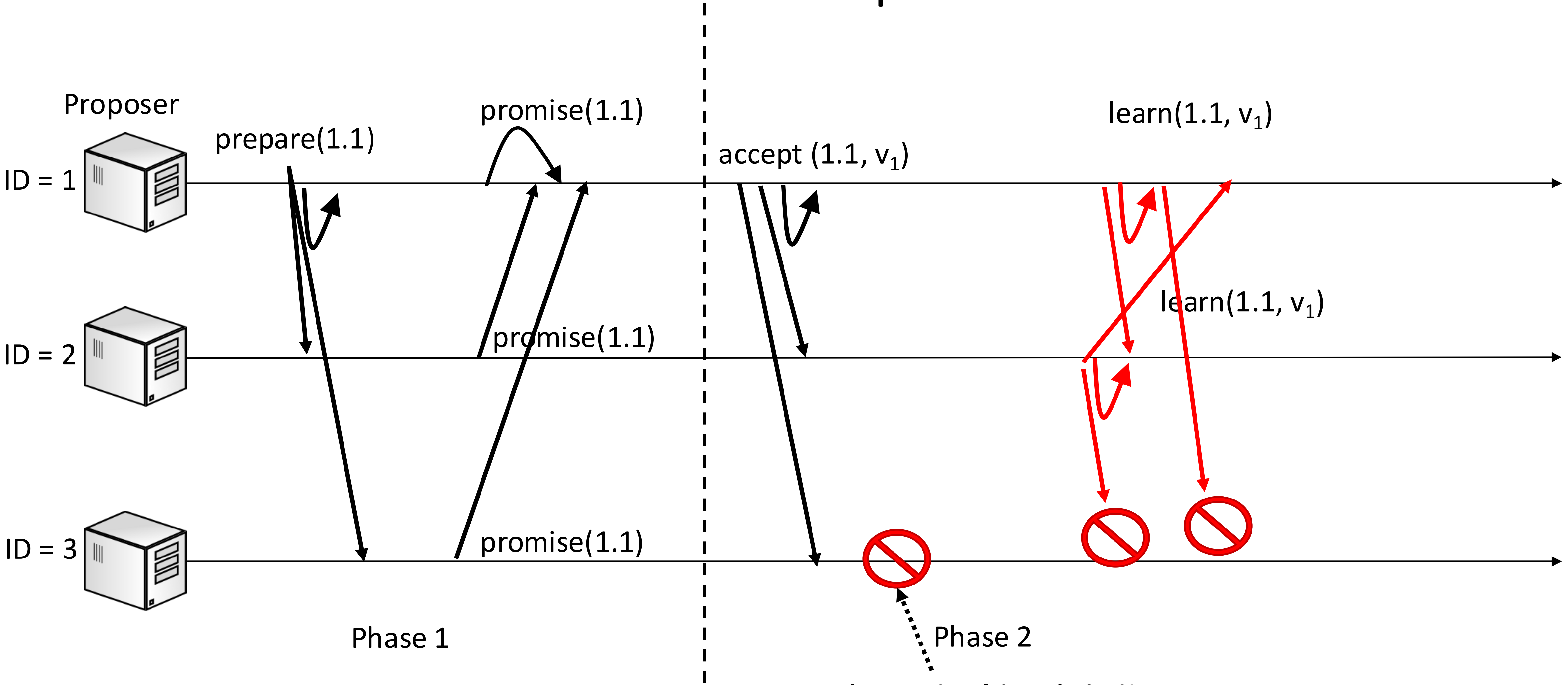
Scenario 2: Proposer fails after accept



Scenario 2: Proposer fails after accept

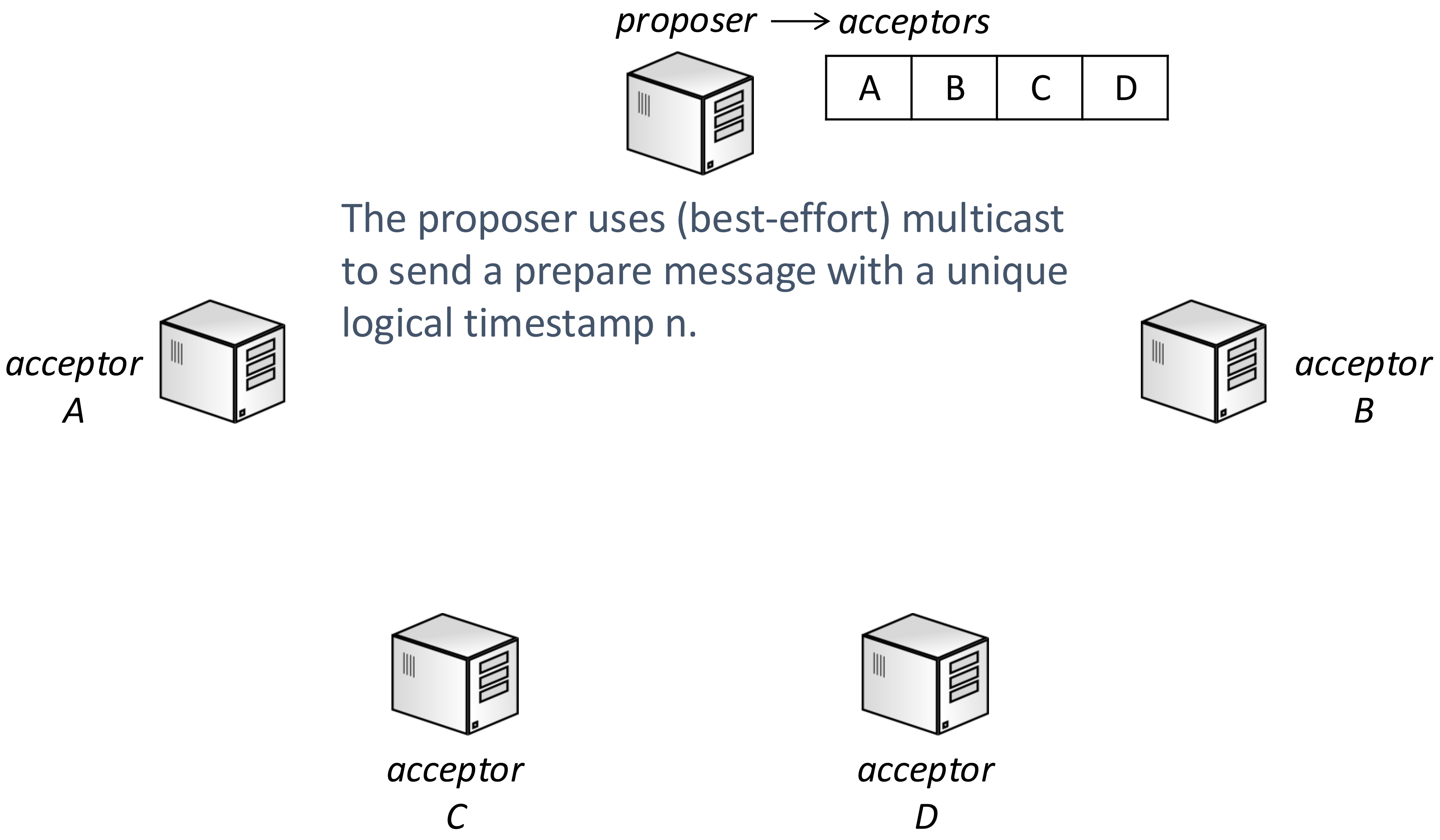


Scenario 3: Acceptor fails after receiving accept



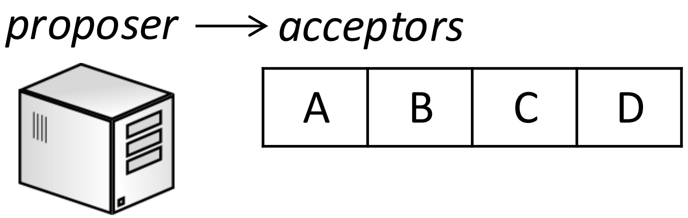
Acceptor (at Node 3) has failed!

