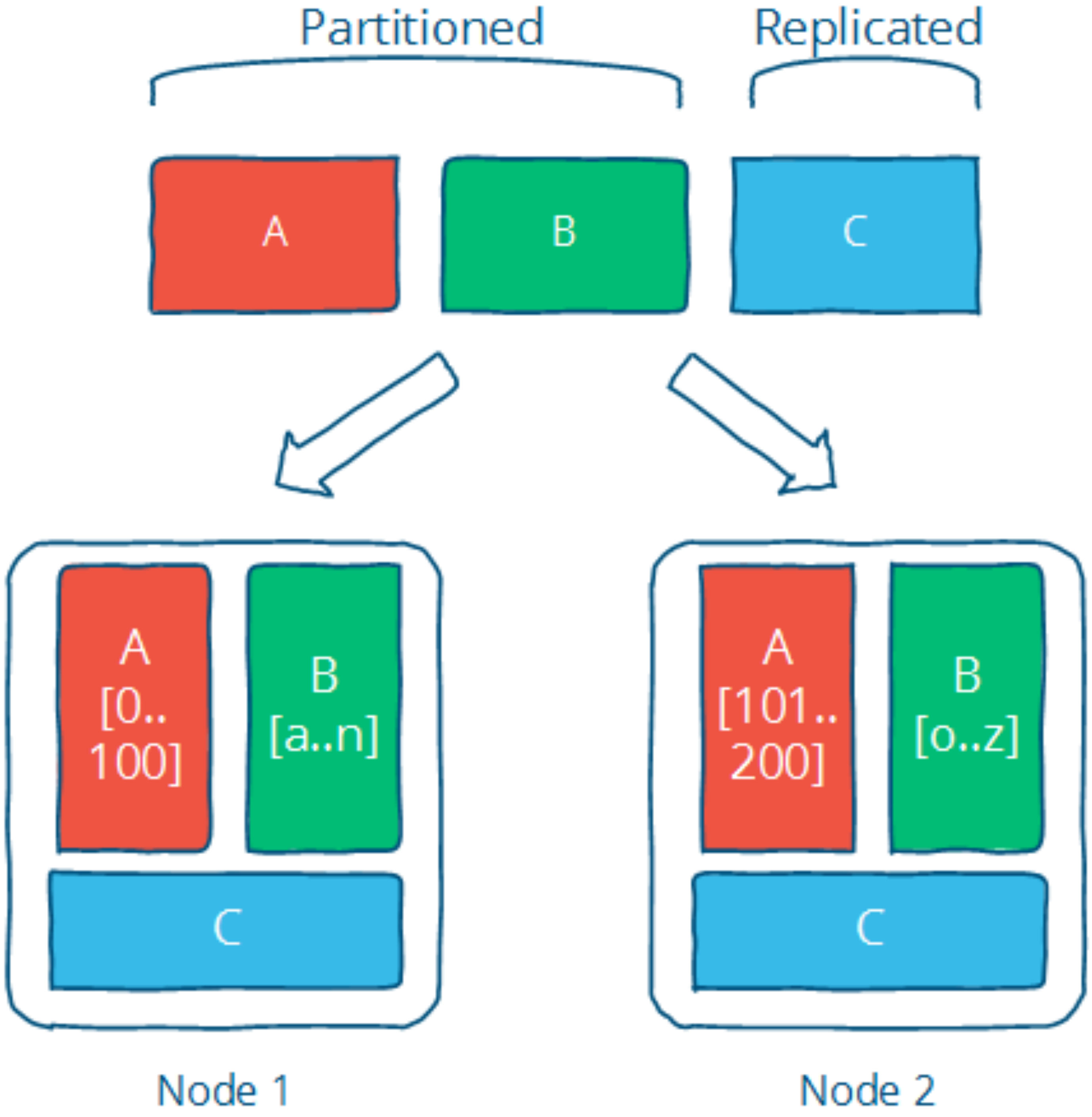
Data Design Techniques



Partitioning

- Partitioning is dividing the dataset into smaller distinct independent sets
- Partitioning is used to reduce the impact of dataset growth since each partition is a subset of the data.
- Partitioning improves performance by limiting the amount of data to be examined and by locating related data in the same partition.
- Partitioning improves availability by allowing partitions to fail independently, increasing the number of nodes that need to fail before availability is sacrificed.
- Partitioning is also very much application-specific, so it is hard to say much about it without knowing the specifics.

Replication

- Replication is making copies of the same data on multiple machines; this allows more servers to take part in the computation.

- Replication - copying or reproducing something - is the primary way in which we can fight latency.

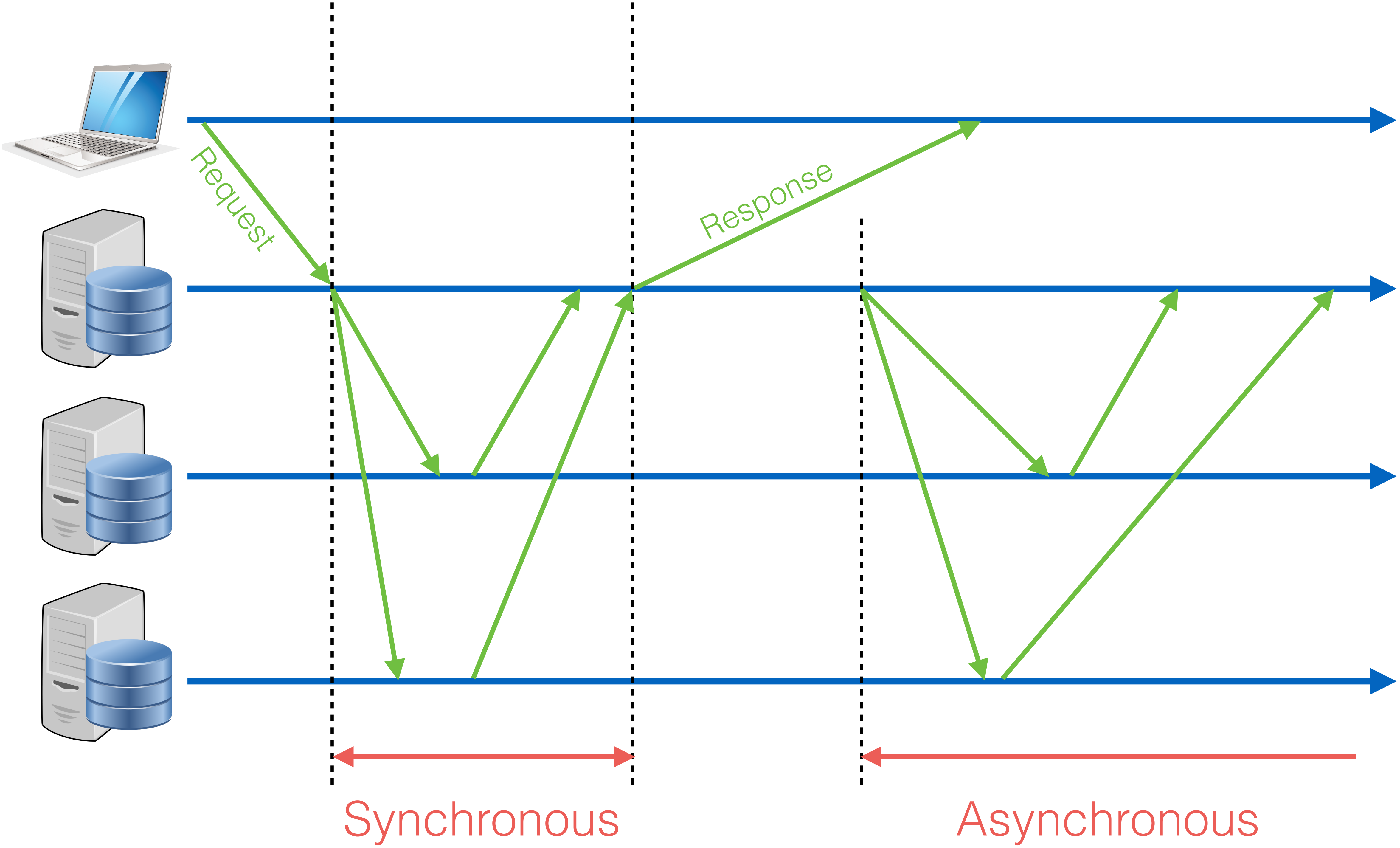
- Replication improves performance by making additional computing power and bandwidth applicable to a new copy of the data.

- Replication improves availability by creating additional copies of the data, increasing the number of nodes that need to fail before availability is sacrificed.

Replication & Consistency

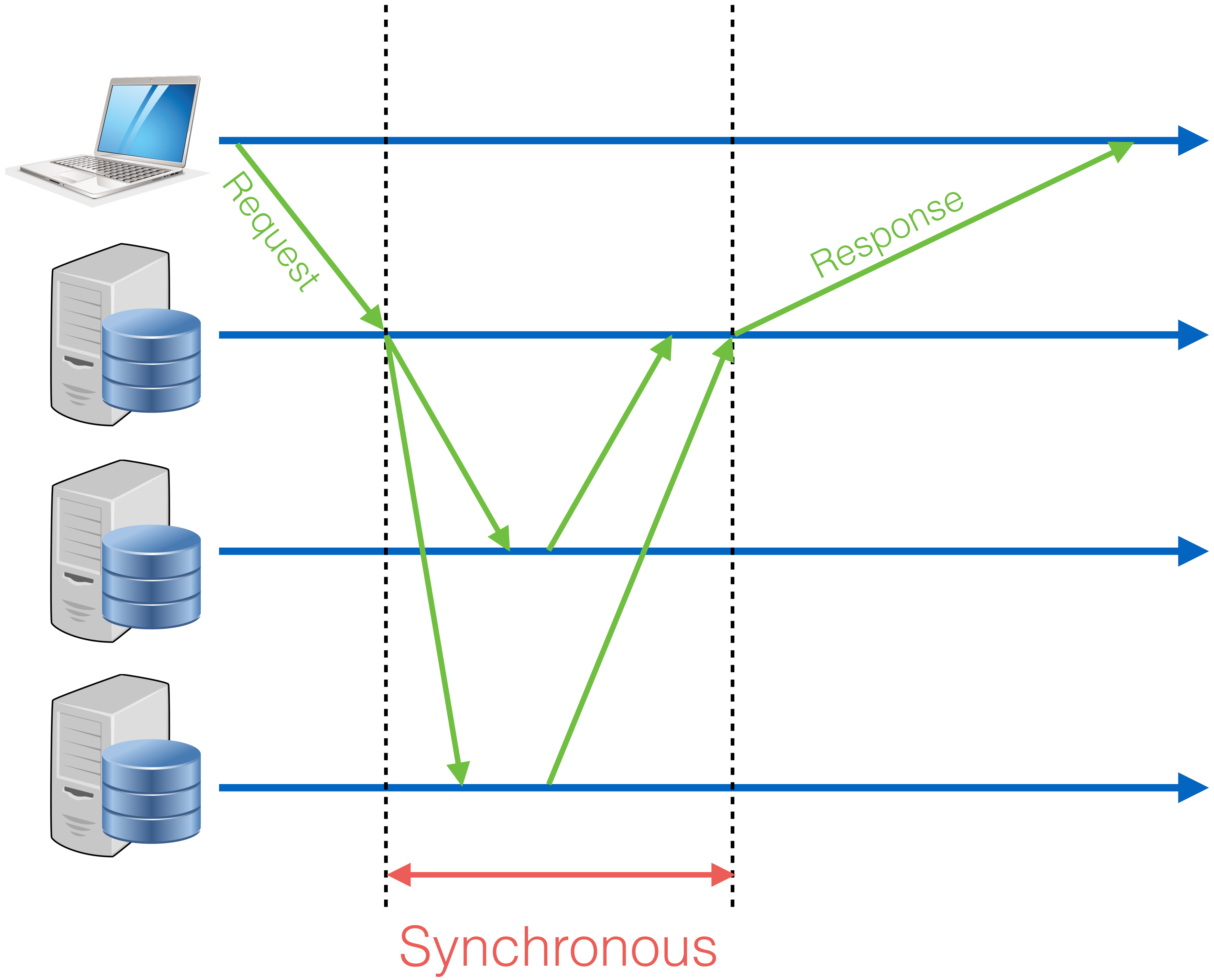
- one of many problems in distributed systems
- it is often the part that people are most interested in
- provides a context for many subproblems, such as leader election, failure detection, consensus and atomic broadcast
- References:
    - Coulouris, Dollimore, Kindberg, Blair, Distributed Systems - Concepts and Design, ch 15
    - http://book.mixu.net/distsys
    - http://research.microsoft.com/en-us/people/philbe/chapter7.pdf
    - http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf
    - http://static.googleusercontent.com/media/research.google.com/it//archive/paxos_made_live.pdf

Communication Patterns



Synchronous Replication

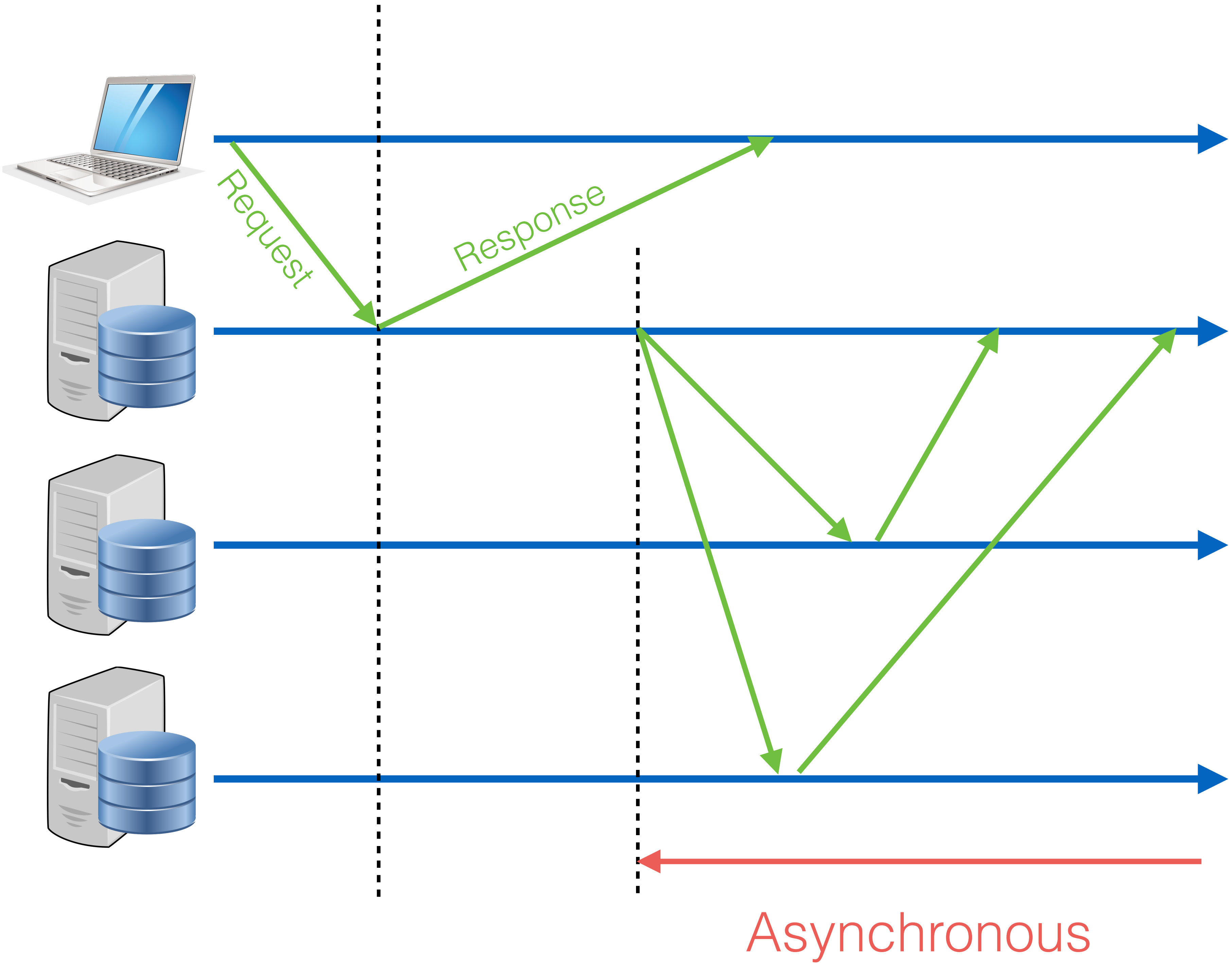
or active, or eager, or push, or pessimistic



- Three stages
- Client blocks
- N-to-N write
- **Transactions**
- High latency
- Low network robustness
- Low availability
- High durability

Asynchronous Replication

or passive, or pull, or lazy



- Master answers immediately
- Client does not block
- 1-to-N write
- Low latency
- High network robustness
- High availability
- Weak or probabilistic durability

Designing Replication Strategies

	Master (primary)	Group (everywhere)
Asynch (lazy)		
Synch (eager)		

Pros & Cons

Synchronous

+ No inconsistencies
+ Local copies always updated
+ Atomic changes
- Global single transaction
- Performance

Group

+ Anyone transaction
+ Load balancing
- Synchronized copies

Asynchronous

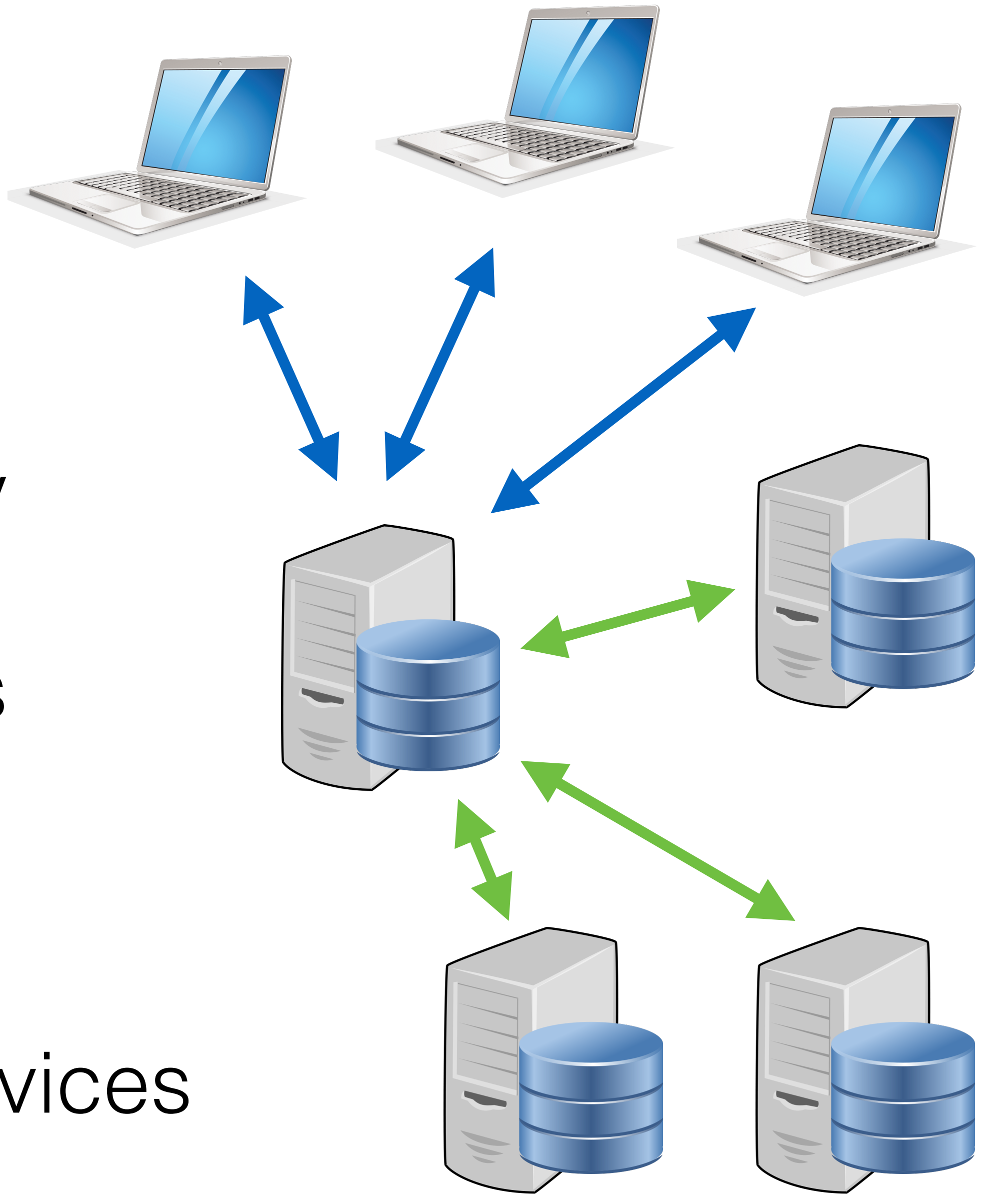
+ Local transaction
+ Performance
- Inconsistencies
- Local copies not updated
- Replication not transparent

Primary

+ Simple
+ Centralized updates
- Load unbalancing
- Single POF
- Local copies not updated

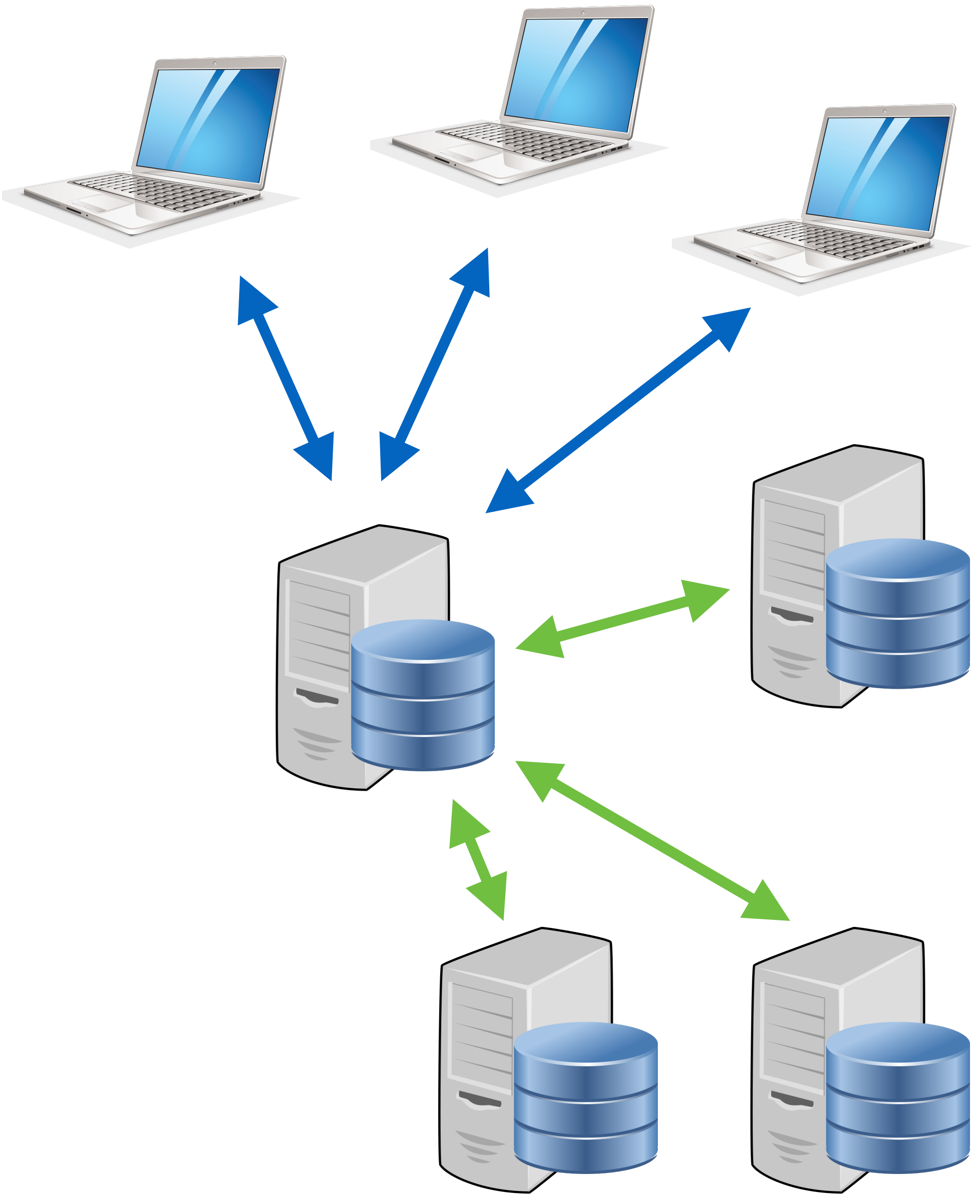
Asynch + Primary

- Primary Copy Replication
  - Simplest approach:
    - All requests go to primary copy
    - Other copies serve as backups
  - Widely used in practice
    - Keep backups of important services
    - Backup can go “live” when primary fails