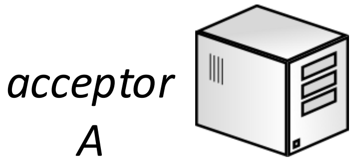
Protocol – Phase 1 – sending prepare



The proposer uses (best-effort) multicast to send a prepare message with a unique logical timestamp n.

Protocol – Phase 1 – responding with promise





Each acceptor compares ***proposal number n*** with any *previous proposal they received (and promised)*. If *n* is higher, a *promise* is sent to the proposer, indicating agreed participation.

- if an acceptor has accepted a previous proposal, the operation from this proposal is included in the promise.

Protocol – Phase 2 proposer sending accept

proposer

If the proposer receives promises from **a majority of acceptors** (half + 1), it sends an *accept(**n, op**)* to the acceptors.

- *op* is set **either** to an operation chosen by the proposer **or** to the highest value reported by any acceptor via *promise()*, if an acceptor has previously accepted an operation from another proposer

this allows the proposer to decide whether or not there are any unfinished consensus rounds

acceptor

acceptor

acceptor

acceptor

Protocol – Phase 2 receiving accept

proposer

acceptor

An acceptor that receives an *accept(n, op)* must accept it unless it has already sent a *promise()* to a proposer with a higher *n* value. If so, the acceptor notes the value *op*, (best-effort) multicasts the <n, op> to all learners.

acceptor

acceptor

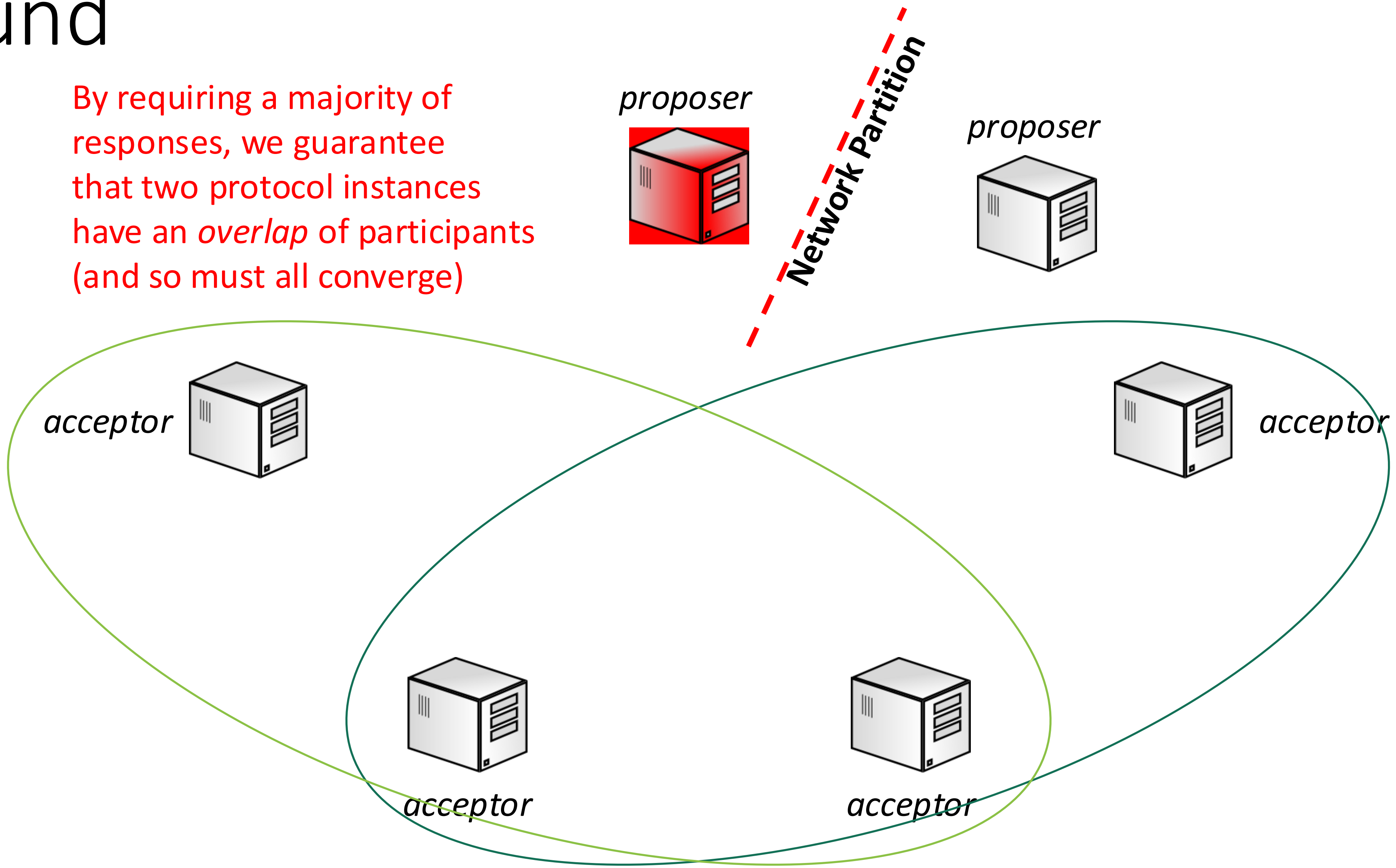
acceptor

Protocol – Phase 2 - learn

Finally, the learner must receive a *learn(n, op)* from a **majority of acceptors** (half+1) to