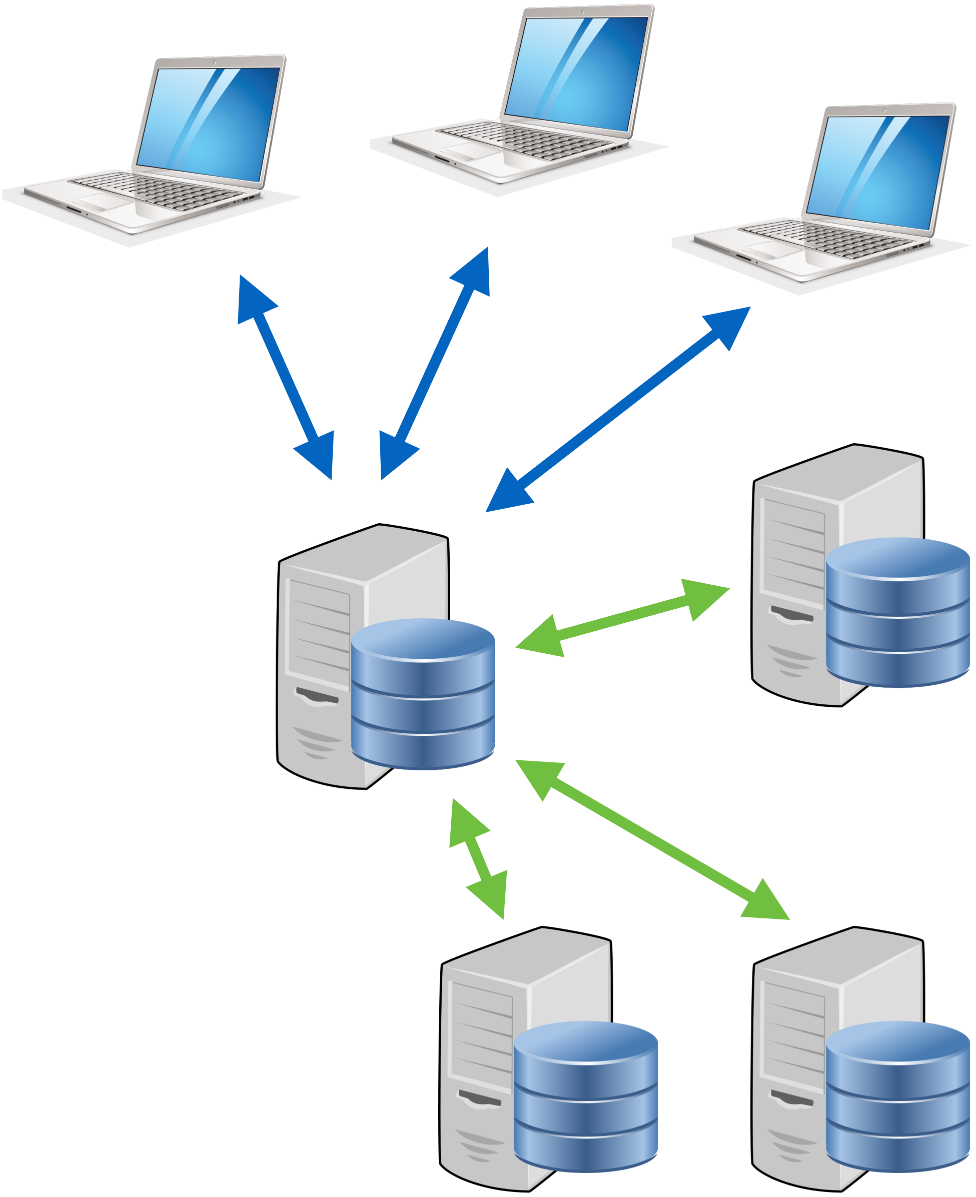
Primary Copy Replication

- All updates are performed on the primary
- log of operations (or alternatively, changes) is shipped across the network to the backup replicas
- Asynch and synch variants
- Asynch used by MySQL and MongoDB



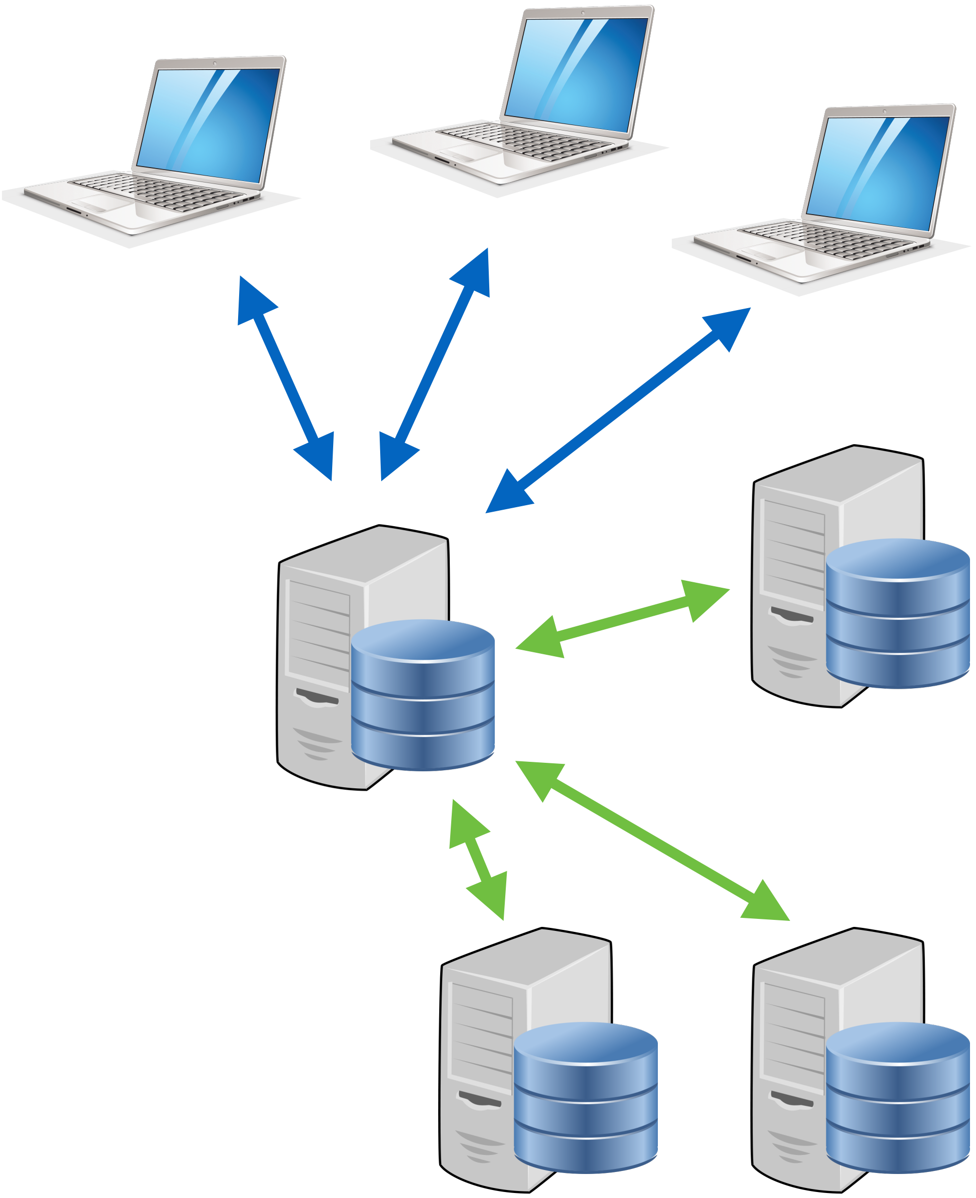
Primary Copy Fault Tolerance

- Failures can cause violations to consistency guarantees
- Site failures, network partitions, undelivered messages
- We need an atomic commitment protocol (ACP)
- Commonly used in database systems (i.e., transactions)



Atomic Commitment Protocols

- 1 coordinator, N participants
- Coordinator knows the names of all participants
- Participants know the name of the coordinator, not necessarily each other
- Each site contains a distributed transaction log surviving failures
- Each process may cast exactly one of two votes: **Yes** or **No**
- Each process can reach exactly one of two decisions: **Commit** or **Abort**
- Consensus Problem!!!!



ACP Conditions

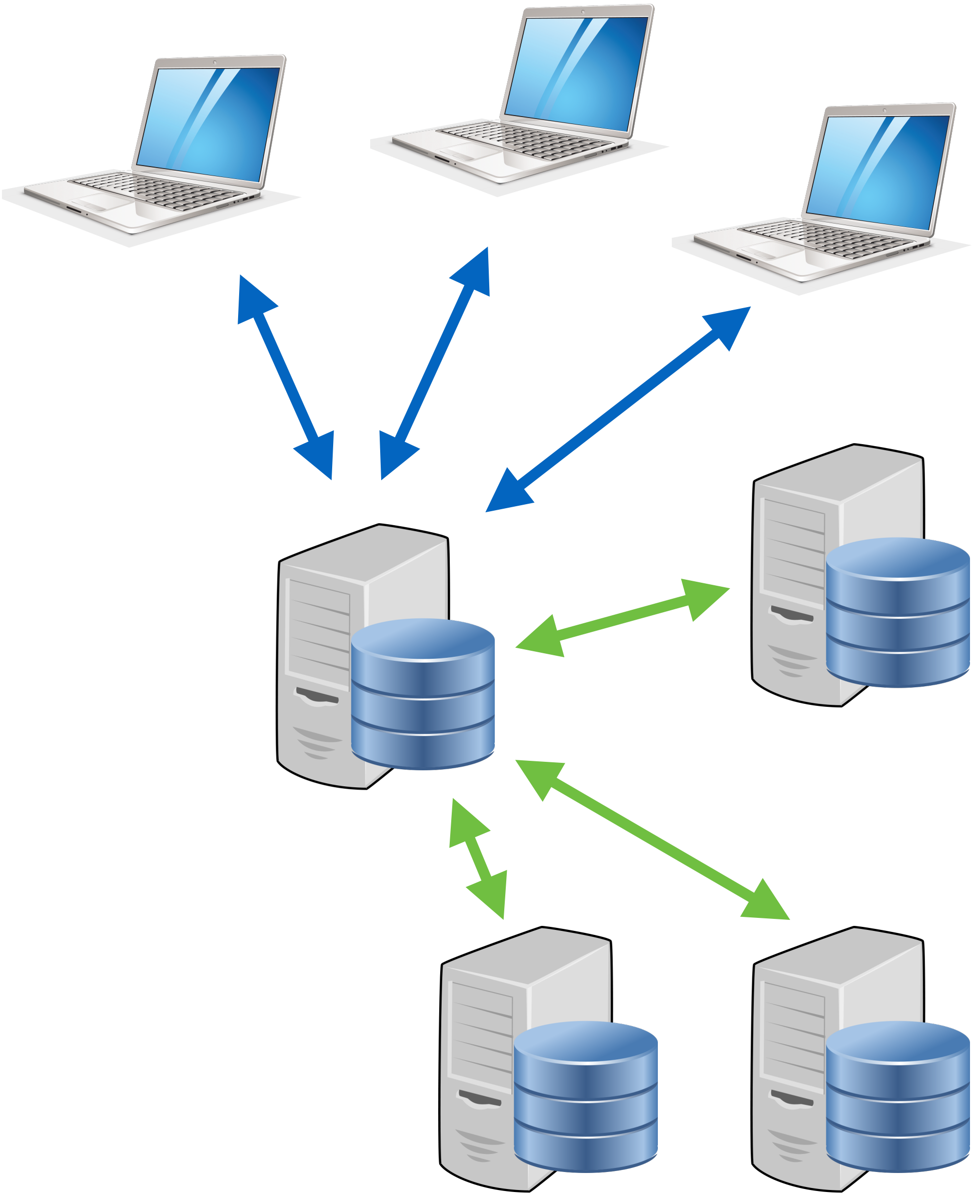
1. All processes that reach a decision reach the same one.

2. A process cannot reverse its decision after it has reached one.

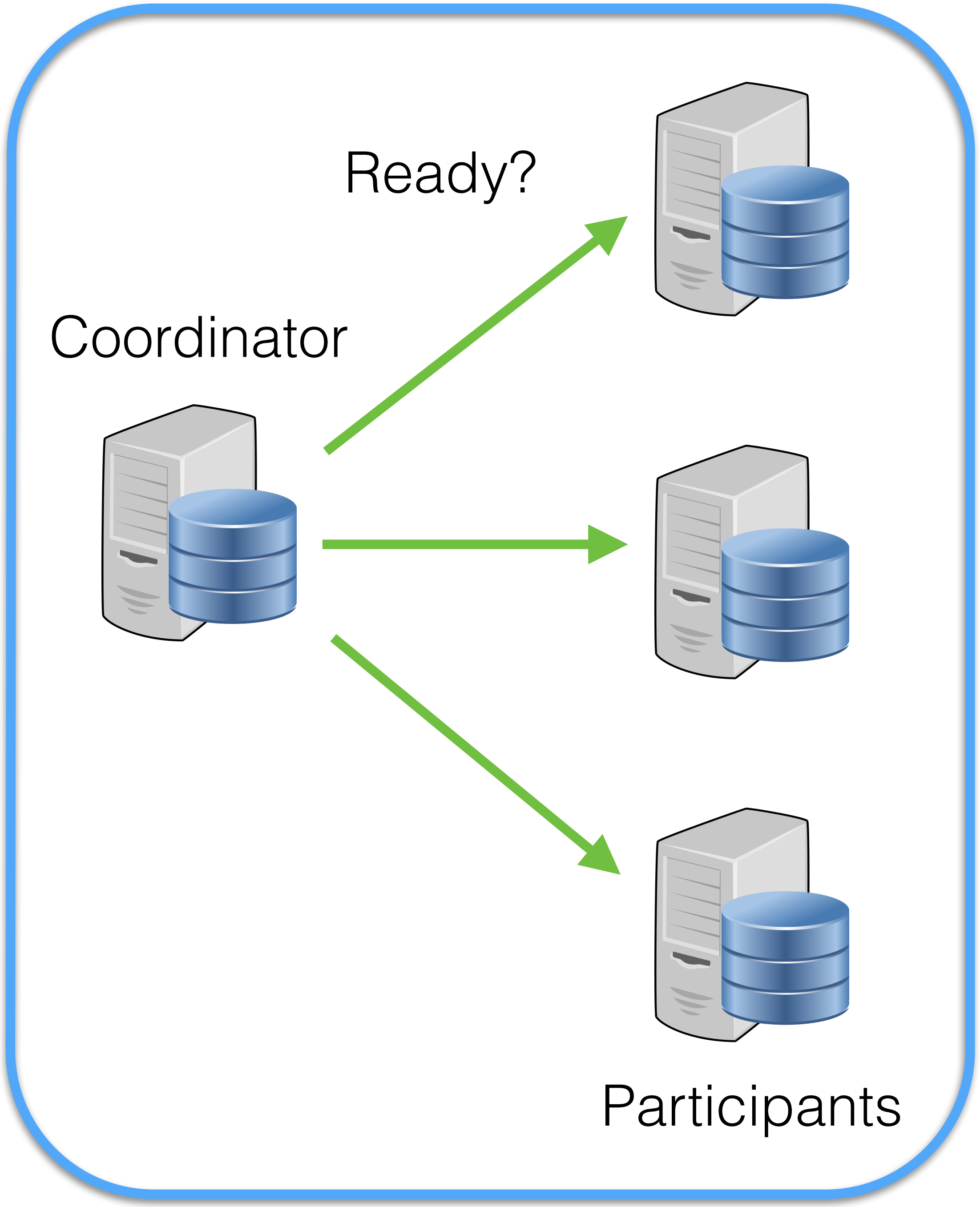
3. The **Commit** decision can only be reached if all processes voted **Yes**.

4. If there are no failures **and** all processes voted **Yes**, then the decision will be to **Commit**.

5. Consider any execution containing only failures that the algorithm is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.



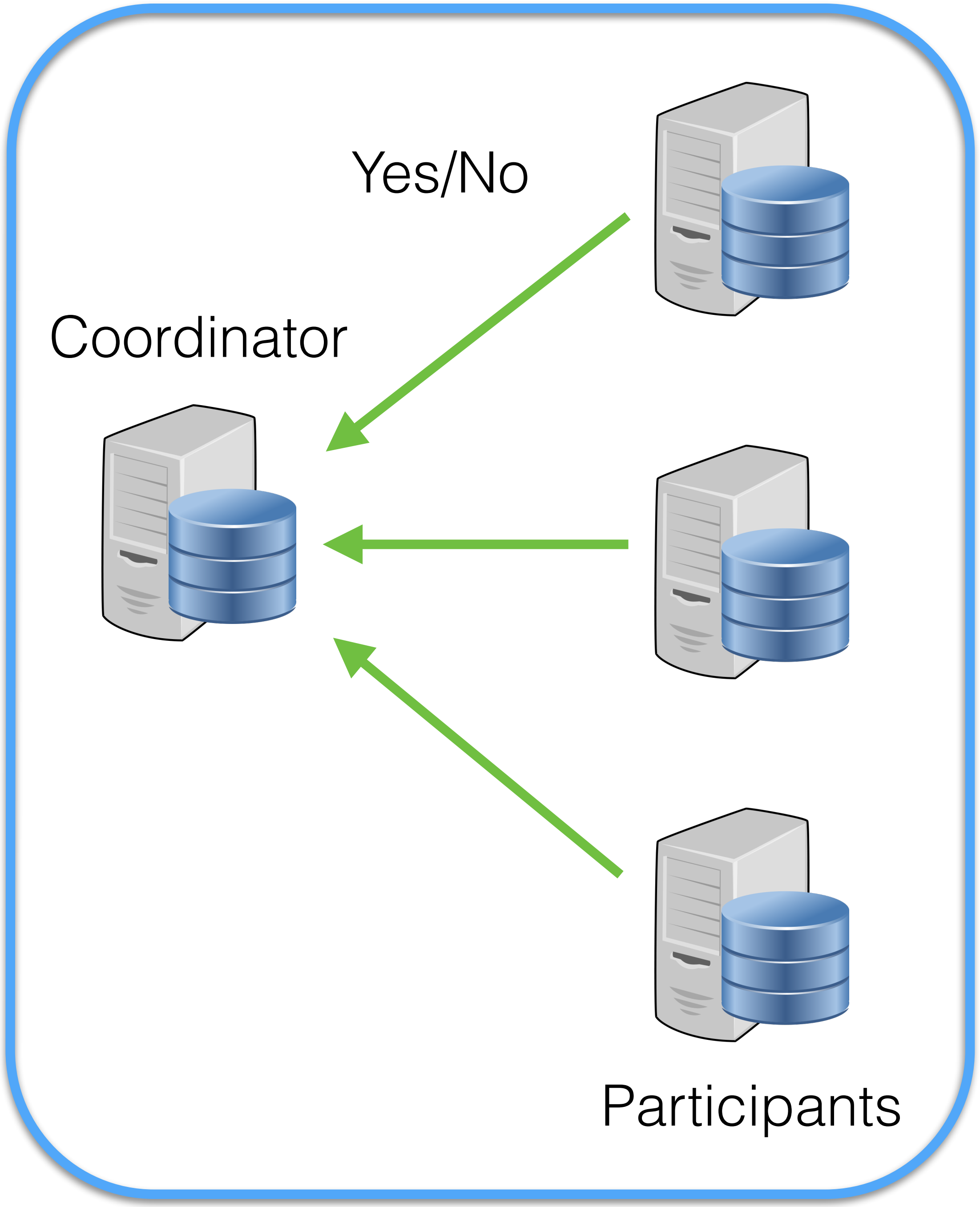
Two Phase Commit (2PC)



Ready?

Coordinator

Participants

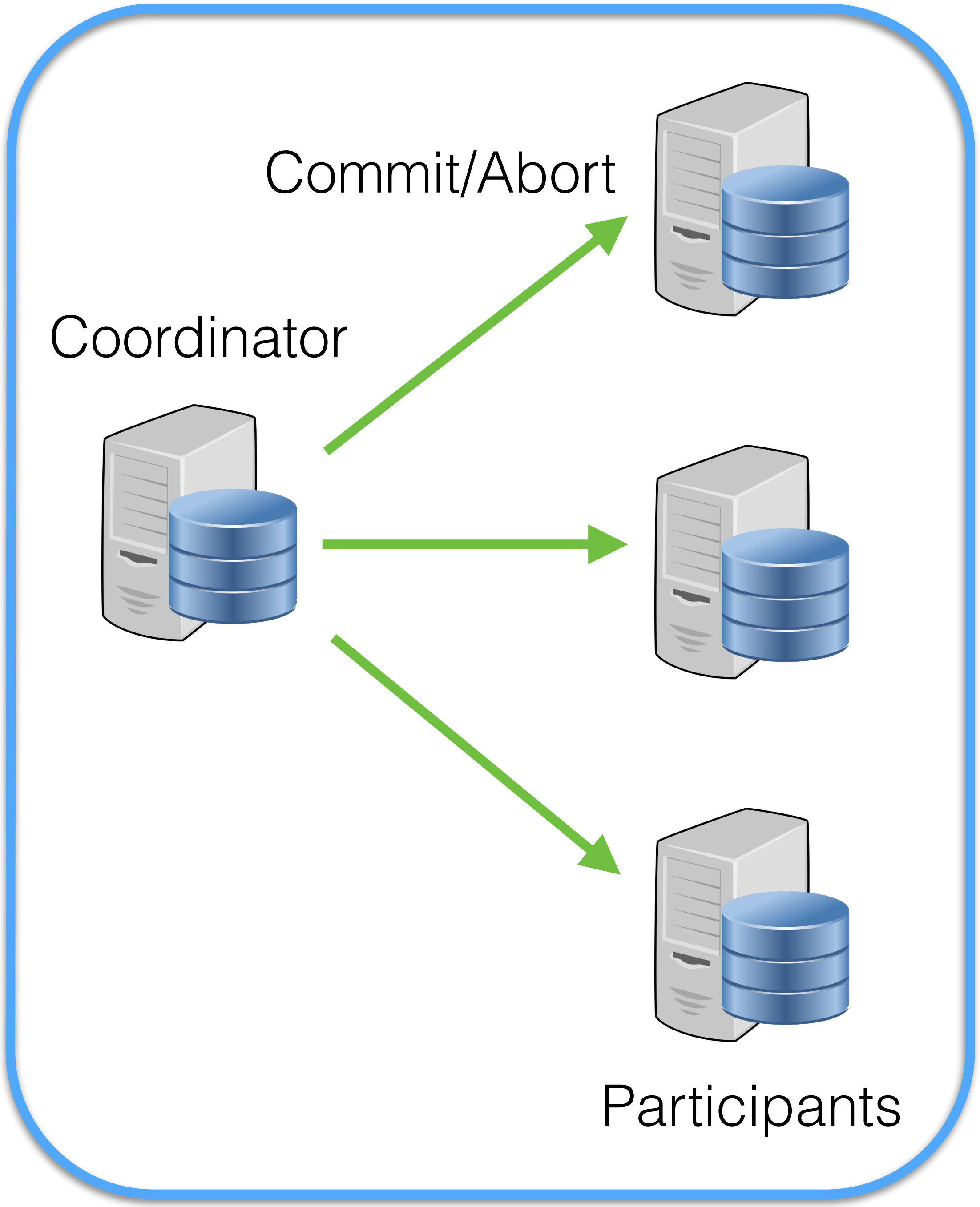


Yes/No

Coordinator

Participants

Two Phase Commit (2PC)



- Terminates if no faults
- Satisfy conditions 1-4
- Does not satisfy condition 5 (as expected)
- *3n* messages for *n* nodes
- What happens with faults?

2PC - Phase 1

```
coordinator sends ready to all participants
```

```
if a participant receives ready from the coordinator:
    if it is ready to commit:
        send yes to the coordinator
    else:
        send no to the coordinator
```

2PC - Phase 2

```
if the coordinator receives only yes messages:
    send commit to all nodes
else
    send abort to all nodes
```

```
if a node receives commit from the coordinator:
    commit the transaction
else (abort received)
    abort the transaction
```

2PC and Faults

- Fail-stop model: We assume that a failed node does not re-emerge
- Failures are detected (instantly)
  - Timeouts are used in practical systems to detect failures (FLP theorem does not apply there!)
- When a waiting process is interrupted by a timeout, the process must take special action, called a timeout action.

Timeout Actions

- Coordinator waiting: no decision taken yet, can decide to send **abort** or **commit** to participants

- Participant waiting for ready: can unilaterally decide **abort** before it sends **yes**

- Participant waiting for **commit/abort**: has already voted **yes**, so it can not unilaterally decide. 2PC blocks.