execute the operation. If this doesn't happen, a new protocol round is initiated

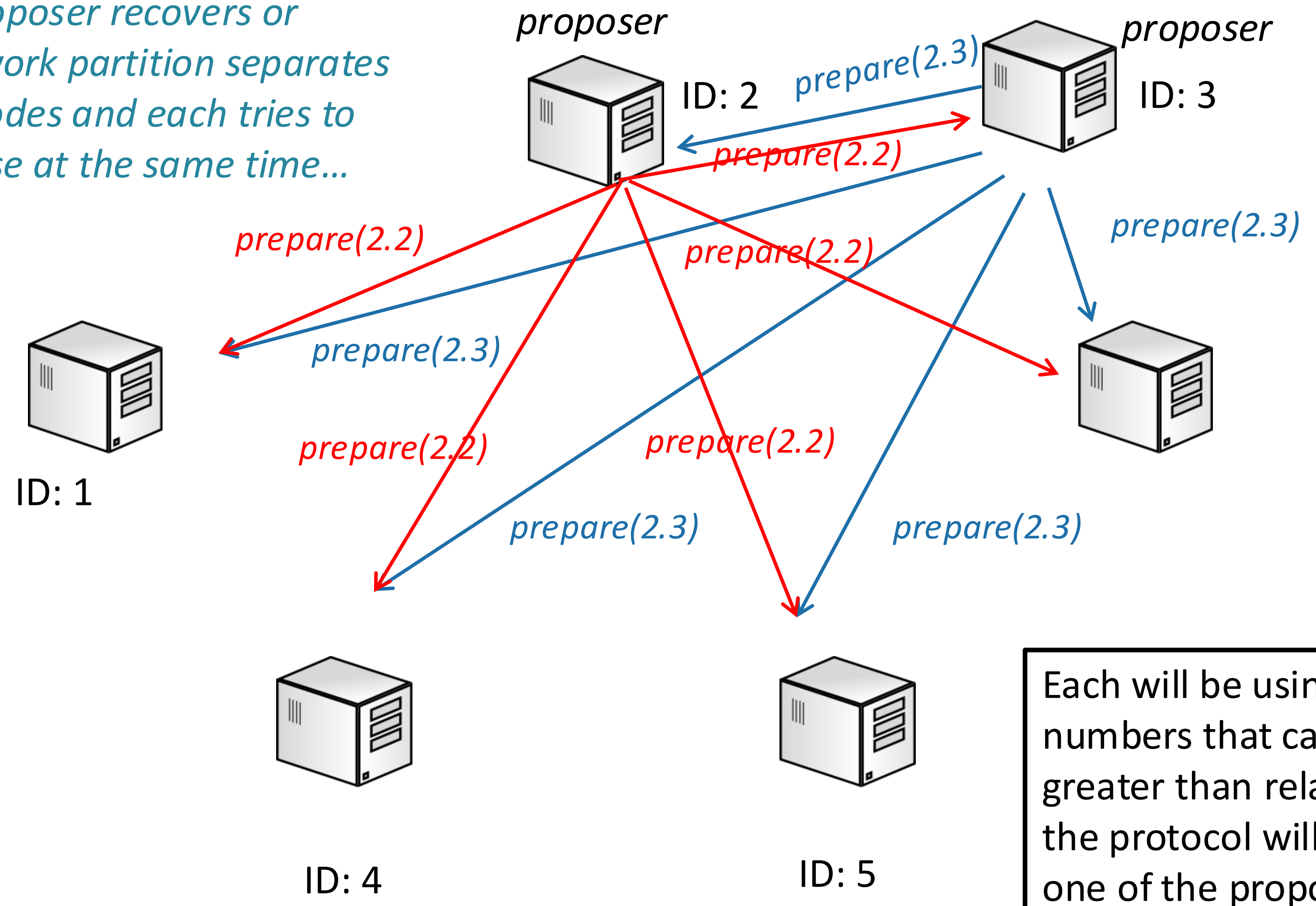
Scenario 4: Multiple proposers in the same round

By requiring a majority of responses, we guarantee that two protocol instances have an *overlap* of participants (and so must all converge)



Scenario 4: Multiple proposers in the same round

old proposer recovers or
a network partition separates
two nodes and each tries to
propose at the same time…



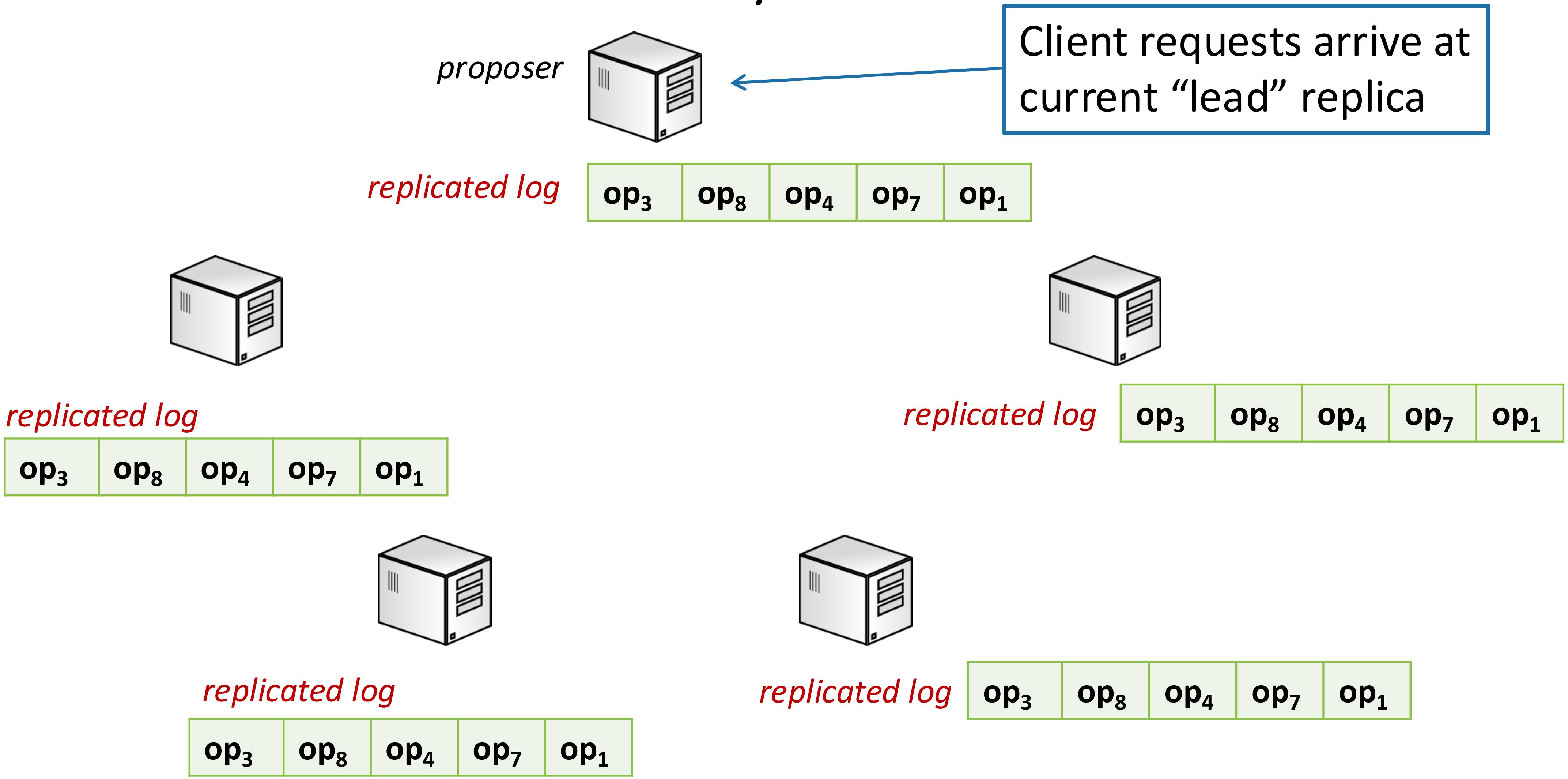
Each will be using unique proposal numbers that can be compared with greater than relationship. Therefore the protocol will converge to accepting one of the proposed values by the majority of acceptors.