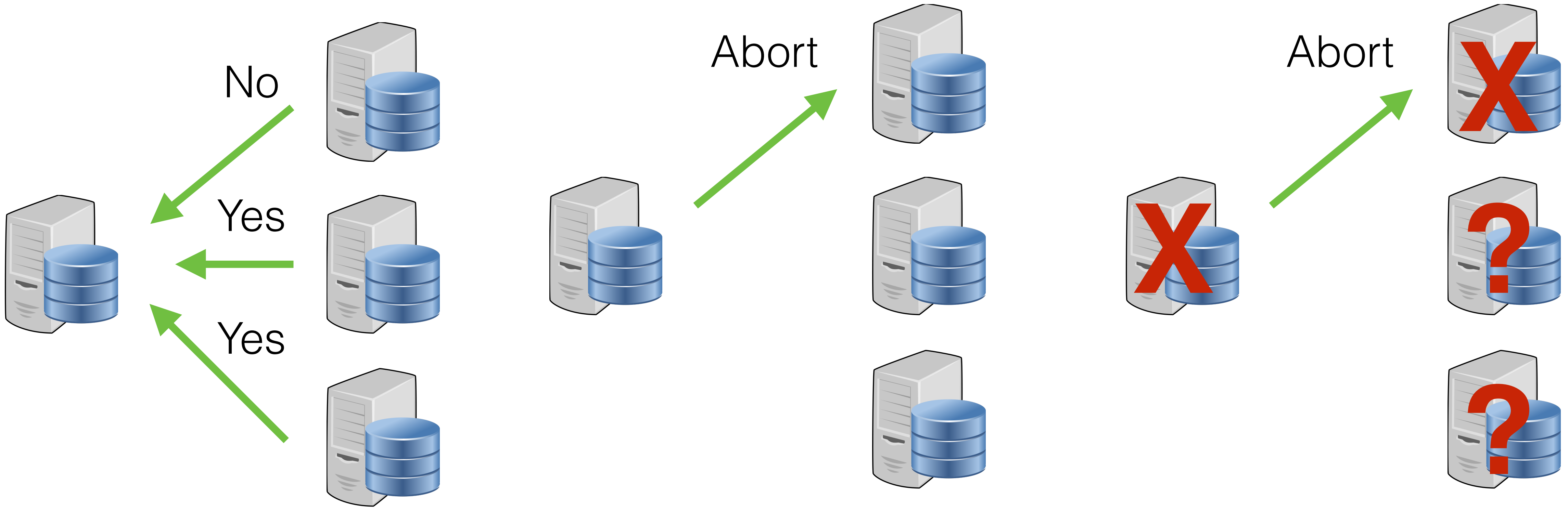
- We need a safety mechanism to deal with block (not part of 2PC).

Safety Mechanism

- All participants must know each other
- Coordinator failures is detected
- A new coordinator is selected
- It must ask the other participants if a participant has already received a **commit**
- A participant that has received a **commit** replies **yes**, otherwise it sends **no** and *promises not to accept a commit that may arrive from the old coordinator*
- If some participant replied **yes**, the new coordinator broadcasts **commit**
- What if we have multiple failures???

2PC and multiple failures

- Multiple participant failures:
  - No problem as long as the coordinator lives
- Coordinator and participant failures:

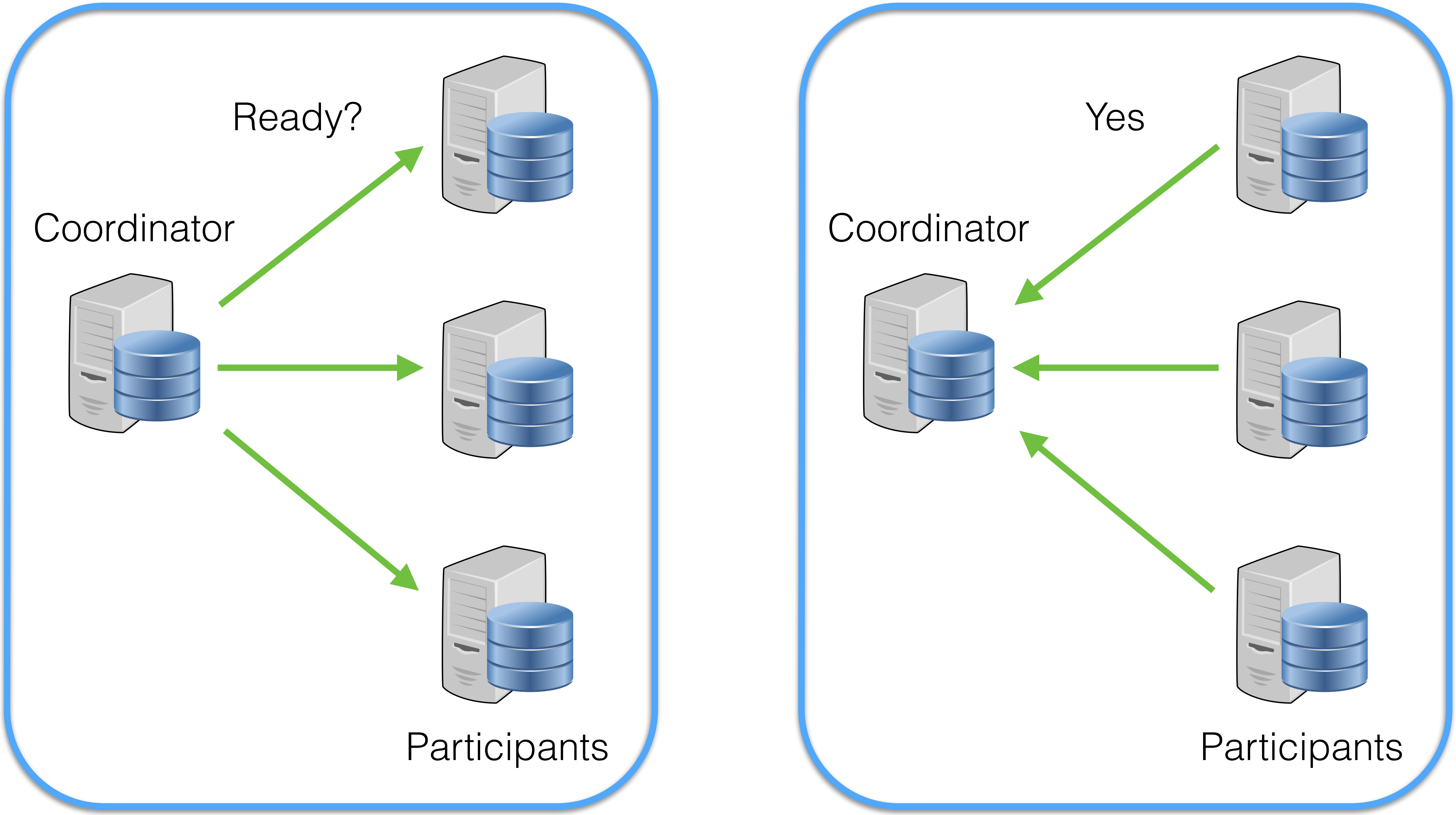


- The remaining participants cannot take a decision

2PC and multiple failures

- Solution: Add another phase to the protocol!
  - The new phase precedes the commit phase
  - The goal is to inform all participants that all are ready to commit or abort
  - At the end of this phase, every participant knows whether or not all participants want to commit before any participant has actually committed or aborted!
- This protocol is called the three-phase commit protocol (3PC)

Three Phase Commit (3PC)



Ready?

Coordinator

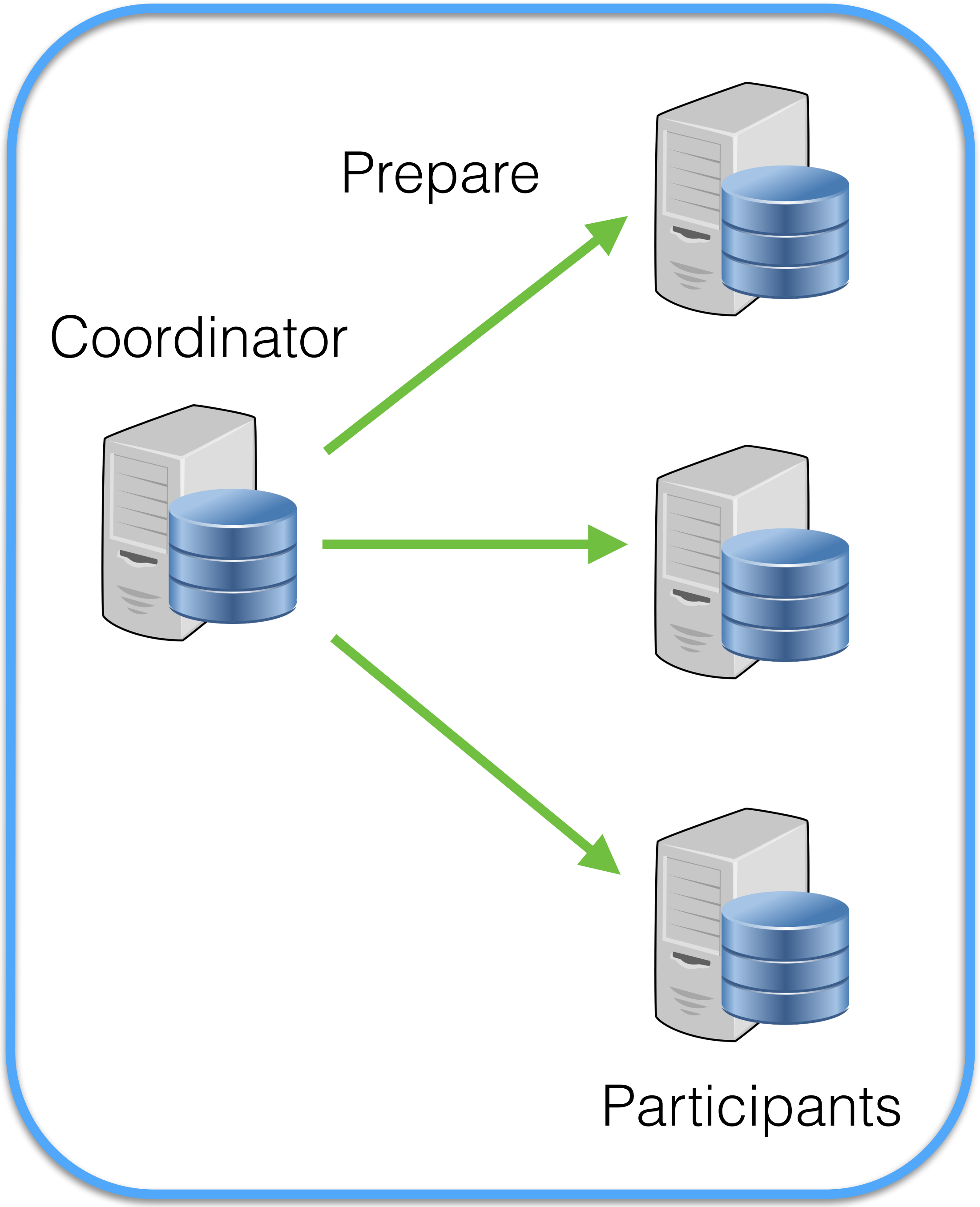
Participants

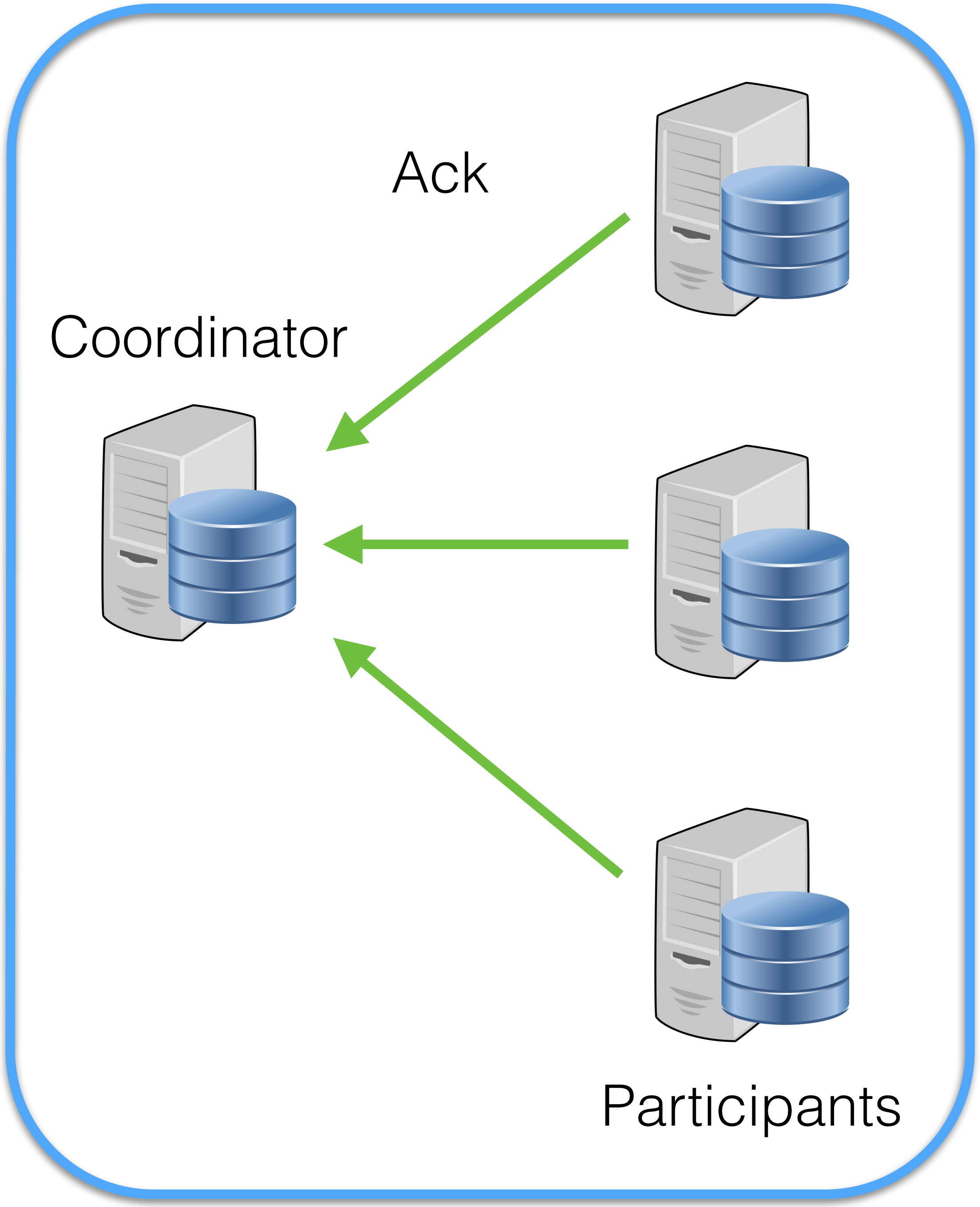
Yes

Coordinator

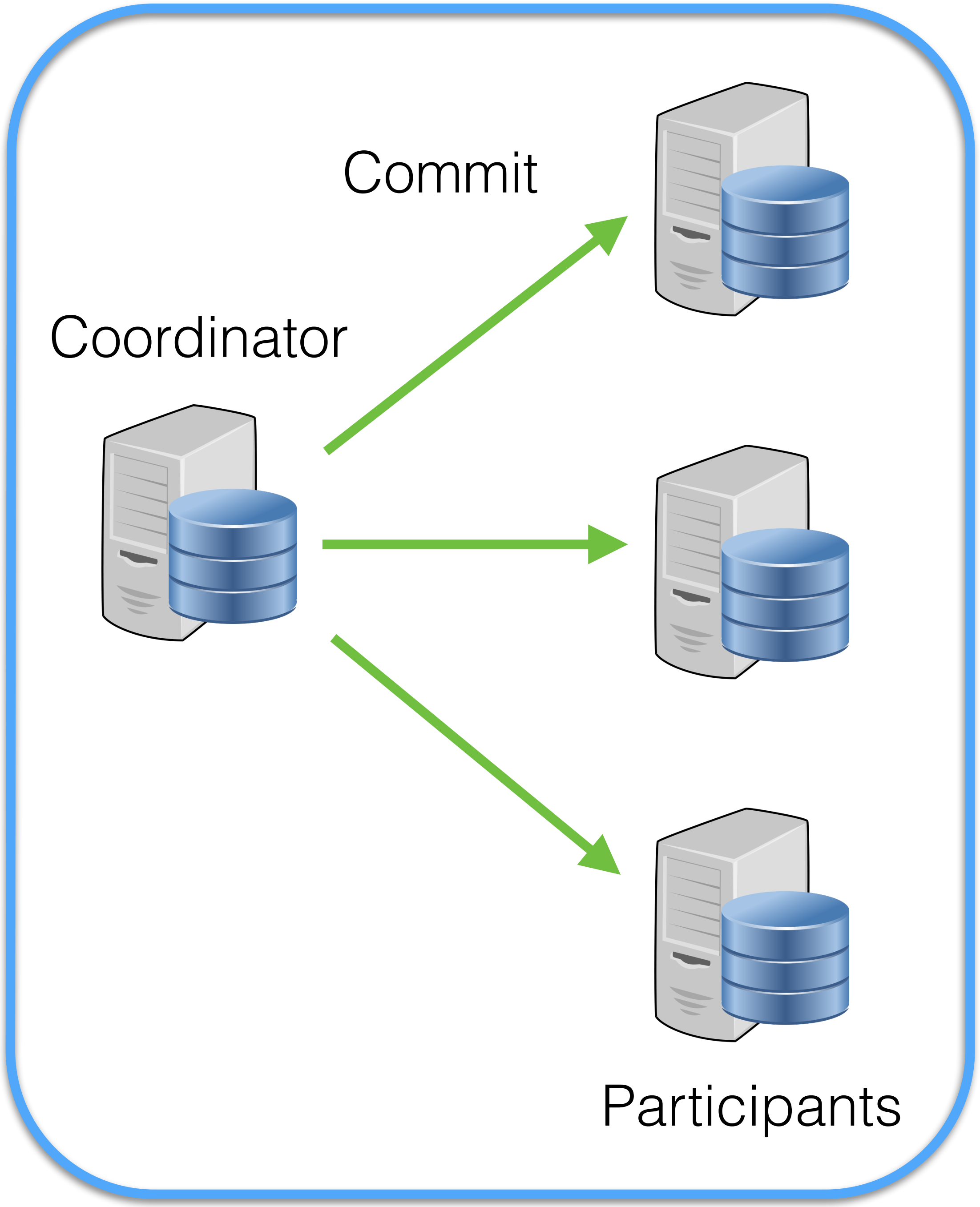
Participants

Three Phase Commit (3PC)

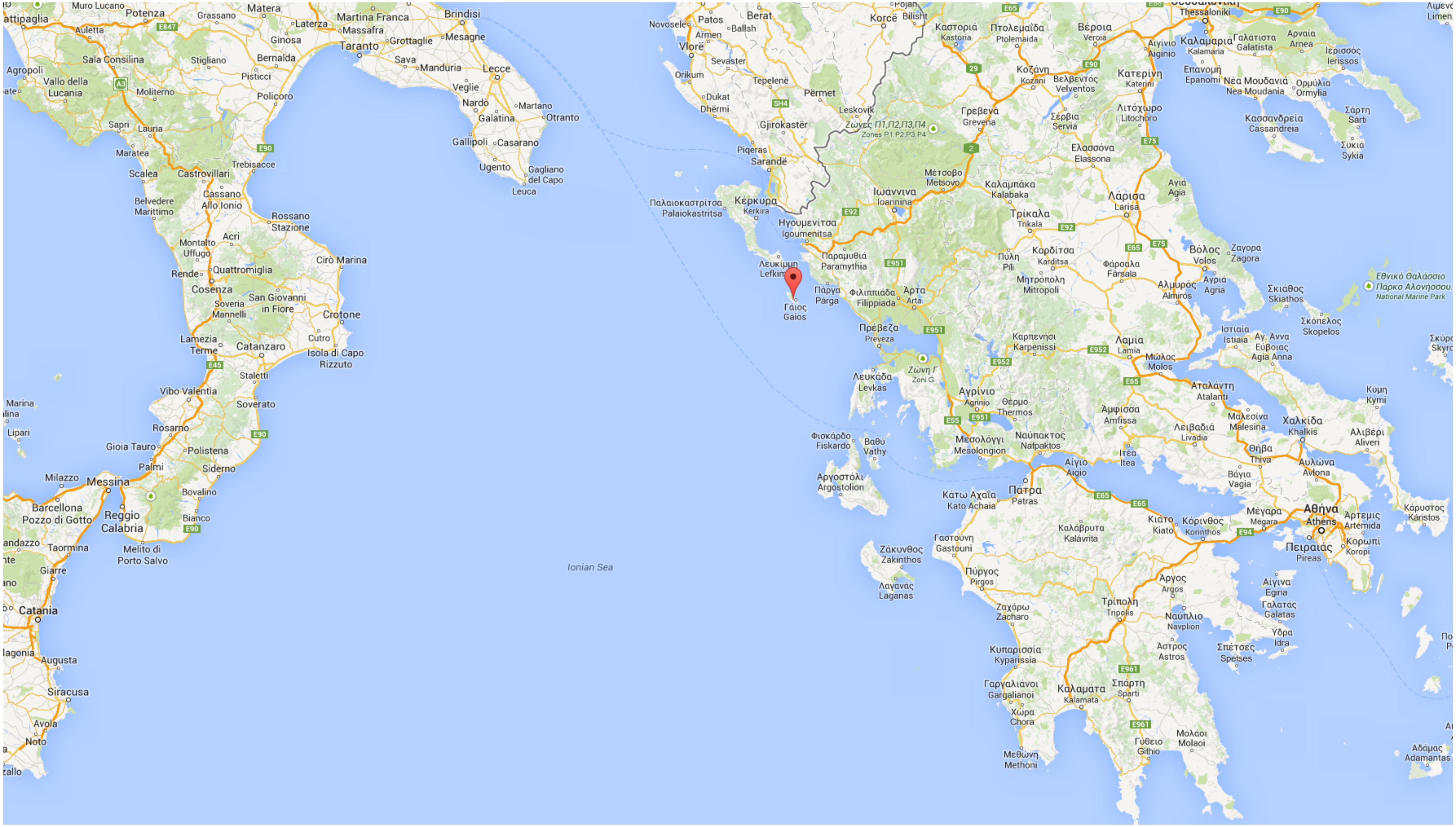




Three Phase Commit (3PC)



PAXOS



Muro Lucano
Potenza
Grassano
Matera
Laterza
Martina Franca
Massafra
Brindisi
Grottaglie
Mesagne
Taranto
Ginosa
Bernalda
Sava
Manduria
Lecce
Veglie
Nardò
Martano
Galatina
Otranto
Gallipoli
Casarano
Ugento
Gagliano del Capo
Leuca
Auletta
Sala Consilina
Stigliano
Pisticci
Policoro
Agropoli
Vallo della Lucania
Moliterno
Sapri
Lauria
Maratea
Trebisacce
Scalea
Castrovillari
Cassano Allo Ionio
Belvedere Marittimo
Rossano Stazione
Acri
Montalto Uffugo
Cirò Marina
Rende
Quattromiglia
Cosenza
San Giovanni in Fiore
Soveria Mannelli
Crotone
Lamezia Terme
Catanzaro
Cutro
Isola di Capo Rizzuto
Staletti
Vibo Valentia
Soverato
Marina
Lipari
Rosarno
Gioia Tauro
Polistena
Palmi
Siderno
Milazzo
Messina
Bovalino
Barcellona Pozzo di Gotto
Reggio Calabria
Bianco
Melito di Porto Salvo
Taormina
Giarre
Catania
Augusta
Siracusa
Avola
Noto
Ionian Sea
Novoselë
Patos
Berat
Ballsh
Fier
Vlorë
Sevaster
Orikum
Tepelenë
Përmet
Dukat
Dhërmi
Leskovik
Gjirokastër
Korçë
Bilisht
Piqeras
Sarandë
Ζώνες Π1,Π2,Π3,Π4
Zones P1,P2,P3,P4
Καστοριά
Kastoria
Πτολεμαΐδα
Ptolemaida
Βέροια
Veroia
Thessaloniki
Κοζάνη
Kozani
Βελβεντός
Velventos
Κατερίνη
Katerini
Γρεβενά
Grevena
Σέρβια
Servia
Λιτόχωρο
Litochoro
Ελασσόνα
Elassona
Μέτσοβο
Metsovo
Ιωάννινα
Ioannina
Καλαμπάκα
Kalabaka
Λάρισα
Larisa
Αγιά
Agia
Τρίκαλα
Trikala
Παλαιοκαστρίτσα
Palaiokastritsa
Κέρκυρα
Kerkira
Ηγουμενίτσα
Igoumenitsa
Παραμυθιά
Paramythia
Λευκίμμη
Lefkimi
Γάιος
Gaios
Πάργα
Parga
Φιλιππιάδα
Filippiada
Άρτα
Arta
Πύλη
Pili
Καρδίτσα
Karditsa
Μητρόπολη
Mitropoli
Φάρσαλα
Farsala
Βόλος
Volos
Ζαγορά
Zagora
Αλμυρός
Almiros
Αγριά
Agria
Πρέβεζα
Preveza
Καρπενήσι
Karpenissi
Λαμία
Lamia
Μώλος
Molos
Ζώνη Γ
Zoni G
Λευκάδα
Levkas
Αγρίνιο
Agrinio
Θέρμο
Thermos
Αταλάντη
Atalanti
Αμφισσα
Amfissa
Λειβαδιά
Livadia
Μαλεσίνα
Malesina
Χαλκίδα
Khalkis
Φισκάρδο
Fiskardo
Βαθύ
Vathy
Μεσολόγγι
Mesolongion
Ναύπακτος
Nafpaktos
Ιτέα
Itea
Θήβα
Thiva
Αυλώνα
Avlona
Αργοστόλι
Argostolion
Αίγιο
Aigio
Βάγια
Vagia
Κάτω Αχαΐα
Kato Achaia
Πάτρα
Patras
Μέγαρα
Megara
Αθήνα
Athens
Αρτεμις
Artemida
Κιάτο
Kiato
Κόρινθος
Korinthos
Καλάβρυτα
Kalavrita
Πειραιάς
Pireas
Κορωπί
Koropi
Γαστούνη
Gastouni
Ζάκυνθος
Zakinthos
Πύργος
Pirgos
Λαγανάς
Laganas
Αργος
Argos
Αίγινα
Egina
Γαλατάς
Galatas
Ζαχάρω
Zacharo
Τρίπολη
Tripolis
Ναύπλιο
Navplion
Ύδρα
Idra
Κυπαρισσία
Kyparissia
Άστρος
Astros
Σπέτσες
Spetses
Γαργαλιάνοι
Gargalianoi
Καλαμάτα
Kalamata
Σπάρτη
Sparti
Χώρα
Chora
Γύθειο
Githio
Μολάοι
Molaoi
Μεθώνη
Methoni
Αίγινιο
Aiginio
Καλαμαριά
Kalamaria
Γαλάτιστα
Galatista
Αρναία
Arnea
Ιερισσός
Ierissos
Επανομή
Epanomi
Νέα Μουδανιά
Nea Moudania
Ορμύλια
Ormylia
Σάρτη
Sarti
Κασσανδρεία
Cassandreia
Συκιά
Sykia
Εθνικό Θαλάσσιο Πάρκο Αλοννήσου
National Marine Park
Σκιάθος
Skiathos
Σκόπελος
Skopelos
Ιστιαία
Istiaia
Αγ. Αννα
Ευβοίας
Agia Anna
Κύμη
Kymi
Αλιβέρι
Aliveri
Κάρυστος
Karistos
Αδάμας
Adamantas

PAXOS

- Leslie Lamport. **The part-time parliament**. ACM Transactions on Computer Systems, 16(2):133–169, May 1998.
- Leslie Lamport described in 1990 the algorithm as the solution to a problem of the parliament on a fictitious Greek island called Paxos (not Italy)
- Many readers were so distracted by the description of the activities of the legislators, they did not understand the meaning and purpose of the algorithm. The paper was rejected.
- Leslie Lamport refused to rewrite the paper. He later wrote that he "was quite annoyed at how humorless everyone working in the field seemed to be"
- After a few years, some people started to understand the importance of the algorithm
- After eight years, Leslie Lamport submitted the paper again, basically unaltered. It got accepted!

Leslie Lamport
ACM Turing Award 2014

Consensus (again)

- We have a collection of processes
- Each process can propose a value
- A single value among the proposed values is chosen
- If no value is proposed, no value should be chosen
- If a value has been chosen, processes should be able to learn the chosen value

System Model

- Asynchronous communications
- Non-byzantine failures:
  - Nodes crash
  - Messages can be lost, duplicated arbitrarily late
  - No corrupted messages
- Fail-recover: crashed nodes may recover later

Roles

- Each nodes has one or more of 3 roles
- Proposer: a node that can propose a certain value for acceptance
- Acceptor: a node that receives proposals from proposers and that can either accept or reject a proposal
- Learner: a node not involved in the decision process that wants to know the final result of the decision process
- We will assume all nodes act both as proposer and as acceptor

Acceptors

- A single acceptor (a.k.a. coordinator) can fail and block the whole procedure

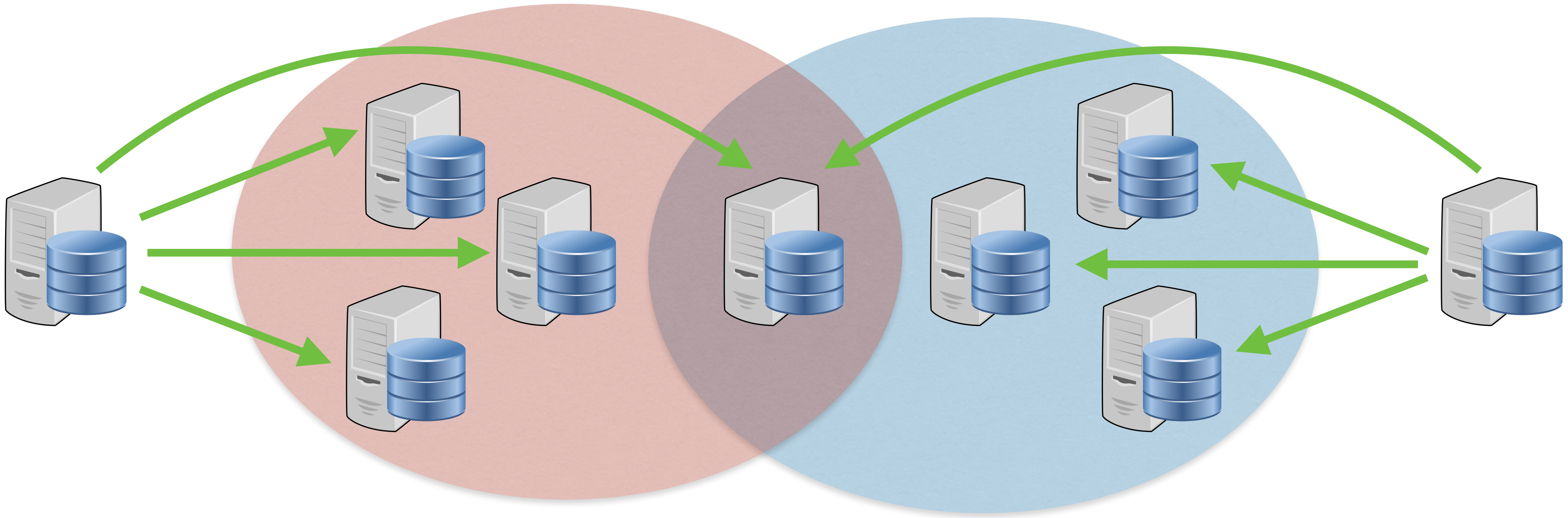
- There is not a coordinator, but multiple acceptors.

- An acceptor may **accept** a single value (e.g., express a single vote)

- How many acceptors do we need?

Majority

- To ensure that a single value is chosen, any majority (50% +1) of acceptors is enough
- The intersection of two majorities is not empty
- An acceptor may **accept** a single value
- A value is **chosen** when a majority of acceptors has accepted it
- If a majority choses a value, no other majority can chose a different value
- If an acceptor crashes, chosen value still available



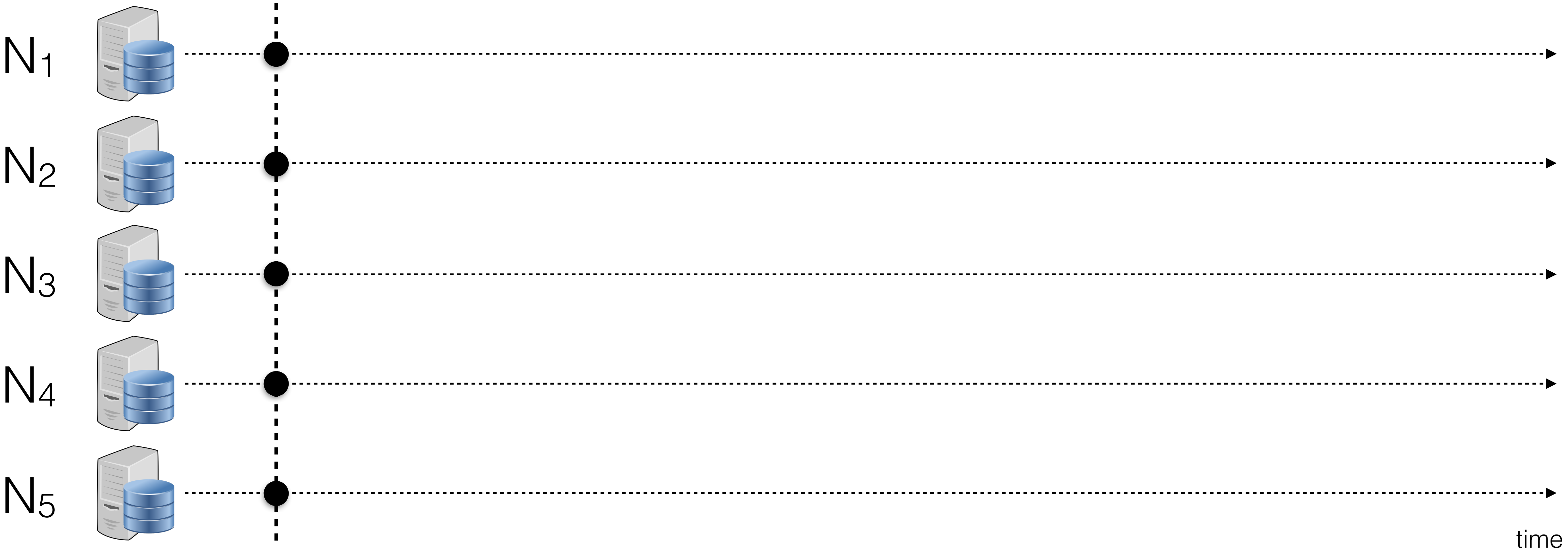
Acceptors

Since there can be a single proposer, the algorithm must guarantee that

P1: An acceptor must accept the first proposed value it receives

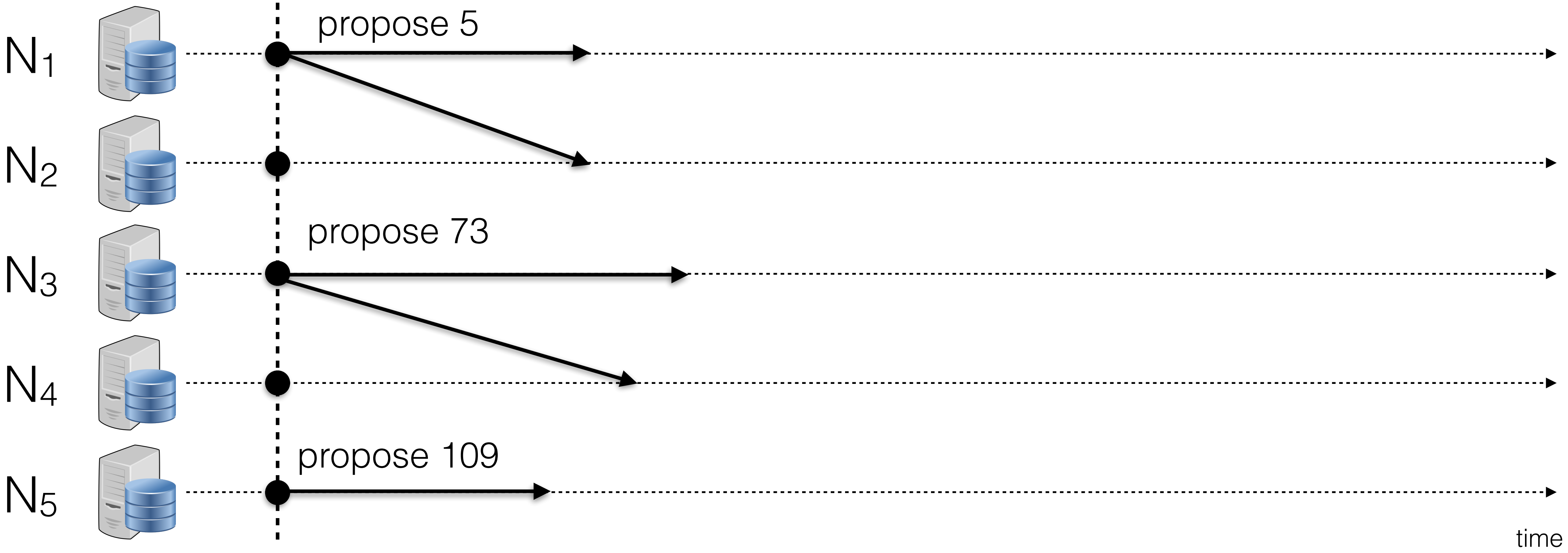
Since there can be a single proposer, the algorithm must guarantee that a single value is chosen only when it is accepted by a majority of acceptors

Split Votes



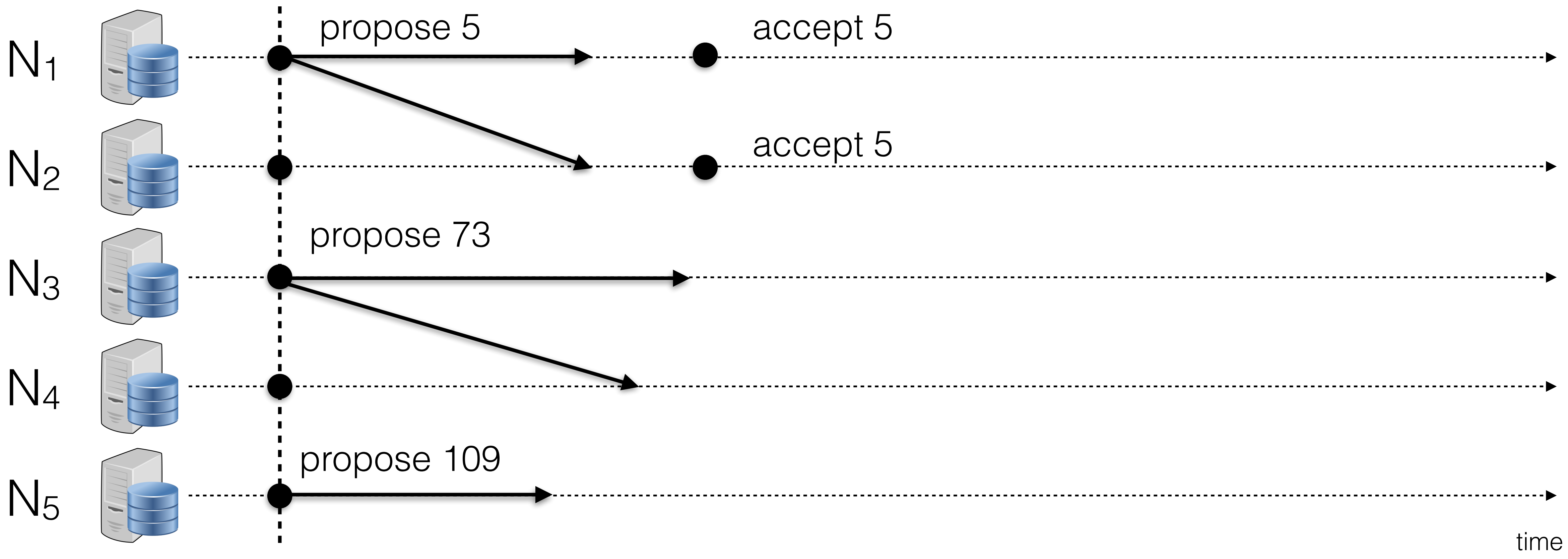
Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Split Votes



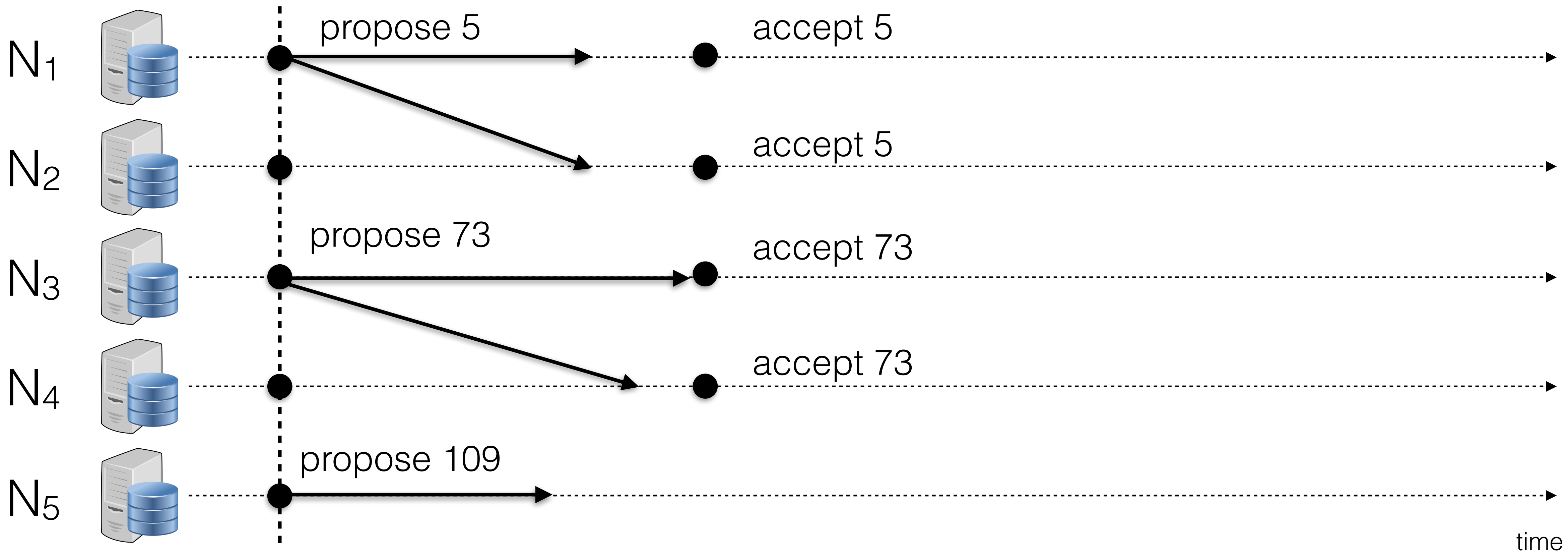
Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Split Votes



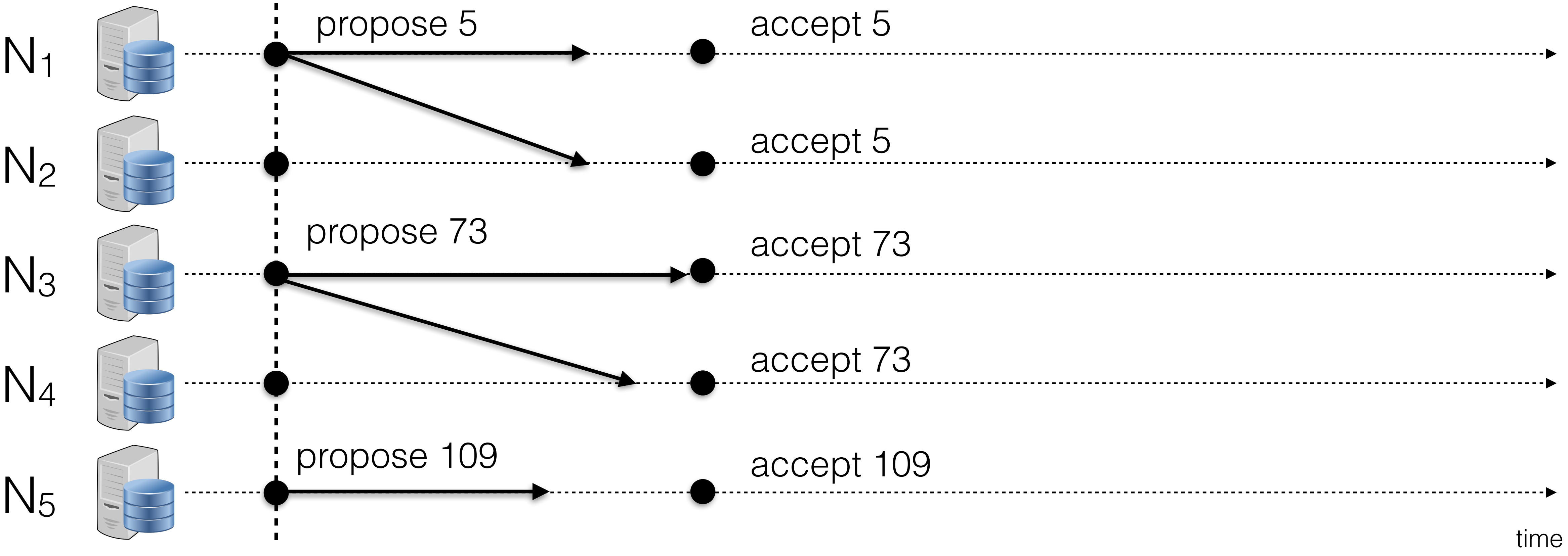
Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Split Votes



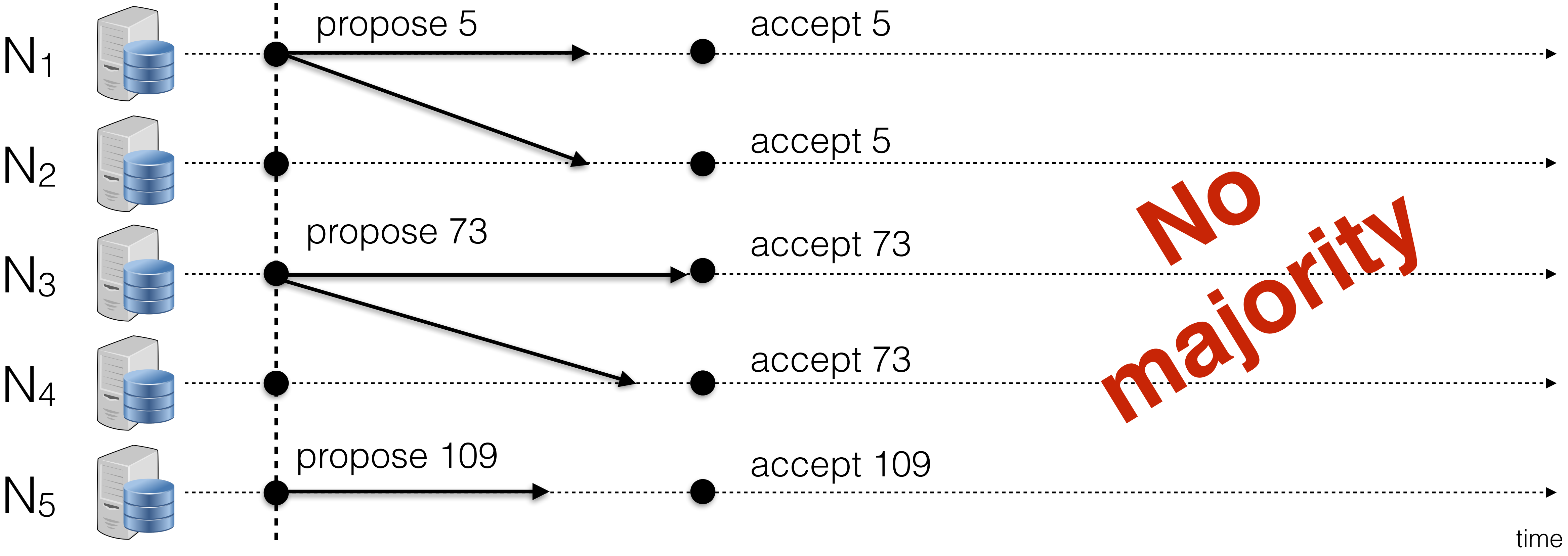
Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Split Votes



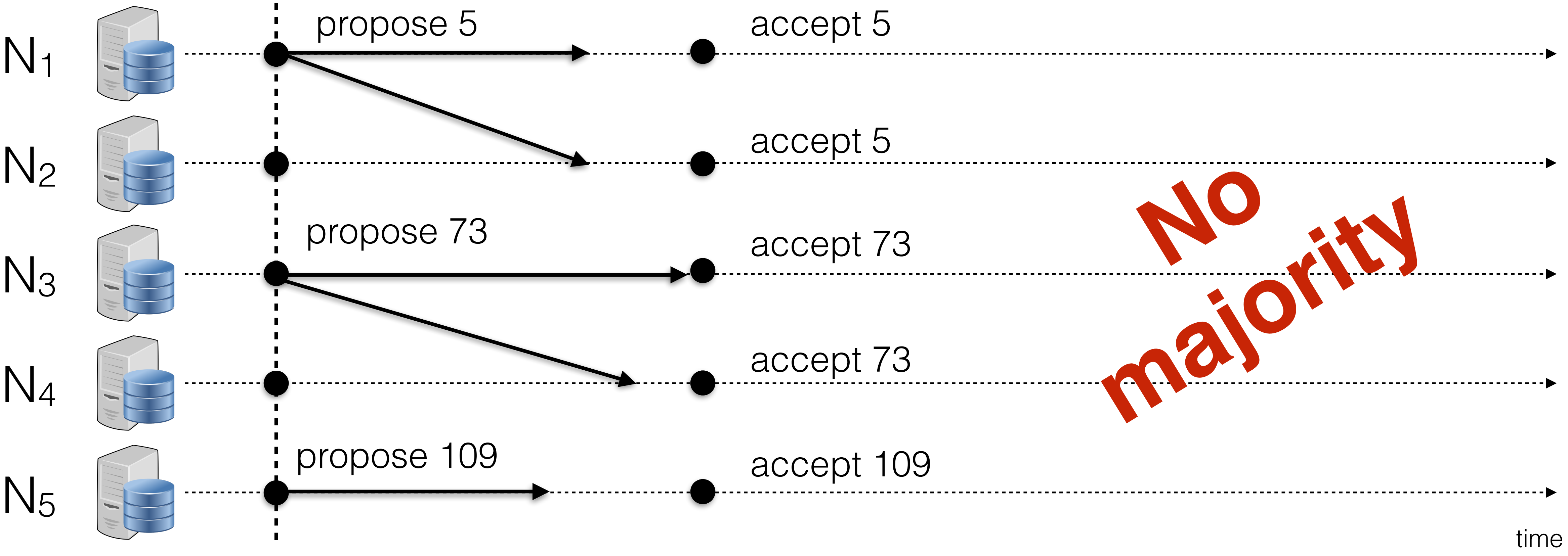
Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Split Votes



Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Split Votes

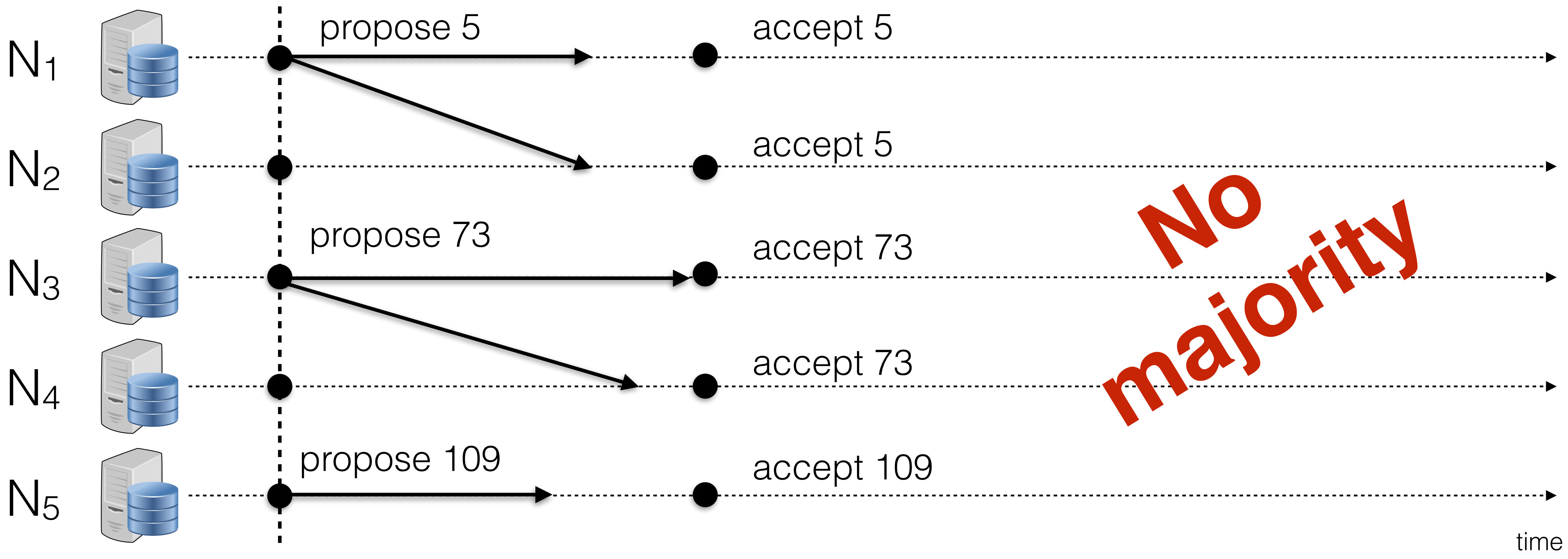


An acceptor must accept the first proposed value it receives

(and can accept more than one proposed value)

Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Split Votes



An acceptor must accept the first proposed value it receives

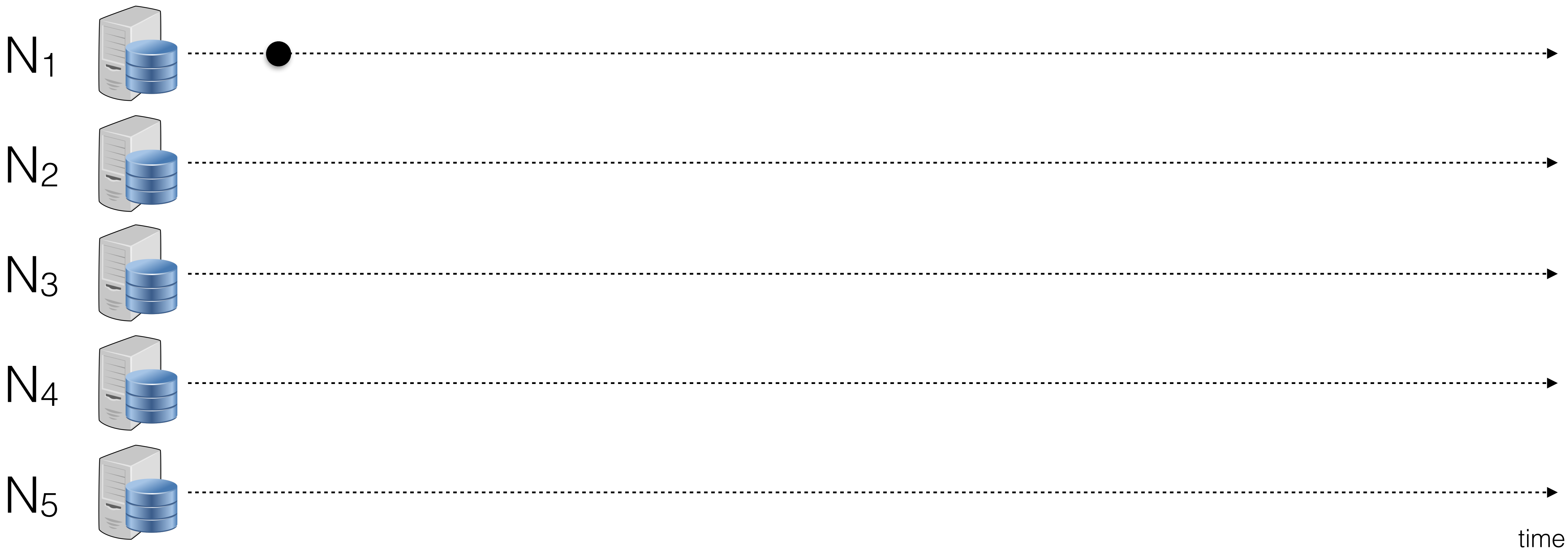
(and can accept more than one proposed value)

An acceptor can **accept** any number of proposed values, but an accepted proposed value may not necessarily be **chosen**.

Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

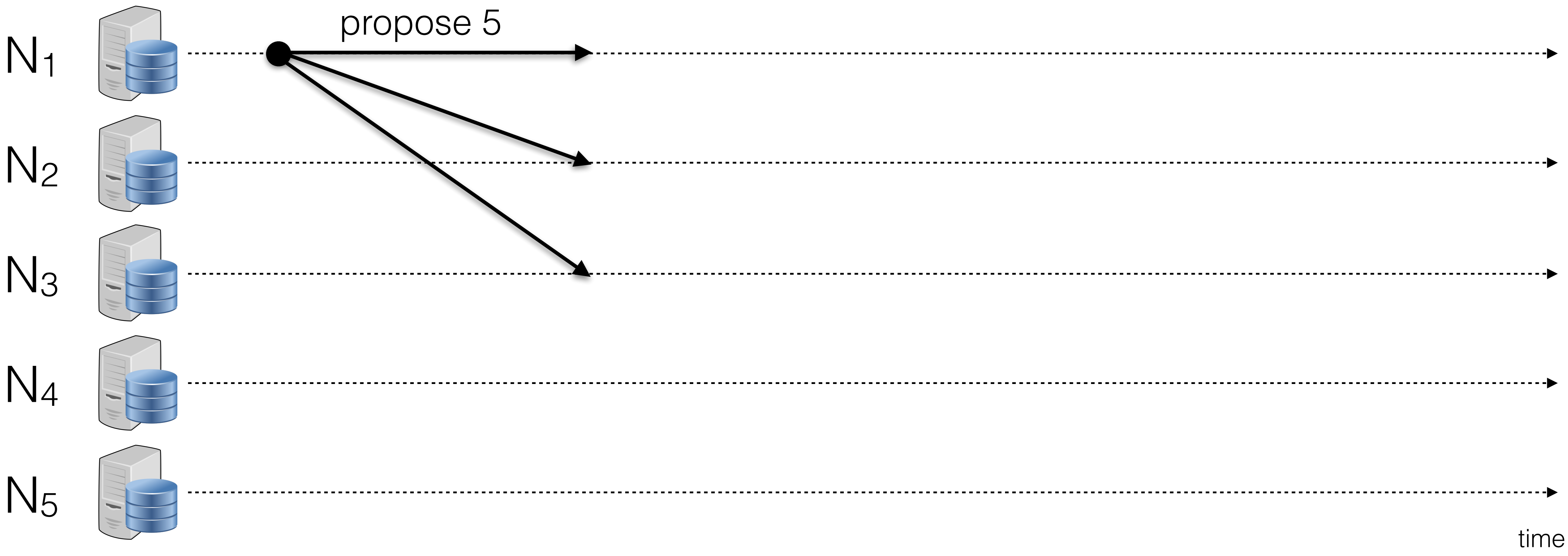
Should acceptors accept every proposed value? No



Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

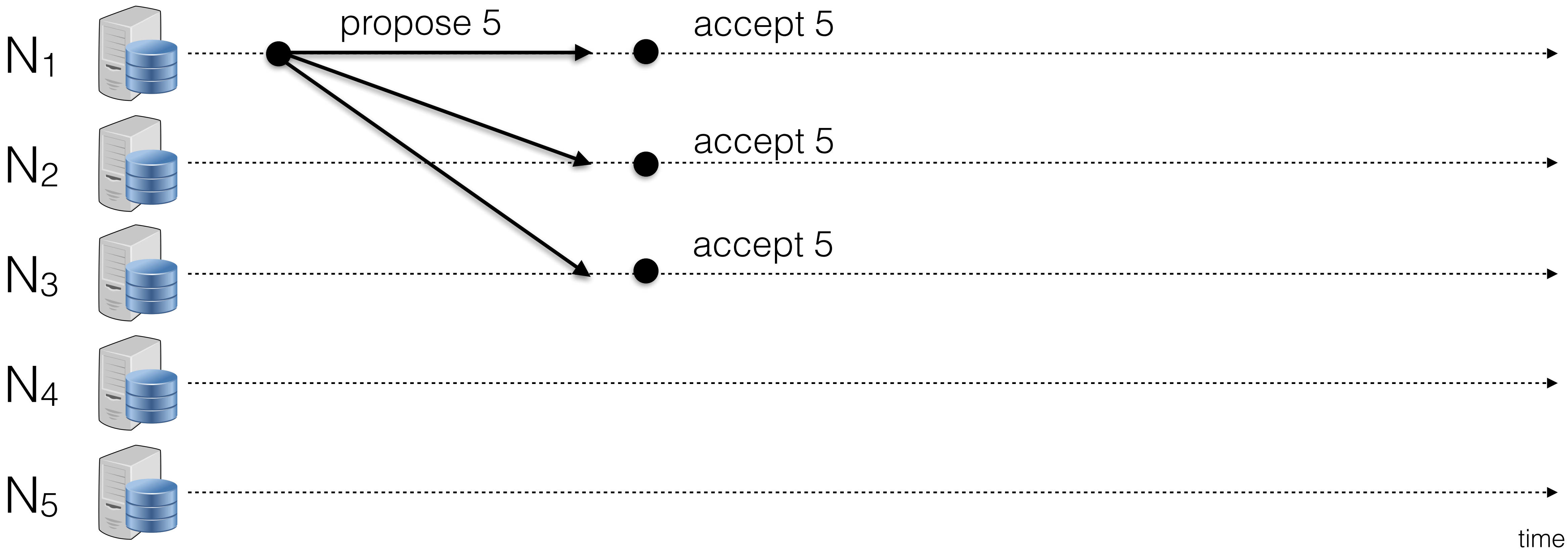
Should acceptors accept <u>every</u> proposed value? No



Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

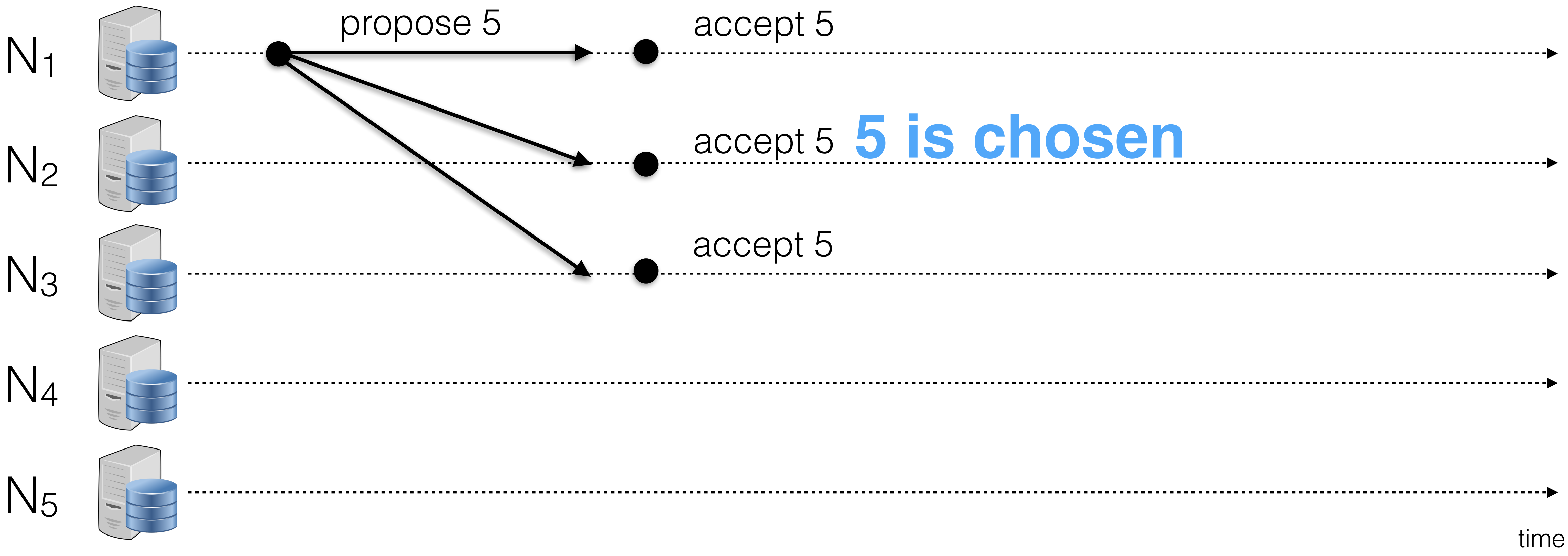
Should acceptors accept <u>every</u> proposed value? No



Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

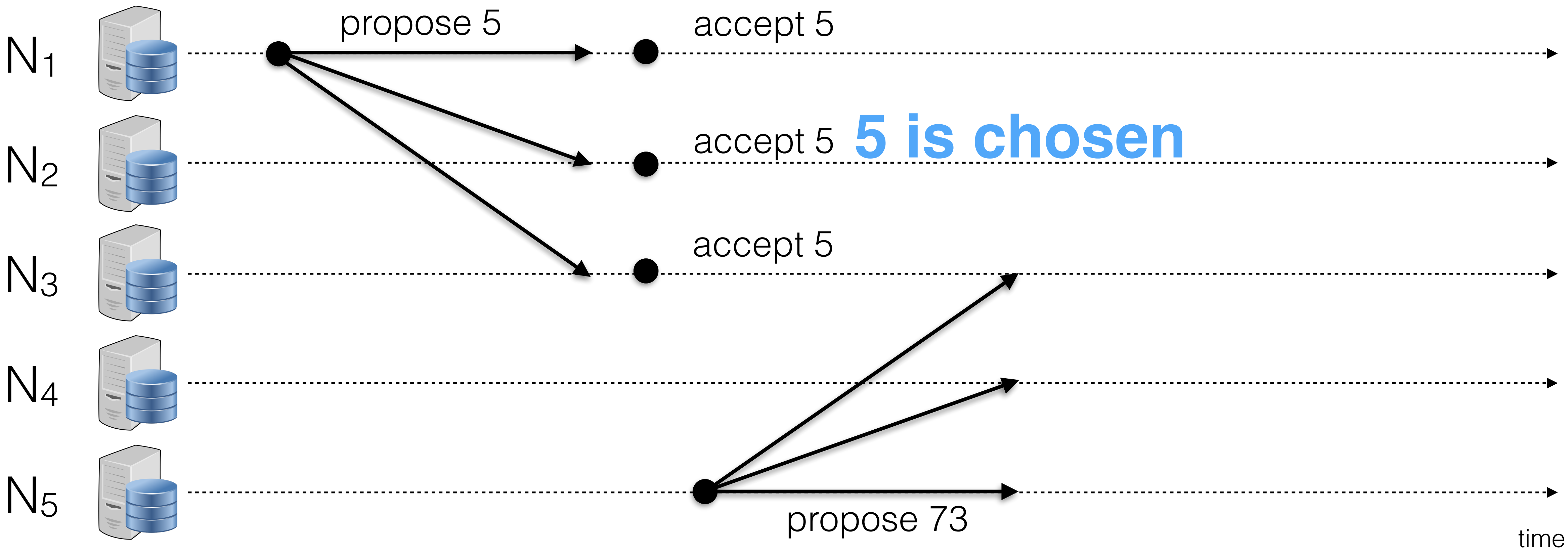
Should acceptors accept every proposed value? No



Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

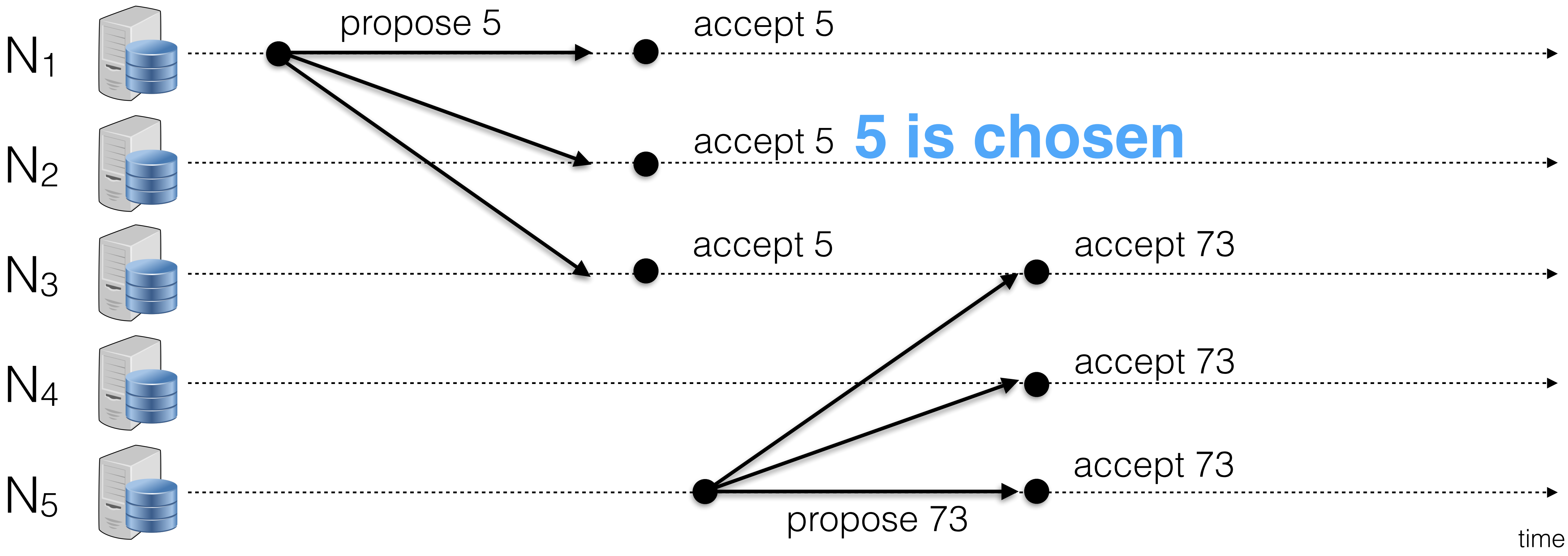
Should acceptors accept every proposed value? No



Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

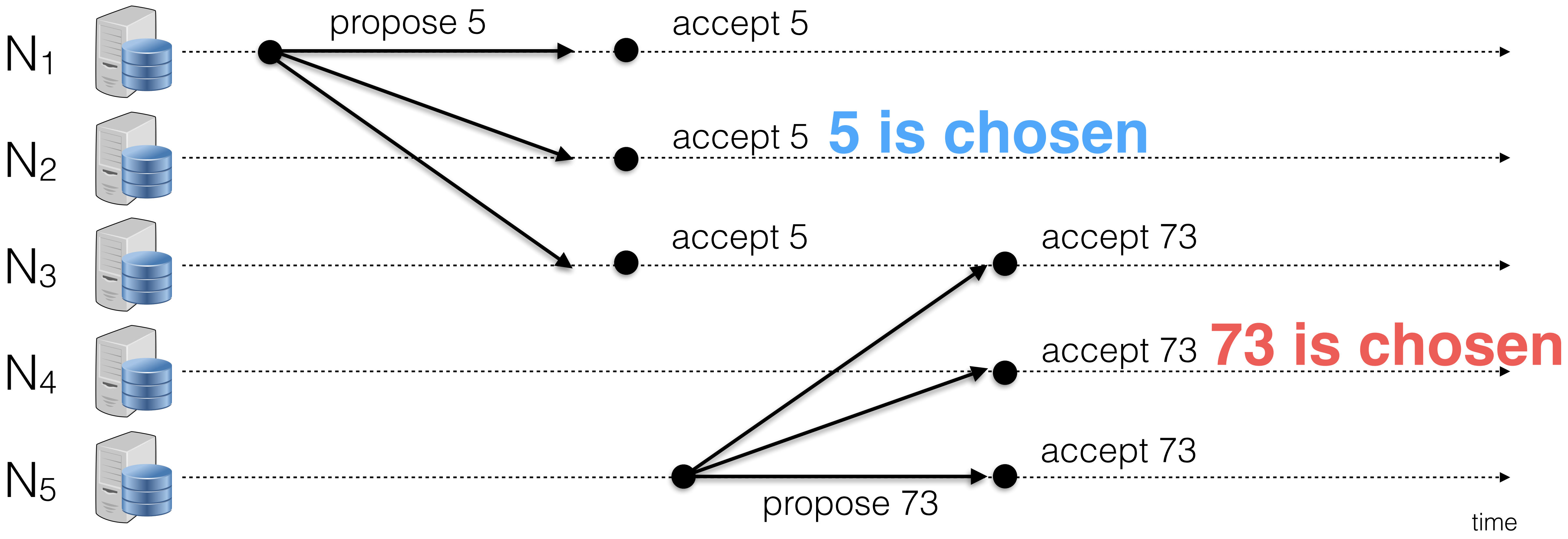
Should acceptors accept every proposed value? No



Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

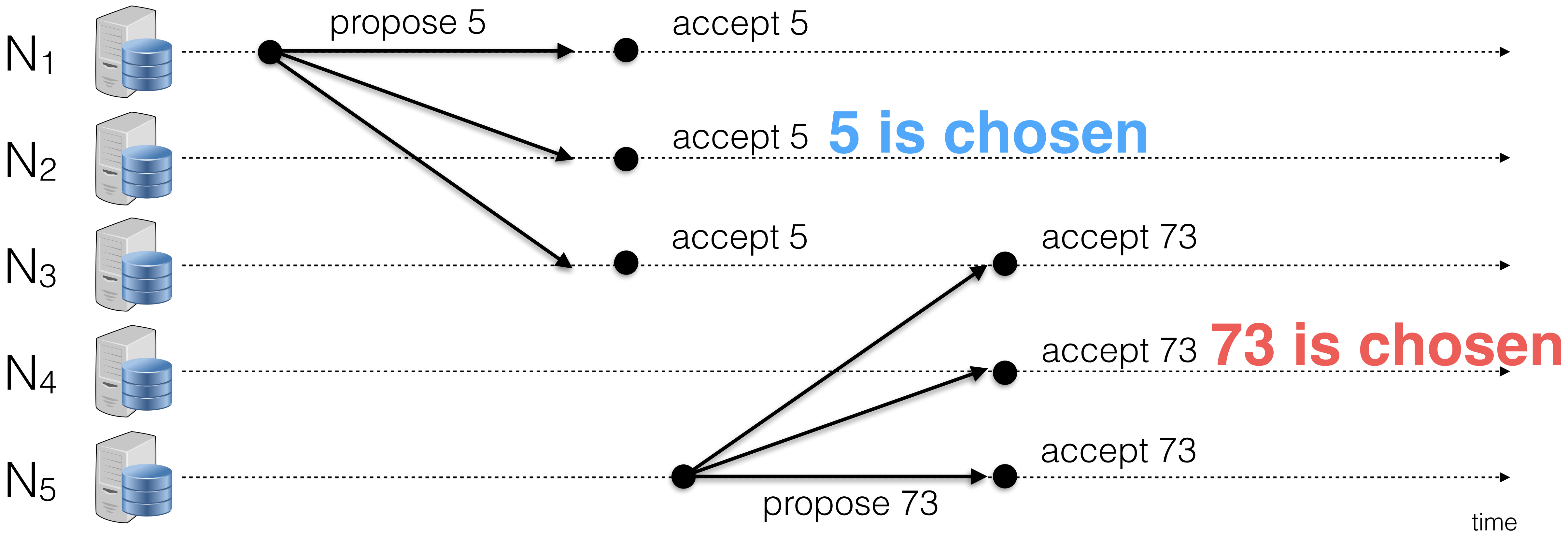
Should acceptors accept every proposed value? No



Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

Should acceptors accept <u>every</u> proposed value? No

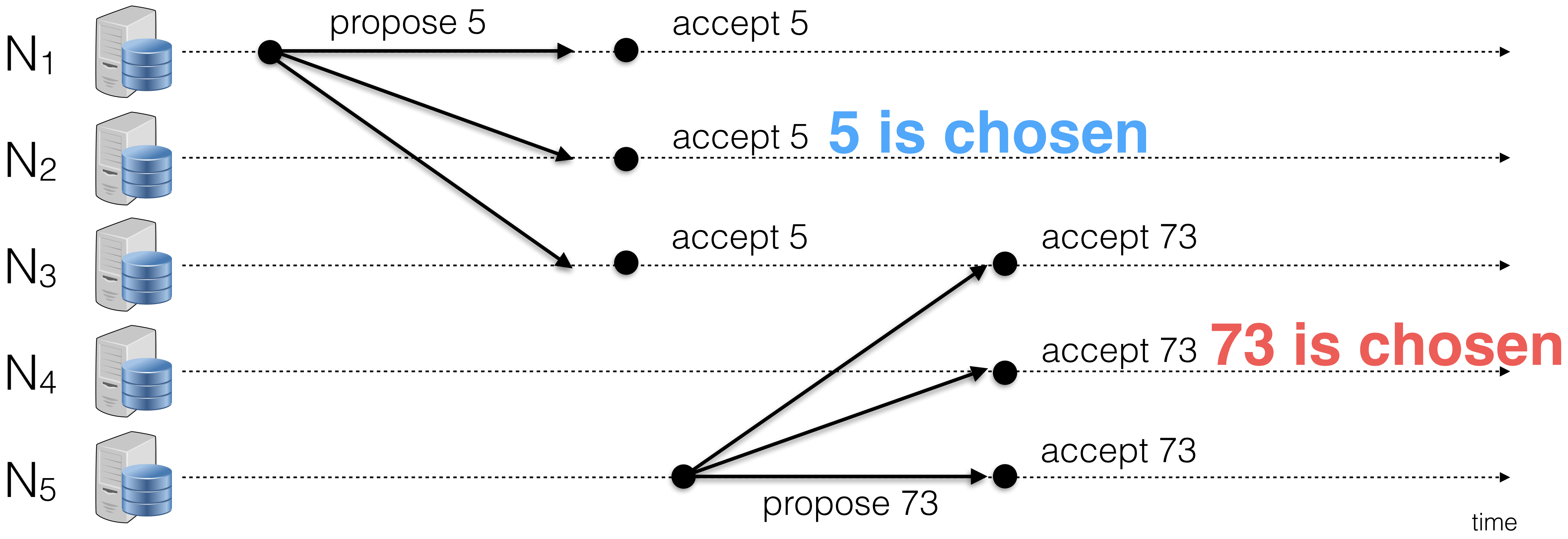


Once a value has been **chosen**, proposers must **propose** that same value

Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

Should acceptors accept <u>every</u> proposed value? No

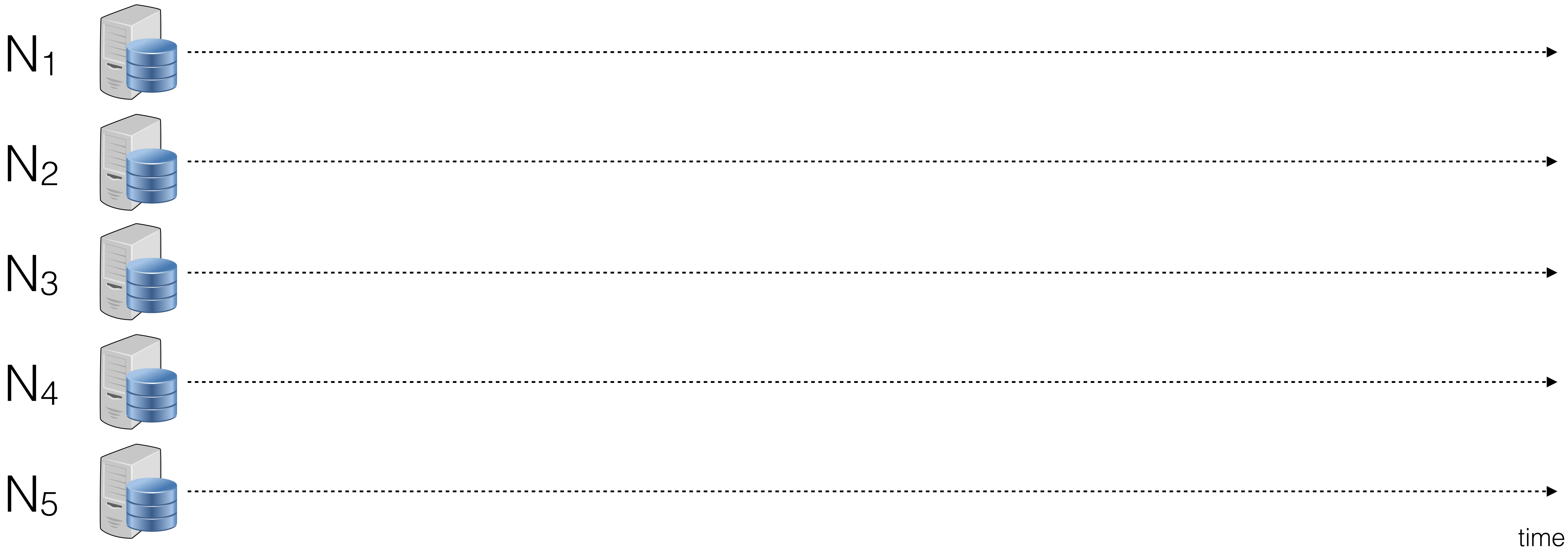


Once a value has been **chosen**, proposers must **propose** that same value

We need a two-phase protocol

Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

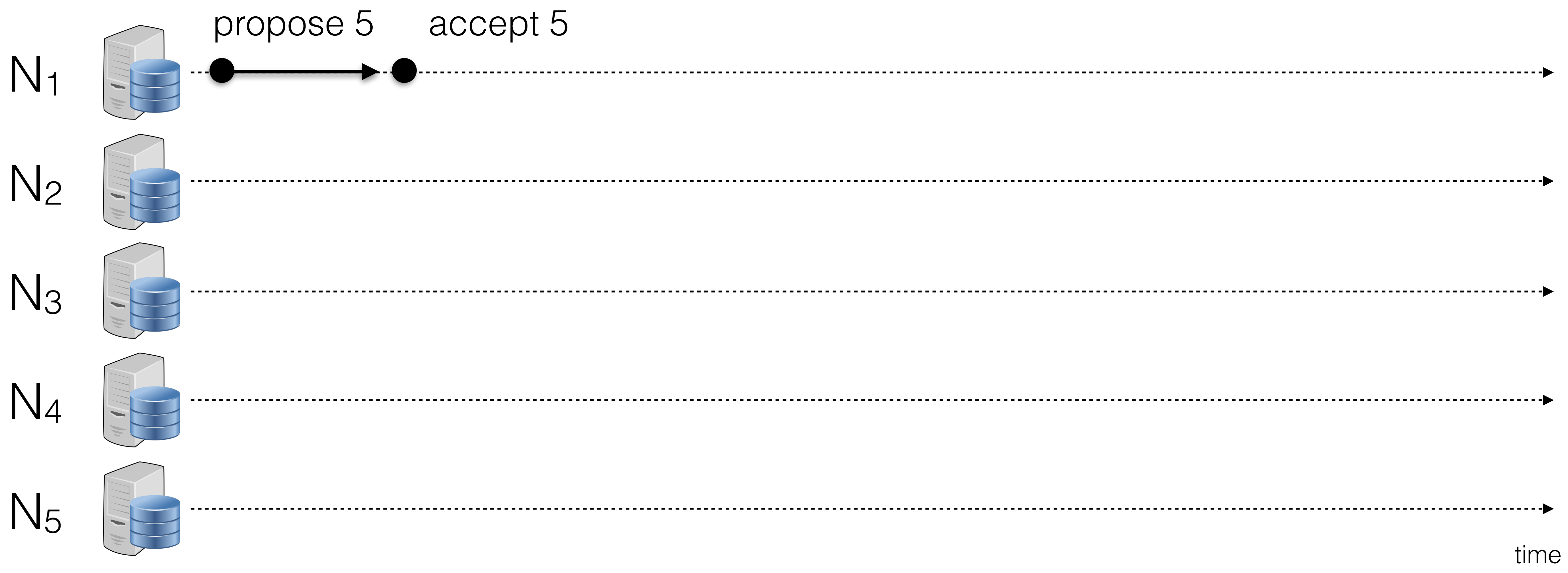


Once a value has been **chosen**, older proposed values must be ignored

We need a to order proposed values and reject old ones

Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

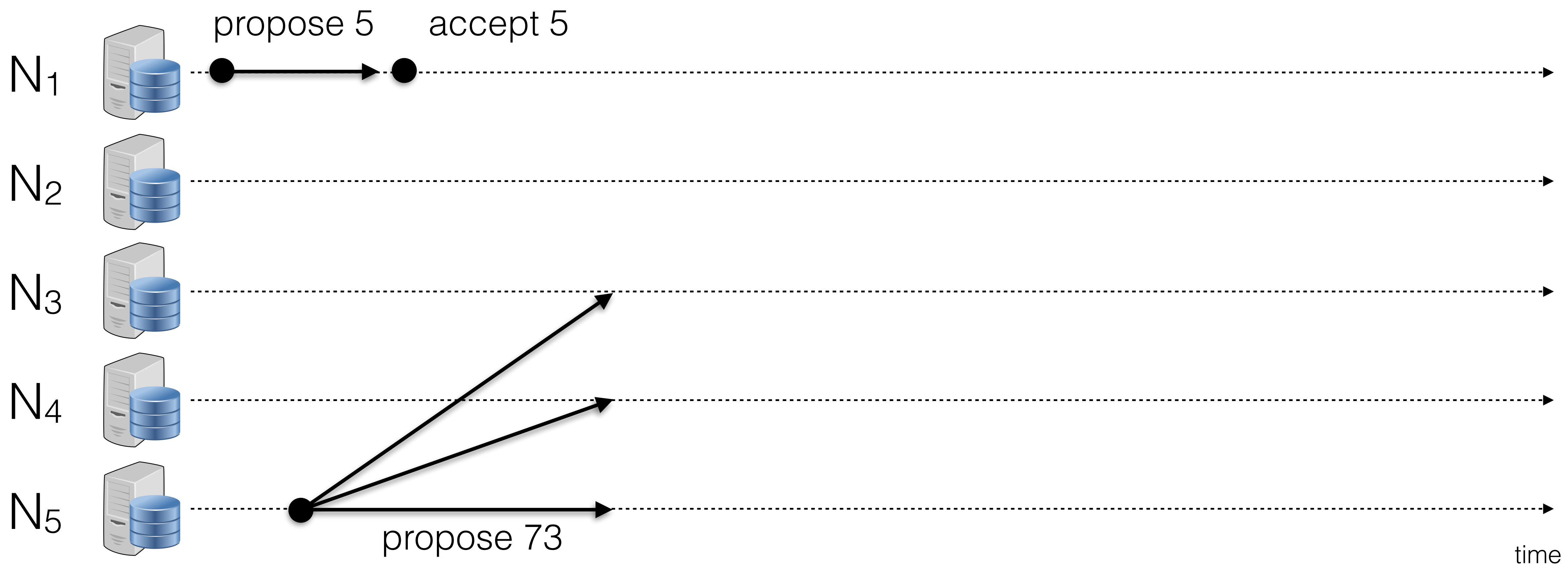


Once a value has been **chosen**, older proposed values must be ignored

We need a to order proposed values and reject old ones

Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

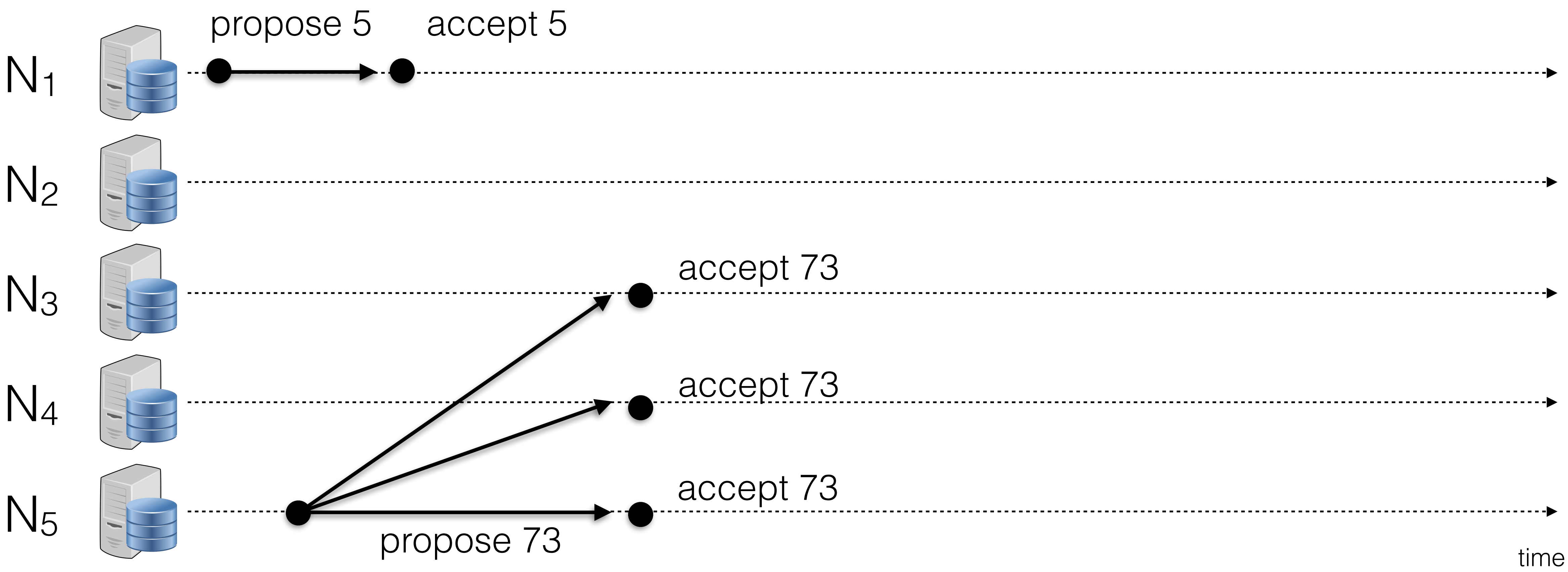


Once a value has been **chosen**, older proposed values must be ignored

We need a to order proposed values and reject old ones

Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

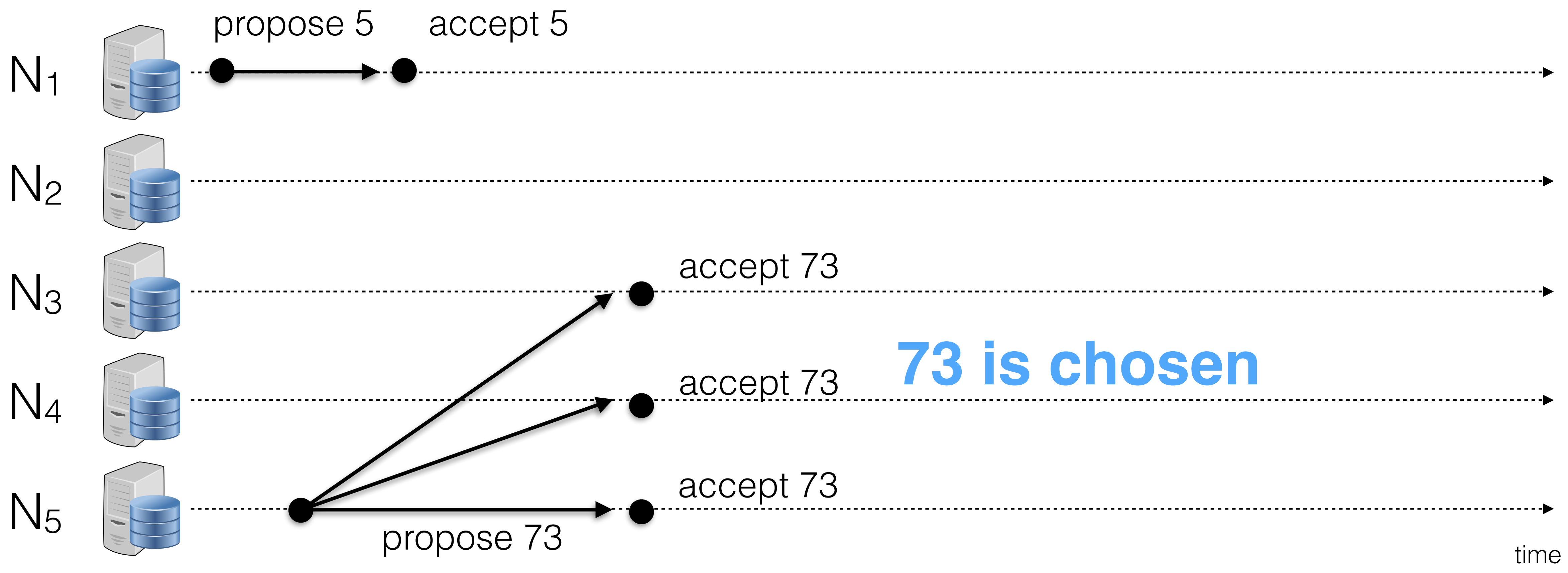


Once a value has been **chosen**, older proposed values must be ignored

We need a to order proposed values and reject old ones

Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices



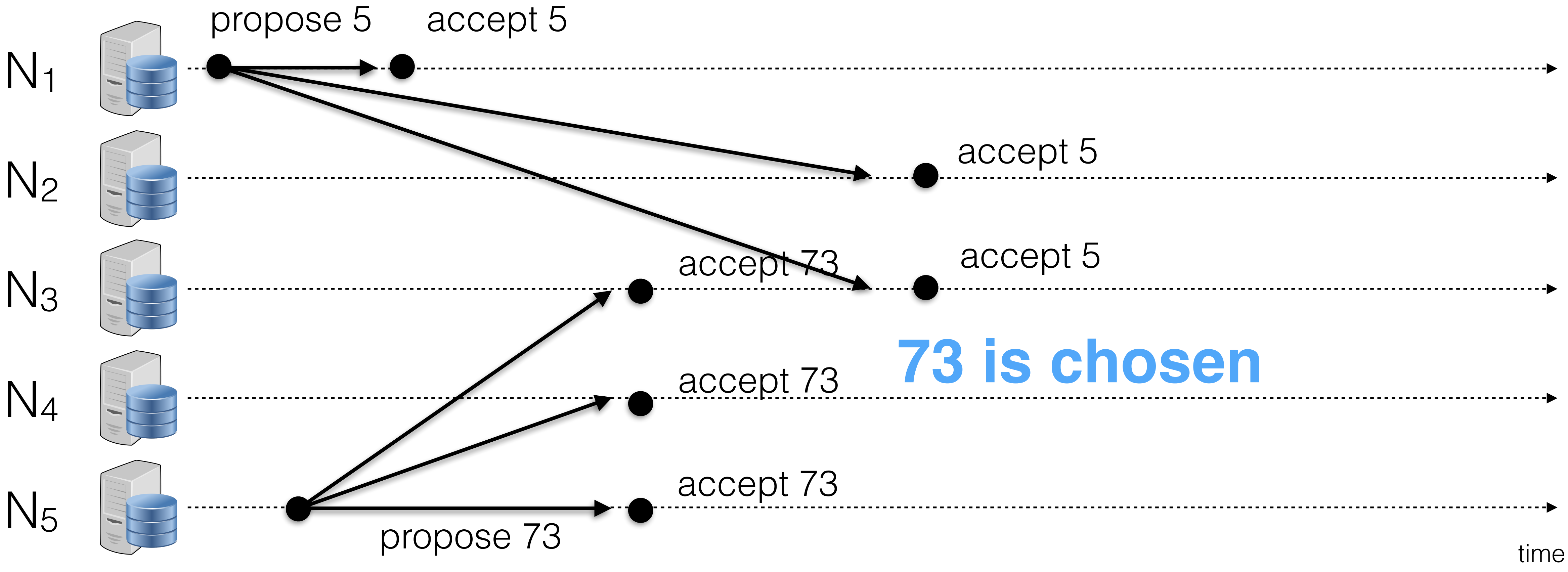
73 is chosen

Once a value has been **chosen**, older proposed values must be ignored

We need a to order proposed values and reject old ones

Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices

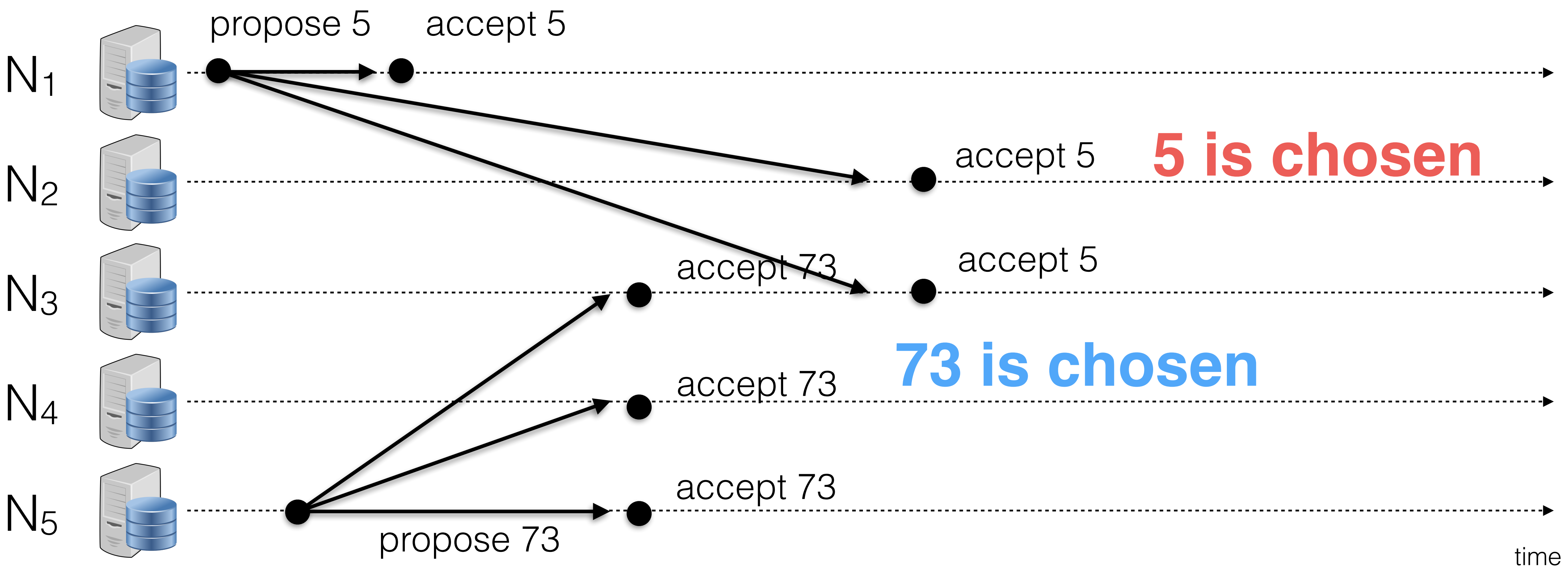


Once a value has been **chosen**, older proposed values must be ignored

We need a to order proposed values and reject old ones

Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Conflicting choices



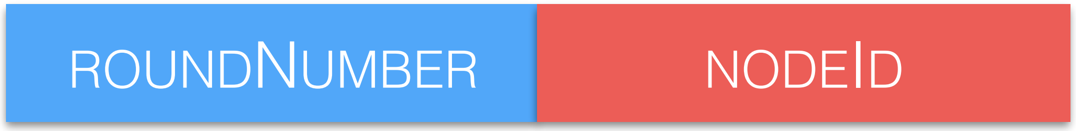
Once a value has been **chosen**, older proposed values must be ignored