Paxos: how it’s really used…

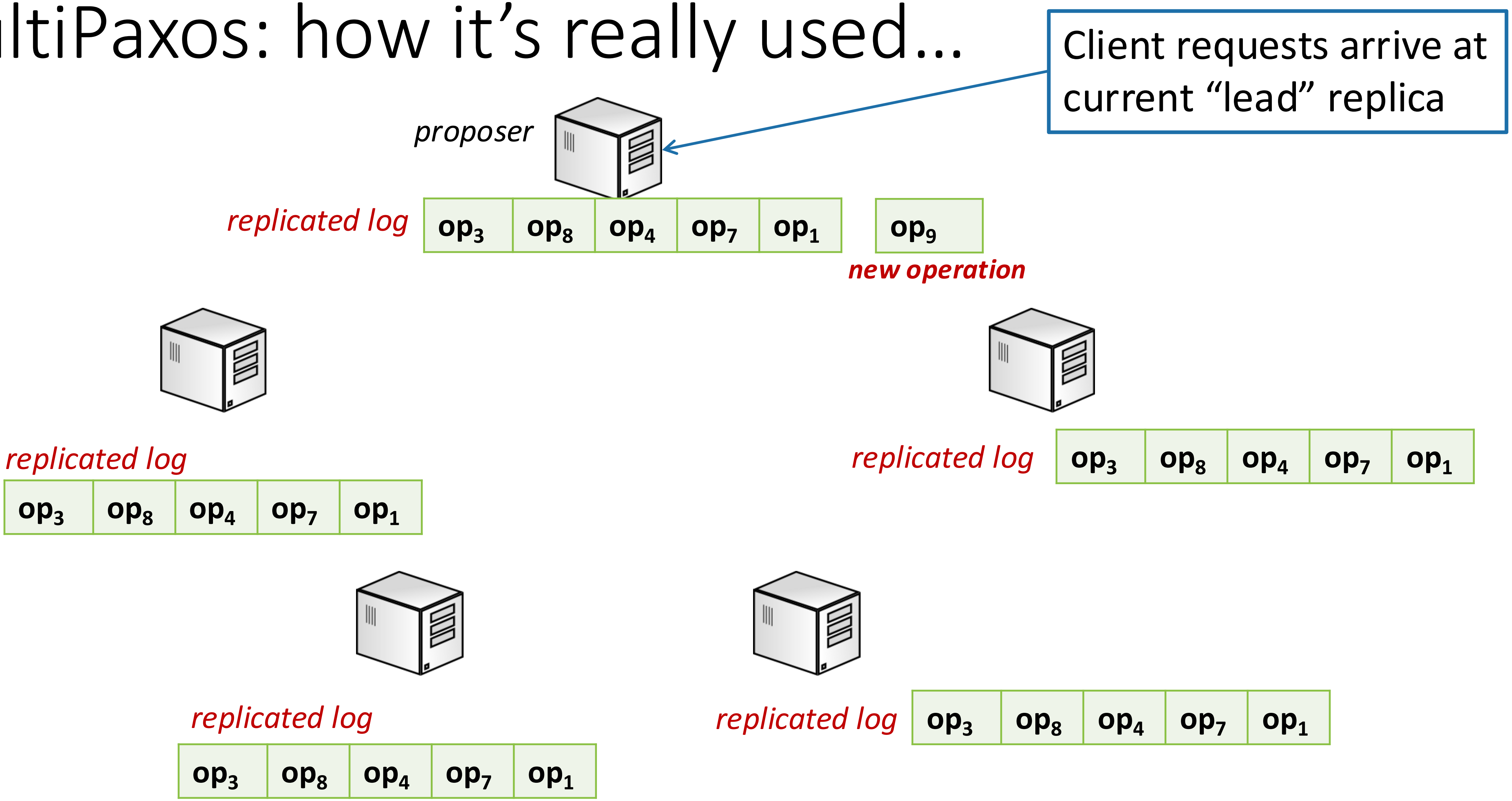
- The ability to agree on one value (which doesn’t change once it’s agreed) might not immediately seem useful