We need a to order proposed values and reject old ones

Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Proposals

- A **proposal** *(v, n)* consists in the **proposed value** *v* and a **proposal number** *n*
- Whenever a proposer issues a new proposal, it chooses a strictly-increasing proposal number
- For a given proposer, a new proposal number must be greater than anything it has seen/used before
- Simple implementation



ROUNDNUMBER

NODEID

- Each server stores MAXROUND, i.e., the largest round number it has seen so far
- To generate a new proposal number: increment MAXROUND and concatenate with NODEID
- Proposers must persist MAXROUND on disk: must not reuse proposal numbers after crash/restart

Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Phase 1: PREPARE

- A proposer broadcasts a *prepare proposal (v, n)* with its own proposal value *v* and proposal number *n*
- An acceptor receives a *prepare proposal (v, n)*:
    - if it has never received a *prepare proposal*:
        - it *promises* to never accept *proposal numbers* lesser than n ($n_{min} = n$)
        - it returns ($\varnothing$, *0*);
    - otherwise it checks its *promise*:
        - if $n > n_{min}$ it sends its last *accepted proposal* (v_{last}, n_{last}); note that $n_{last} < n$.
        - otherwise it does nothing

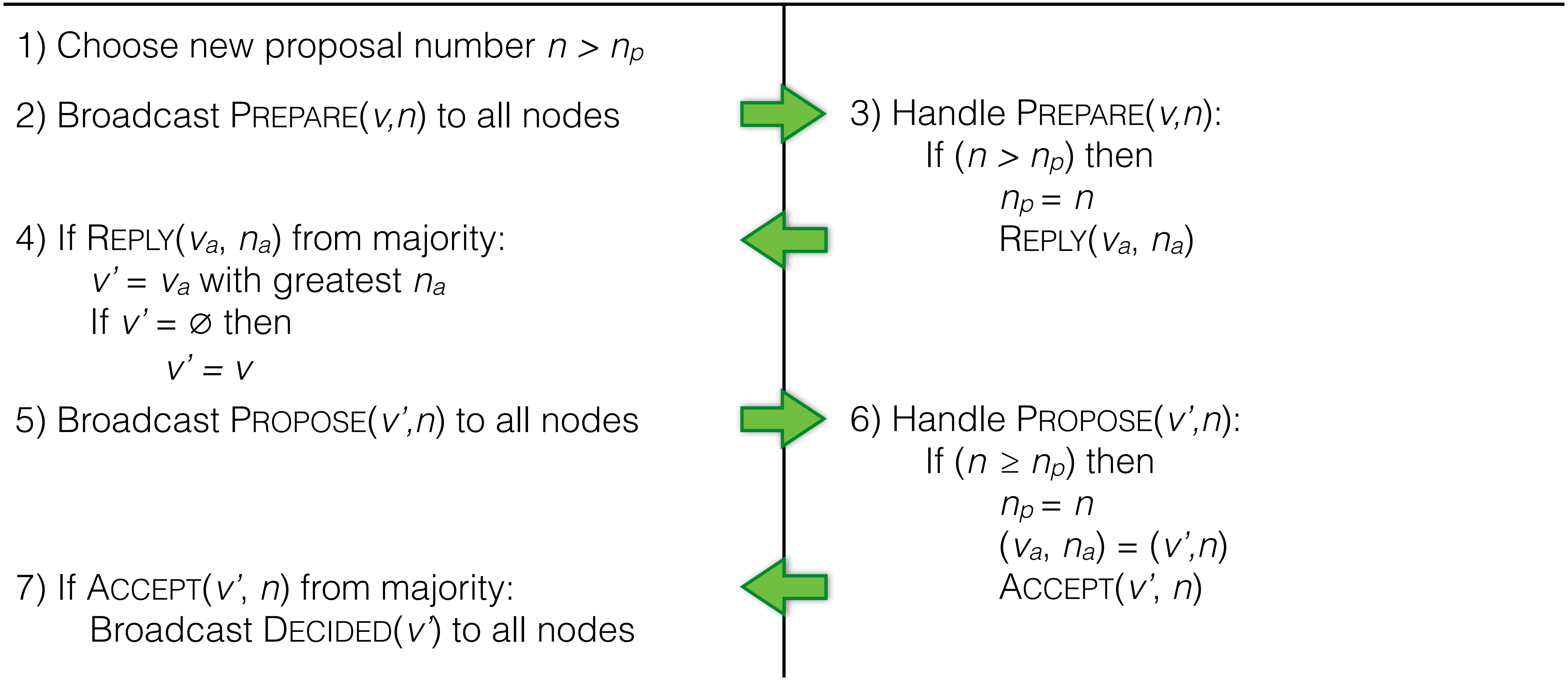
Phase 2: PROPOSE

- When a proposer receives a majority of responses, it broadcast to them a *propose proposal (v', n)*
  - If all acceptors returned (∅, *0*), *v'* is its own proposal value *v*
  - Otherwise *v'* is the v_{last} proposal value in the returned proposals with the greatest proposal number n_{last}
- If a majority of acceptors replies with ACK, the proposal is chosen
- Note that after a timeout, the proposer gives up and may send a new proposal

PAXOS Algorithm

Proposers | **Acceptors**

```
0) $n_p = 0$  highest prepare number seen
   $n_a = 0$  highest accepted proposal number
   $v_a = \varnothing$  highest accepted proposal value
```



```
1) Choose new proposal number $n > n_p$

2) Broadcast PREPARE($v,n$) to all nodes

3) Handle PREPARE($v,n$):
       If ($n > n_p$) then
           $n_p = n$
           REPLY($v_a$, $n_a$)

4) If REPLY($v_a$, $n_a$) from majority:
       $v' = v_a$ with greatest $n_a$
       If $v' = \varnothing$ then
           $v' = v$

5) Broadcast PROPOSE($v',n$) to all nodes

6) Handle PROPOSE($v',n$):
       If ($n \geq n_p$) then
           $n_p = n$
           $(v_a, n_a) = (v',n)$
           ACCEPT($v'$, $n$)

7) If ACCEPT($v'$, $n$) from majority:
       Broadcast DECIDED($v'$) to all nodes
```

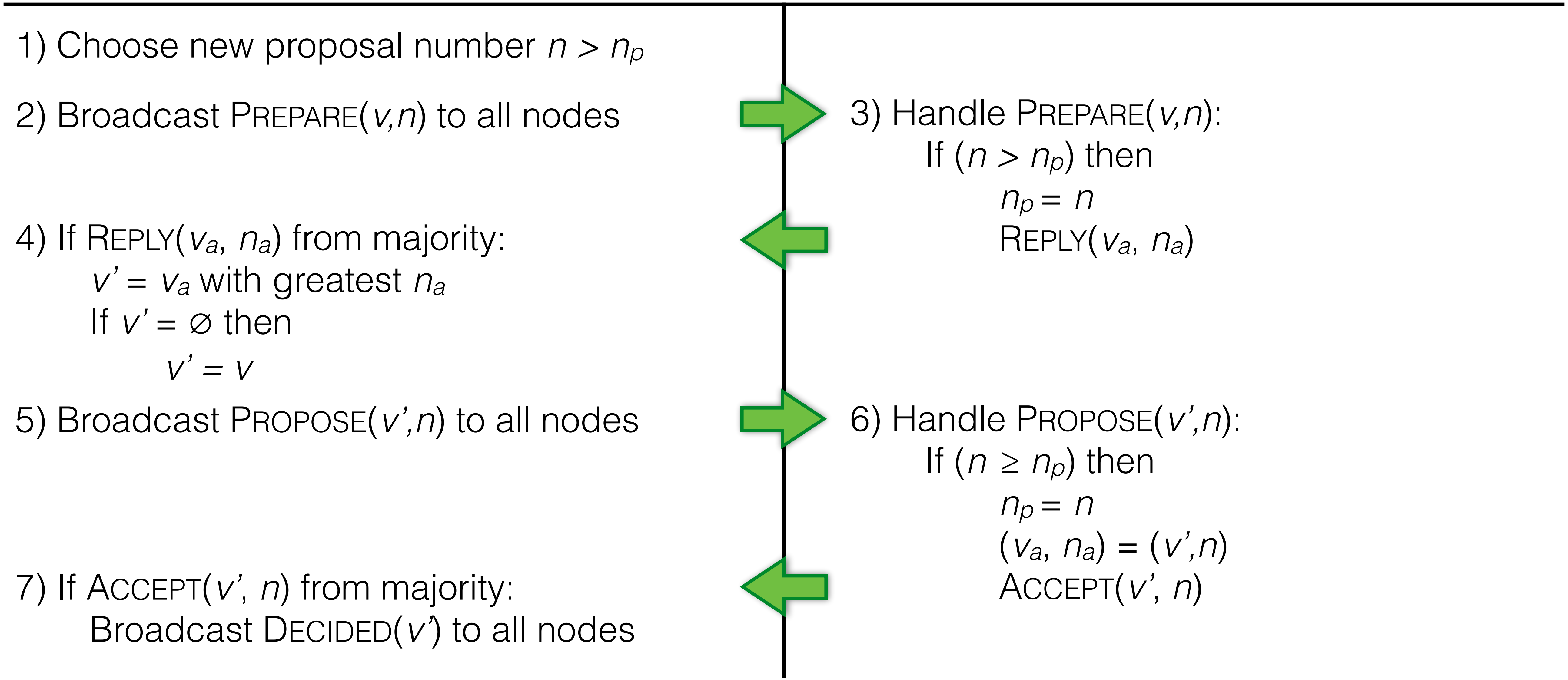
Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

PAXOS Algorithm

Proposers | **Acceptors**

```
0) $n_p = 0$  highest prepare number seen
   $n_a = 0$  highest accepted proposal number
   $v_a = \varnothing$  highest accepted proposal value
```

These value must stably persist on disk



Proposers:

```
1) Choose new proposal number $n > n_p$
2) Broadcast PREPARE($v,n$) to all nodes
4) If REPLY($v_a$, $n_a$) from majority:
       $v' = v_a$ with greatest $n_a$
       If $v' = \varnothing$ then
           $v' = v$
5) Broadcast PROPOSE($v',n$) to all nodes
7) If ACCEPT($v'$, $n$) from majority:
       Broadcast DECIDED($v'$) to all nodes
```

Acceptors:

```
3) Handle PREPARE($v,n$):
       If ($n > n_p$) then
           $n_p = n$
           REPLY($v_a$, $n_a$)
6) Handle PROPOSE($v',n$):
       If ($n \geq n_p$) then
           $n_p = n$
           ($v_a$, $n_a$) = ($v',n$)
           ACCEPT($v'$, $n$)
```

Slides based on https://ramcloud.stanford.edu/~ongaro/userstudy/

Learning a decision

- After a proposal is chosen, only the proposer knows about it!
- How do the other nodes get informed?
    1. The proposer could inform all nodes directly
        - If the proposer fails, the others are not informed (directly)...
    2. The acceptors could broadcast every time they accept a proposal
        - Much more fault-tolerant
        - Many accepted proposals may not be chosen...
        - Moreover, choosing a value costs $O(n^2)$ messages without failures!
    3. The proposer could inform some nodes directly
        - They will broadcast the decision to other nodes

PAXOS is safe!

If a proposal (v,n) is **chosen**, then for every proposed proposal (u,m) for which $m > n$ it holds that $v = u$

- Assume that there is a proposed proposal (u,m) for which $m > n$ and $u \neq v$. Consider such a proposal with the smallest m.

- Consider the non-empty intersection S of the two majority sets of nodes that are acceptors for (v,n) and (u,m) proposals.

- Since proposal (v,n) has been accepted and $m > n$, nodes in S must have received PREPARE(u,m) after (v,n) has been accepted, thus returning REPLY(v,n'), with $n \leq n' < m$.

- As a consequence, the proposer of (u,m) should propose (v, m), hence $u = v$, that is a contradiction.

PAXOS is correct!

If a value is chosen, all acceptors choose this value

- Once a proposal *(v,n)* is chosen, each following proposal *(u,m)* has the same proposal value, i.e., $u = v$, according to the previous theorem.

- Since every following proposal has the same value *v*, every proposal that is accepted after *(v,n)* is chosen will have the same proposal value *v*.

- Since no other value than *v* can be accepted, no other values can be chosen.

PAXOS is great?

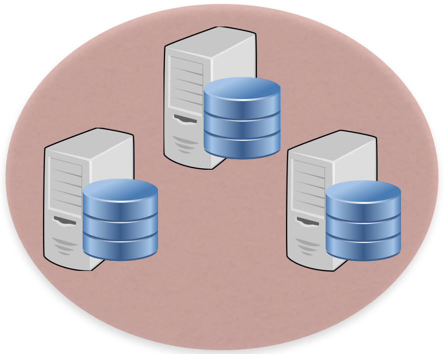
- PAXOS is a deterministic algorithm working for asynchronous systems and tolerating ***f < n/2*** failures.
- Many optimizations exists (Multi-PAXOS, Disk-PAXOS, RAFT)
- **FLP Theorem** [1985]. No totally correct consensus algorithm exists (for the given system model).
- PAXOS disproves the FLP Theorem?

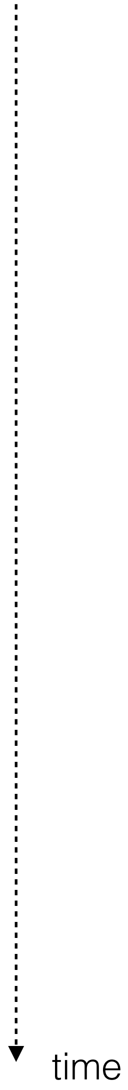
No Liveness Guarantees

- PAXOS only guarantees that if a value is chosen, the other nodes can only choose the same value
- PAXOS does not guarantee that a value is chosen!

No Liveness Guarantees

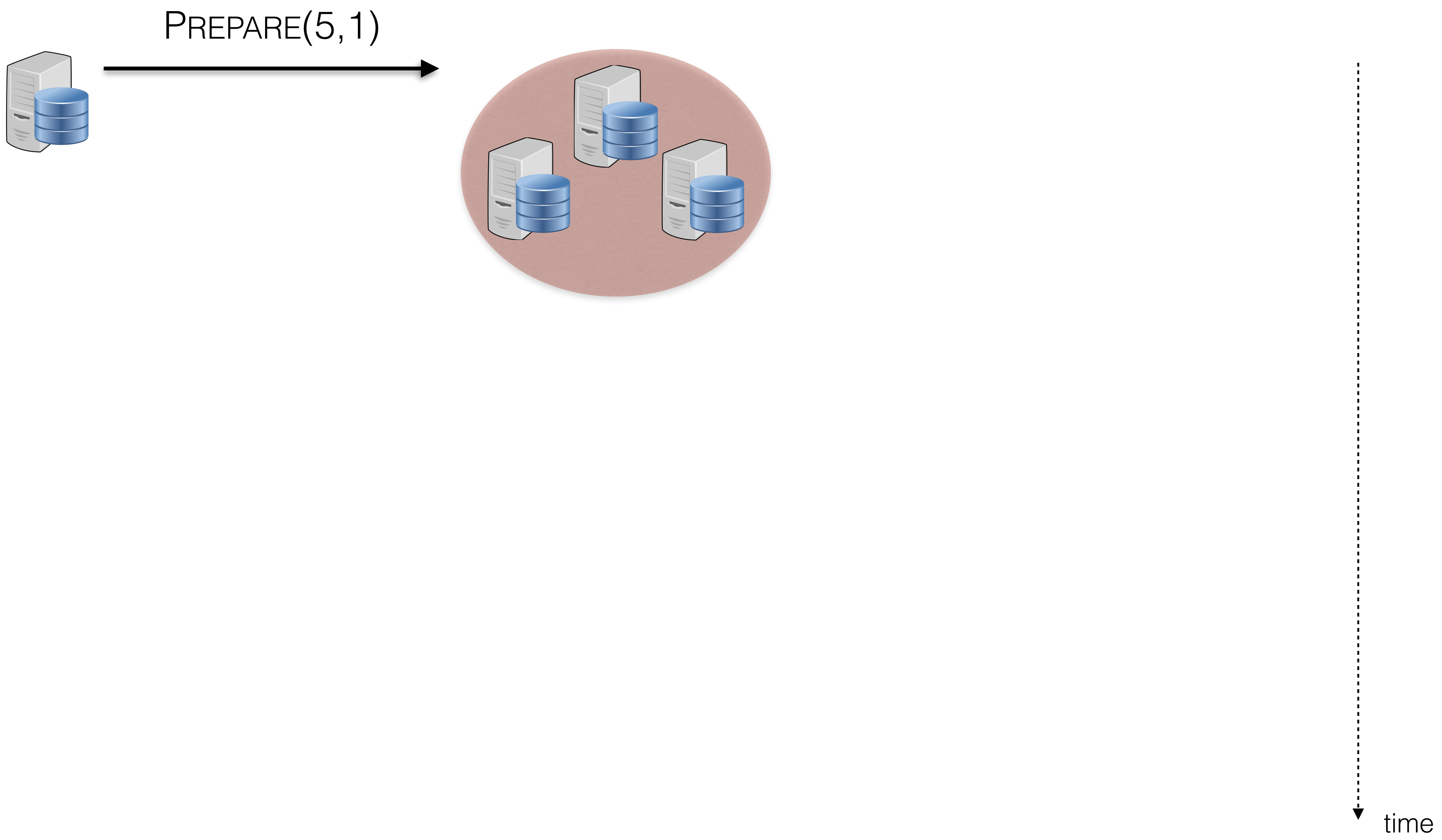
- PAXOS only guarantees that if a value is chosen, the other nodes can only choose the same value
- PAXOS does not guarantee that a value is chosen!





No Liveness Guarantees

- PAXOS only guarantees that if a value is chosen, the other nodes can only choose the same value
- PAXOS does not guarantee that a value is chosen!



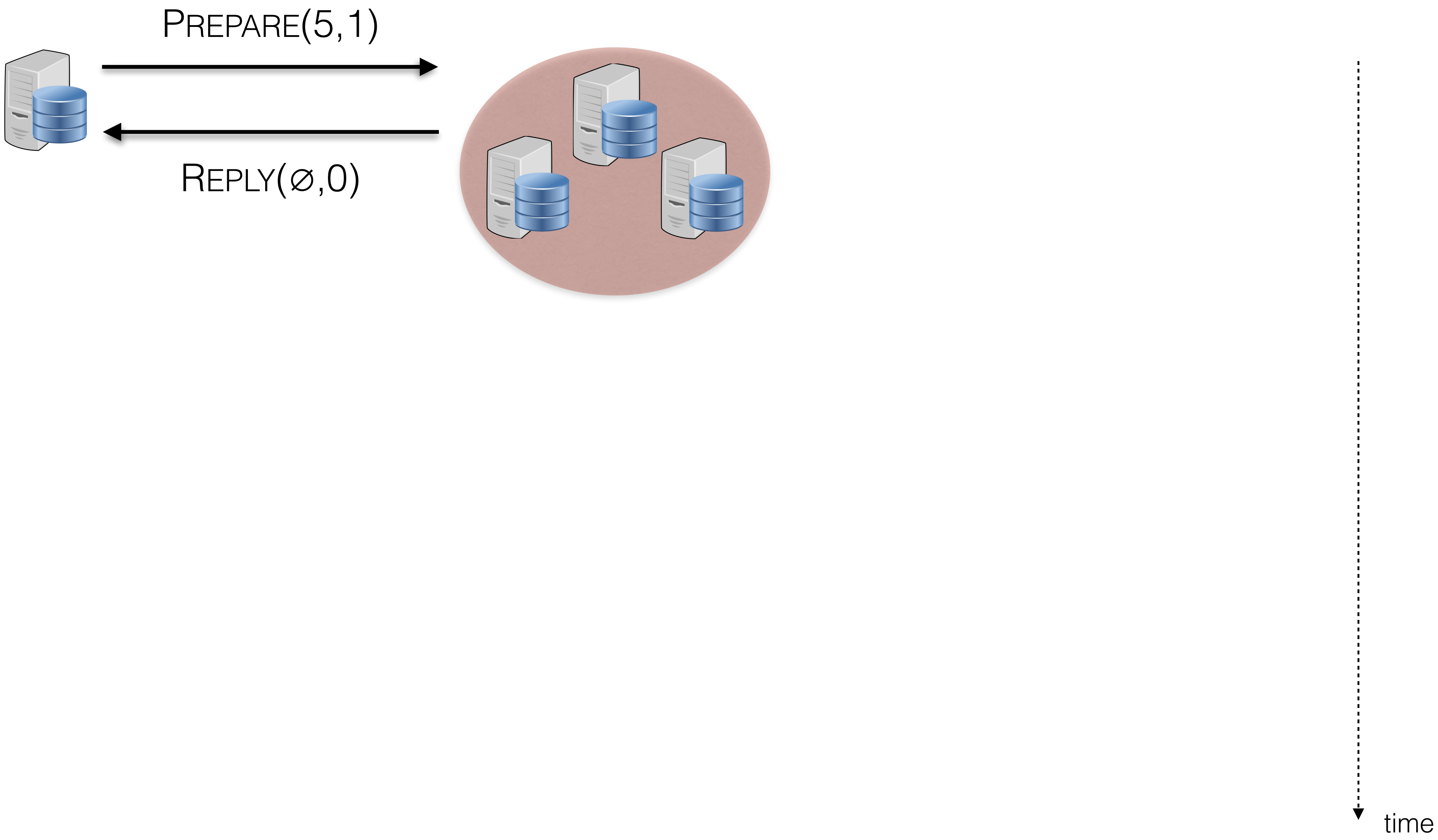
PREPARE(5,1)

time

No Liveness Guarantees

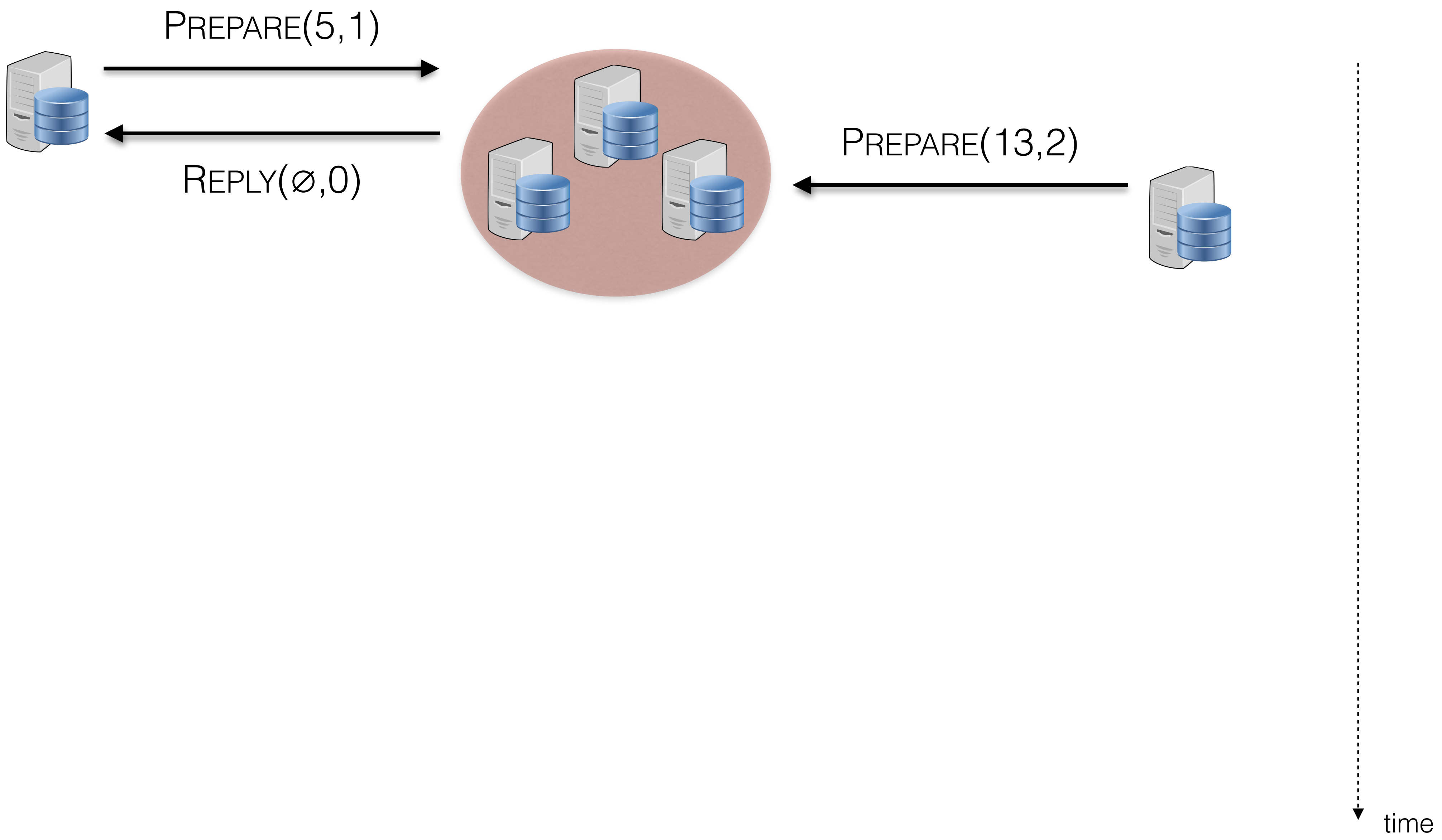
- PAXOS only guarantees that if a value is chosen, the other nodes can only choose the same value

- PAXOS does not guarantee that a value is chosen!