- Paxos is typically used to agree on a series of values, where consensus is reached on each individual value in the series (i.e. Multi-Paxos)
  - But implementing Paxos in this way is not easy...

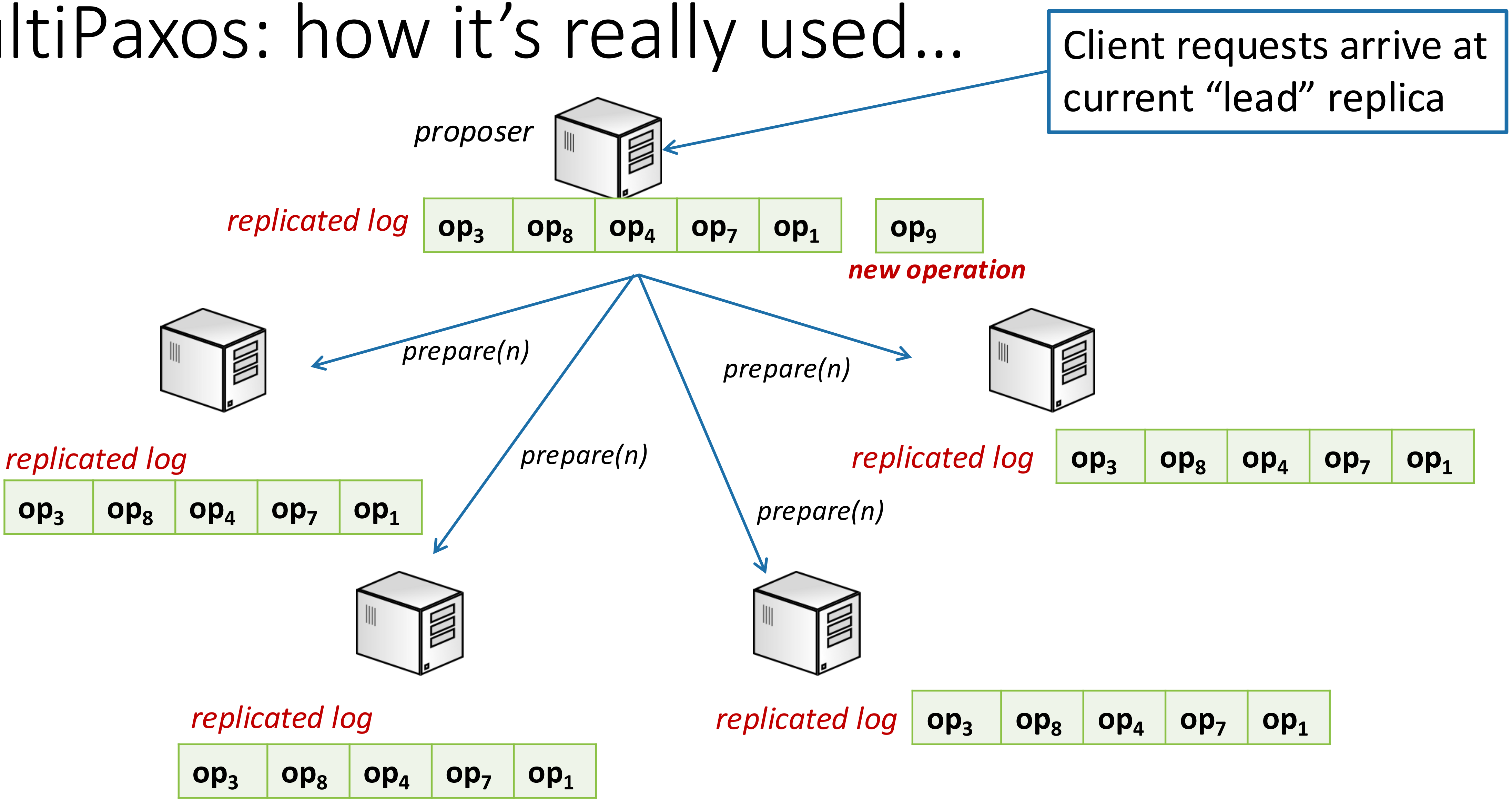
MultiPaxos: how it’s really used…



MultiPaxos: how it's really used…



MultiPaxos: how it's really used...



Two important properties

- **Safety**: nothing bad will happen, i.e., certain critical properties of a system are never violated, e.g.,
  - In a single round:
    - only a single operation is executed
    - A learner never learns an operation unless it has been accepted
- **Liveness**: eventually something good will happen, i.e., the system will (eventually) make progress
  - In order to tolerate f node simultaneously failing, Paxos would need a total of 2f+1 nodes (to still achieve majority with the remaining f+1 nodes)
  - When a majority can not be achieved, the protocol would stop making progress

Consensus vs. Total order multicast

- Consensus and total order group communication are seemingly two different problems, but in fact they are closely related!
- Traditional formulation of consensus is that you got multiple nodes, each node may propose a value, and you want all nodes to decide on the same value.
  - By some process, one value is picked by all nodes.
  - E.g., friends to agree on a restaurant to go.
- You can think of this as equivalent to total order multicast, where you want all the nodes to deliver the messages in the same order.
  - You can keep doing consensus once for each message to be delivered, and consensus will guarantee that all the nodes will make the same decision.
- Therefore, consensus and total order multicast are **formally equivalent problems**.
  - If you have an algorithm for one, you can convert that to an algorithm for the other.

Summary

- Introduced "State Machine Replication" (SMR) replication and difficulty of implementing SMR in the presence of failures
- Introduced distributed consensus as a way of implementing SMR
- Examined PAXOS – a consensus protocol that can enable SMR under failures

Further reading

- Section 7.5 (Distributed Commit) and 7.6 (Recovery) of Tanenbaum & van Steen; Sections 16 & 17 of Coulouris & al
- Chapter 7. Fault Tolerance of Tanenbaum & van Steen; Chapter 18 Coulouris & et. Al
- The part-time parliament, L. LAMPORT, ACM Transactions on Computer Systems, 1998
- Paxos Made Live - An Engineering Perspective, Chandra et al., PODC 2007