No Liveness Guarantees

- PAXOS only guarantees that if a value is chosen, the other nodes can only choose the same value
- PAXOS does not guarantee that a value is chosen!



PREPARE(5,1)

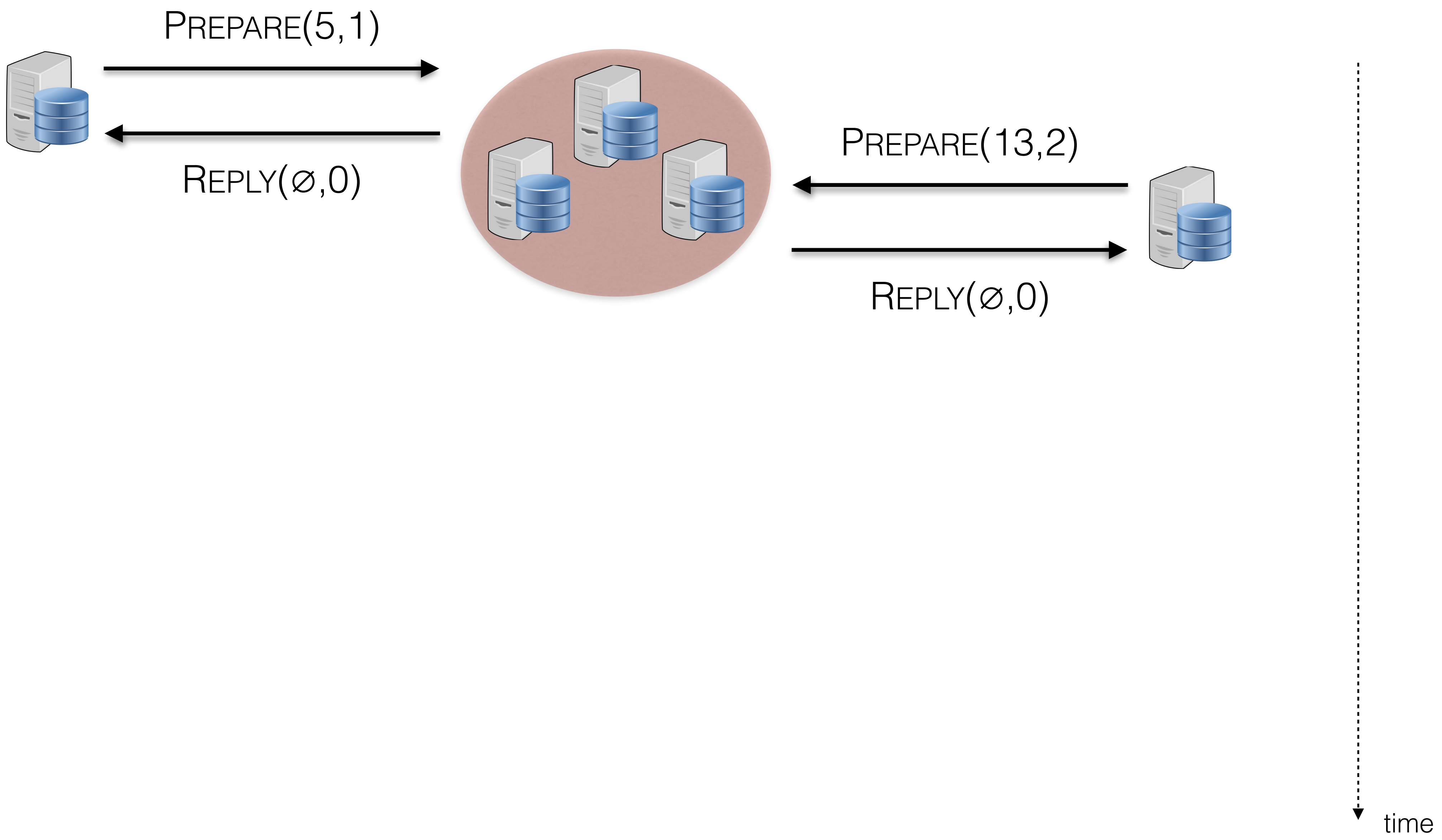
REPLY(∅,0)

PREPARE(13,2)

time

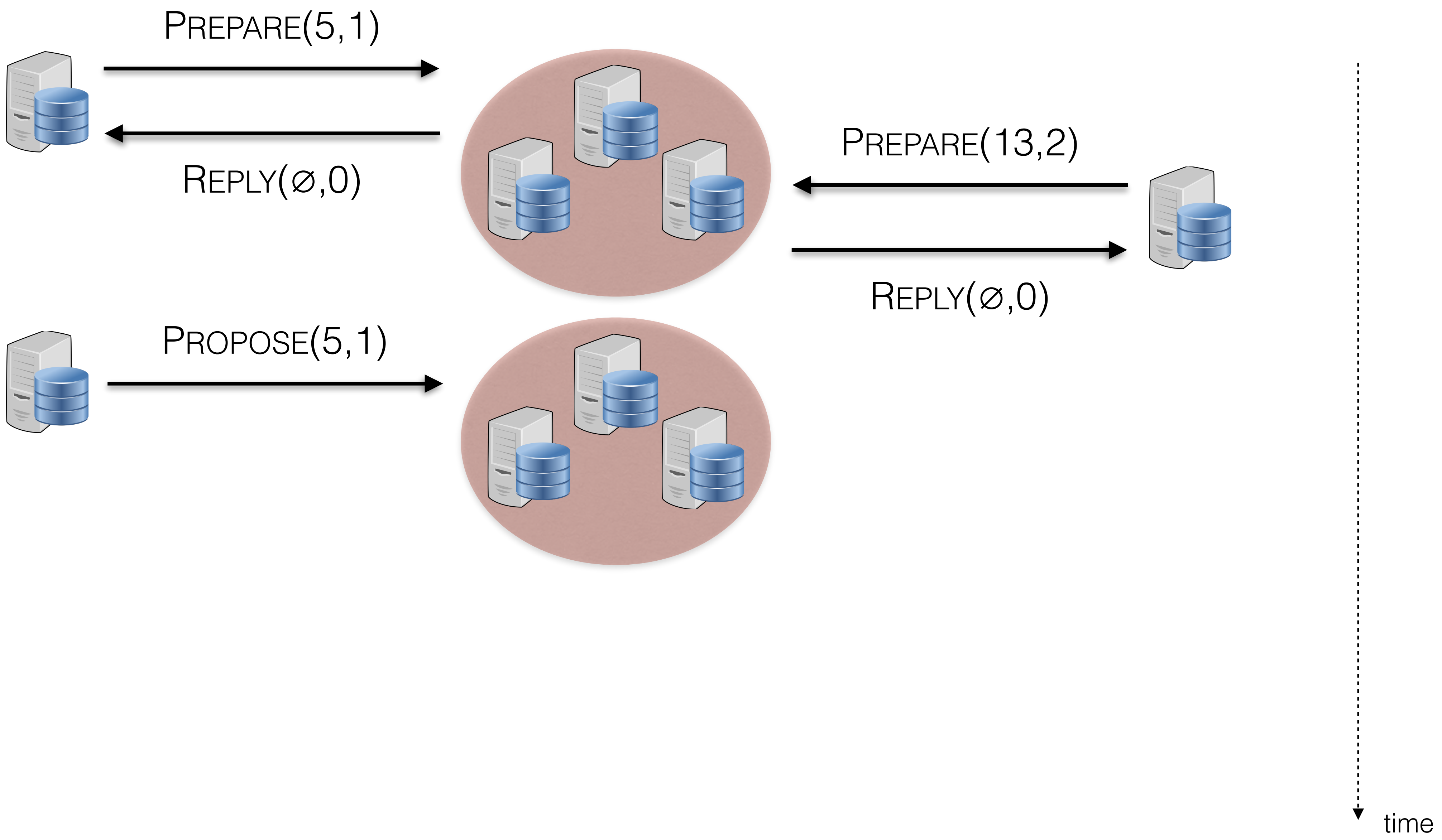
No Liveness Guarantees

- PAXOS only guarantees that if a value is chosen, the other nodes can only choose the same value
- PAXOS does not guarantee that a value is chosen!



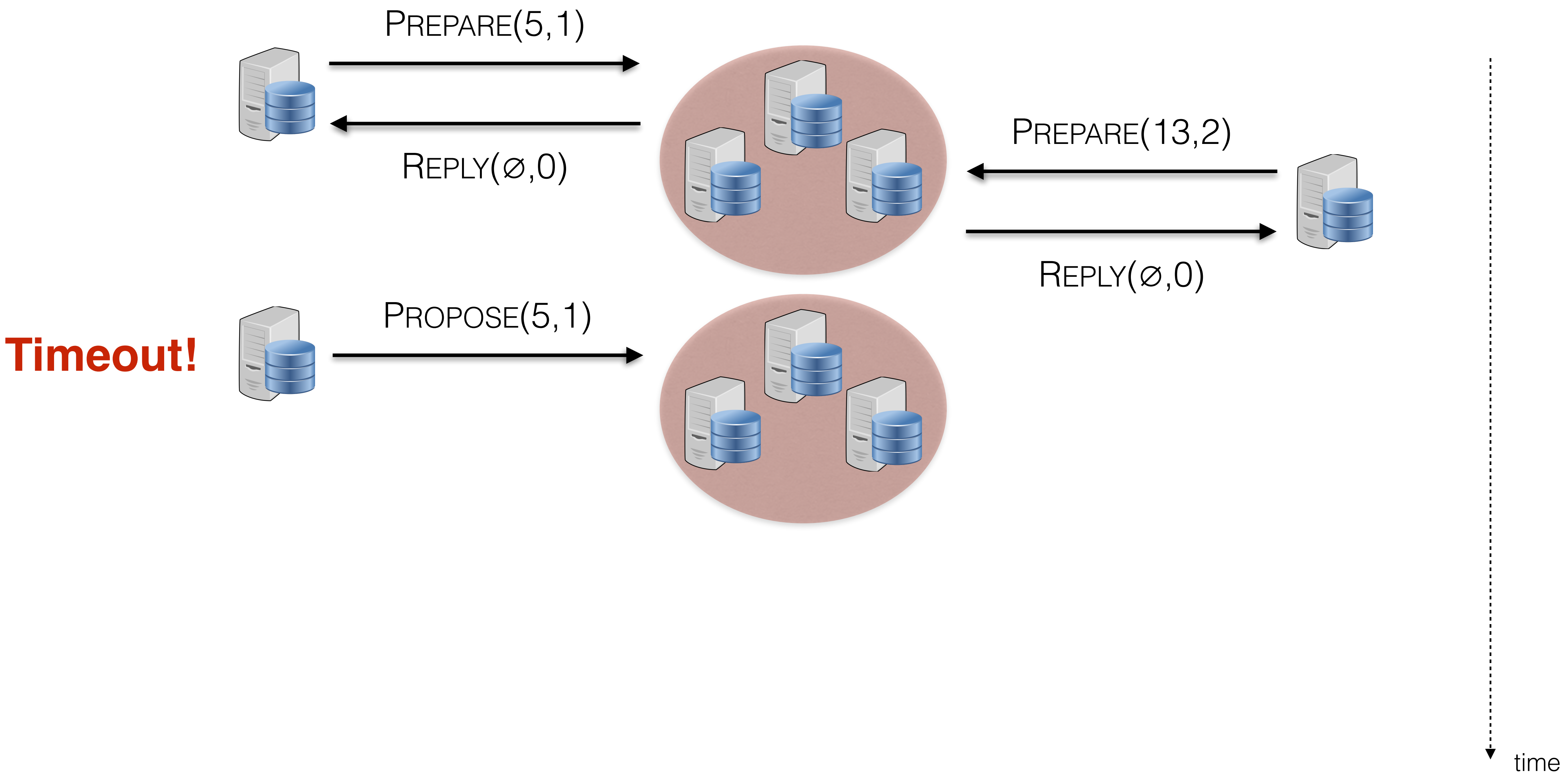
No Liveness Guarantees

- PAXOS only guarantees that if a value is chosen, the other nodes can only choose the same value
- PAXOS does not guarantee that a value is chosen!



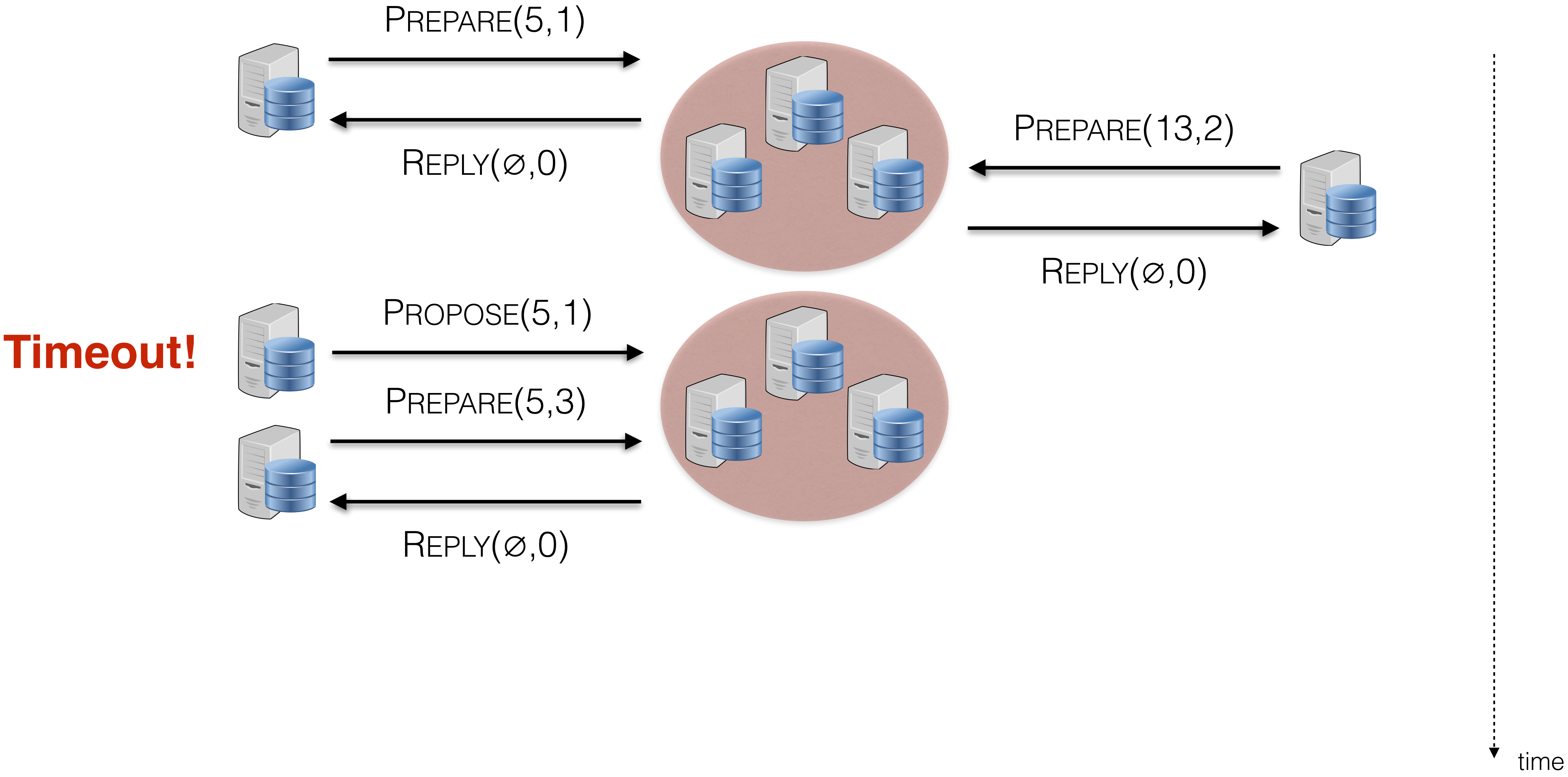
No Liveness Guarantees

- PAXOS only guarantees that if a value is chosen, the other nodes can only choose the same value
- PAXOS does not guarantee that a value is chosen!



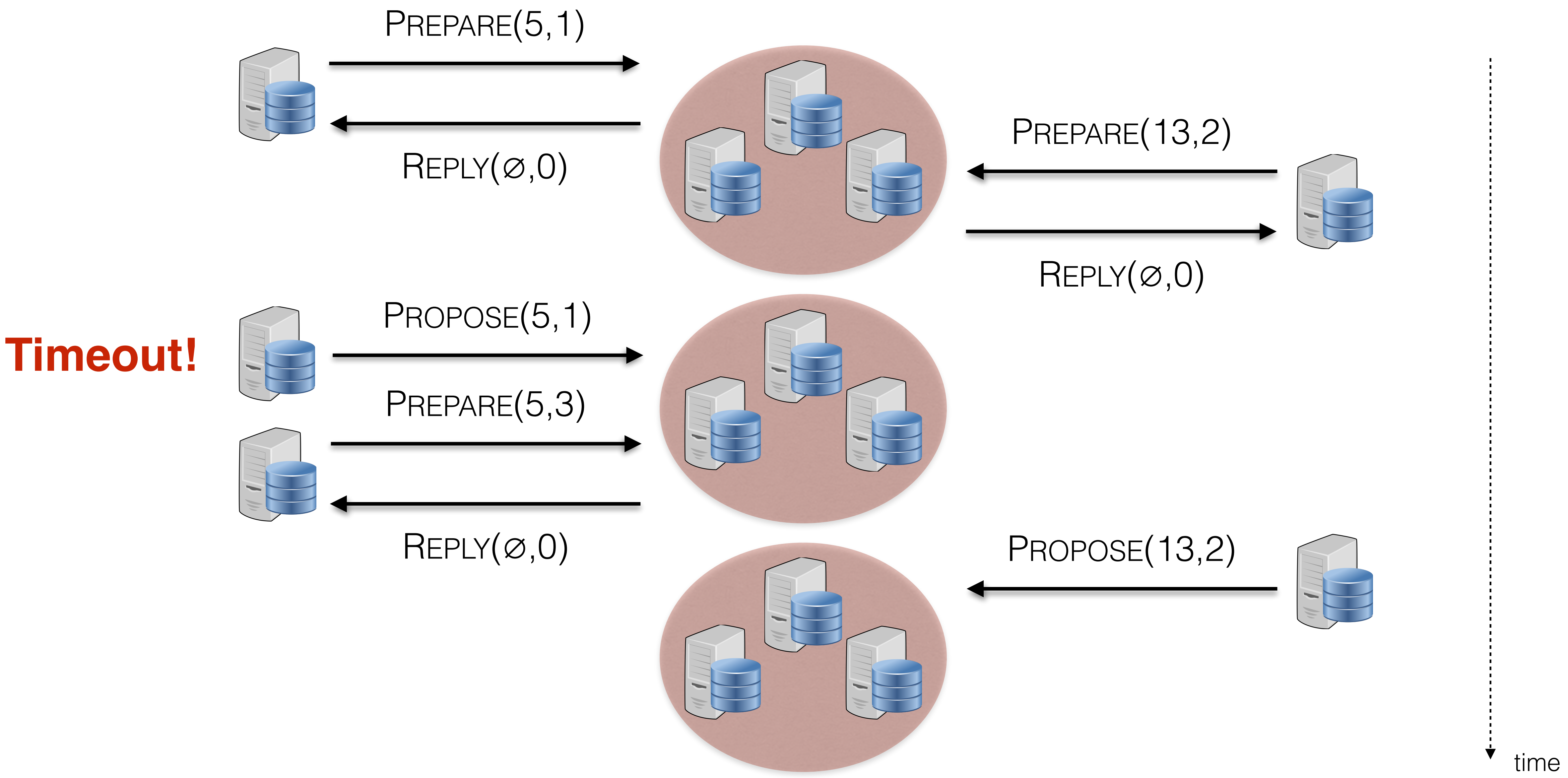
No Liveness Guarantees

- PAXOS only guarantees that if a value is chosen, the other nodes can only choose the same value
- PAXOS does not guarantee that a value is chosen!



No Liveness Guarantees

- PAXOS only guarantees that if a value is chosen, the other nodes can only choose the same value
- PAXOS does not guarantee that a value is chosen!



PREPARE(5,1)

REPLY(∅,0)

PREPARE(13,2)

REPLY(∅,0)

Timeout!

PROPOSE(5,1)

PREPARE(5,3)

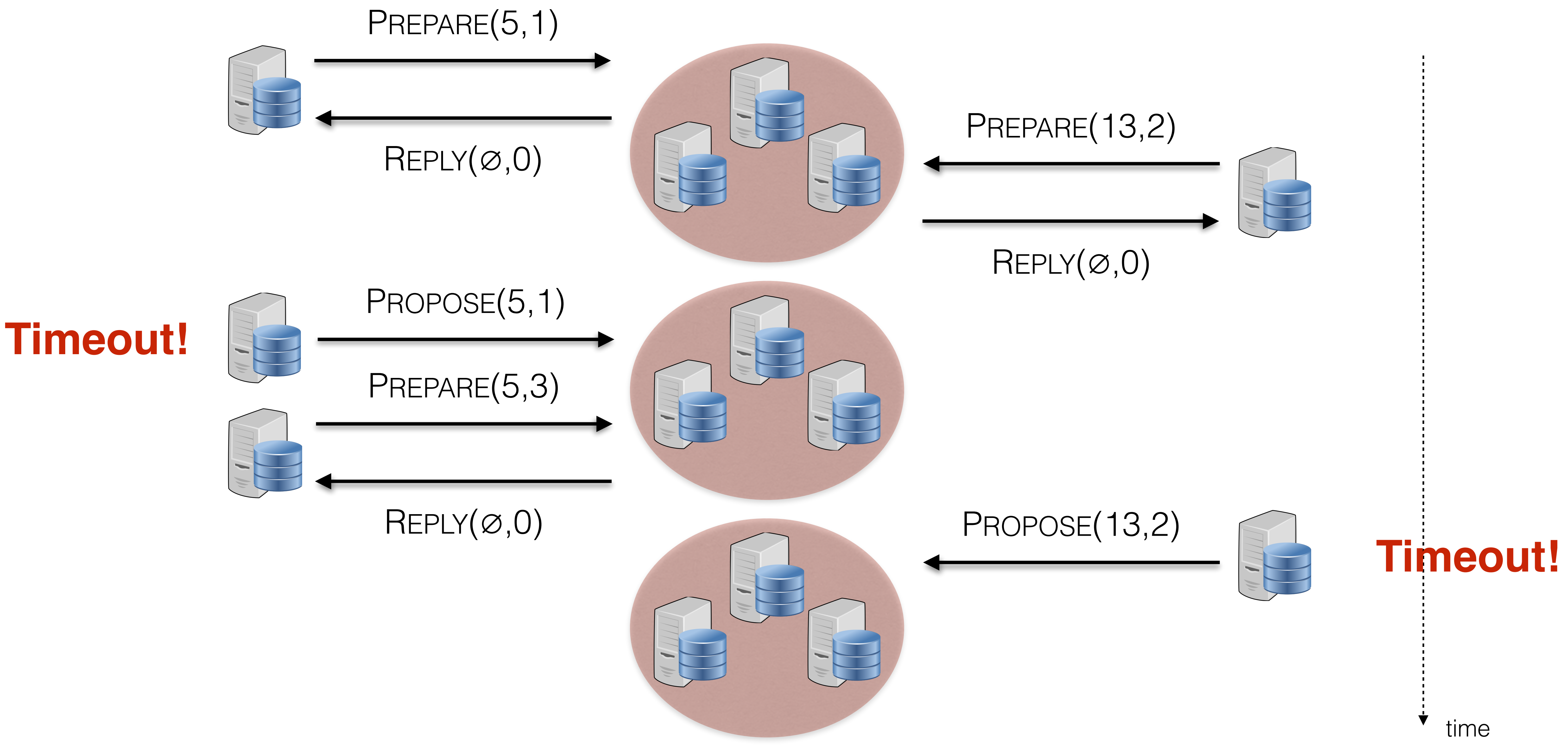
REPLY(∅,0)

PROPOSE(13,2)

time

No Liveness Guarantees

- PAXOS only guarantees that if a value is chosen, the other nodes can only choose the same value
- PAXOS does not guarantee that a value is chosen!



PREPARE(5,1)

REPLY(∅,0)

PREPARE(13,2)

REPLY(∅,0)

Timeout!

PROPOSE(5,1)

PREPARE(5,3)

REPLY(∅,0)

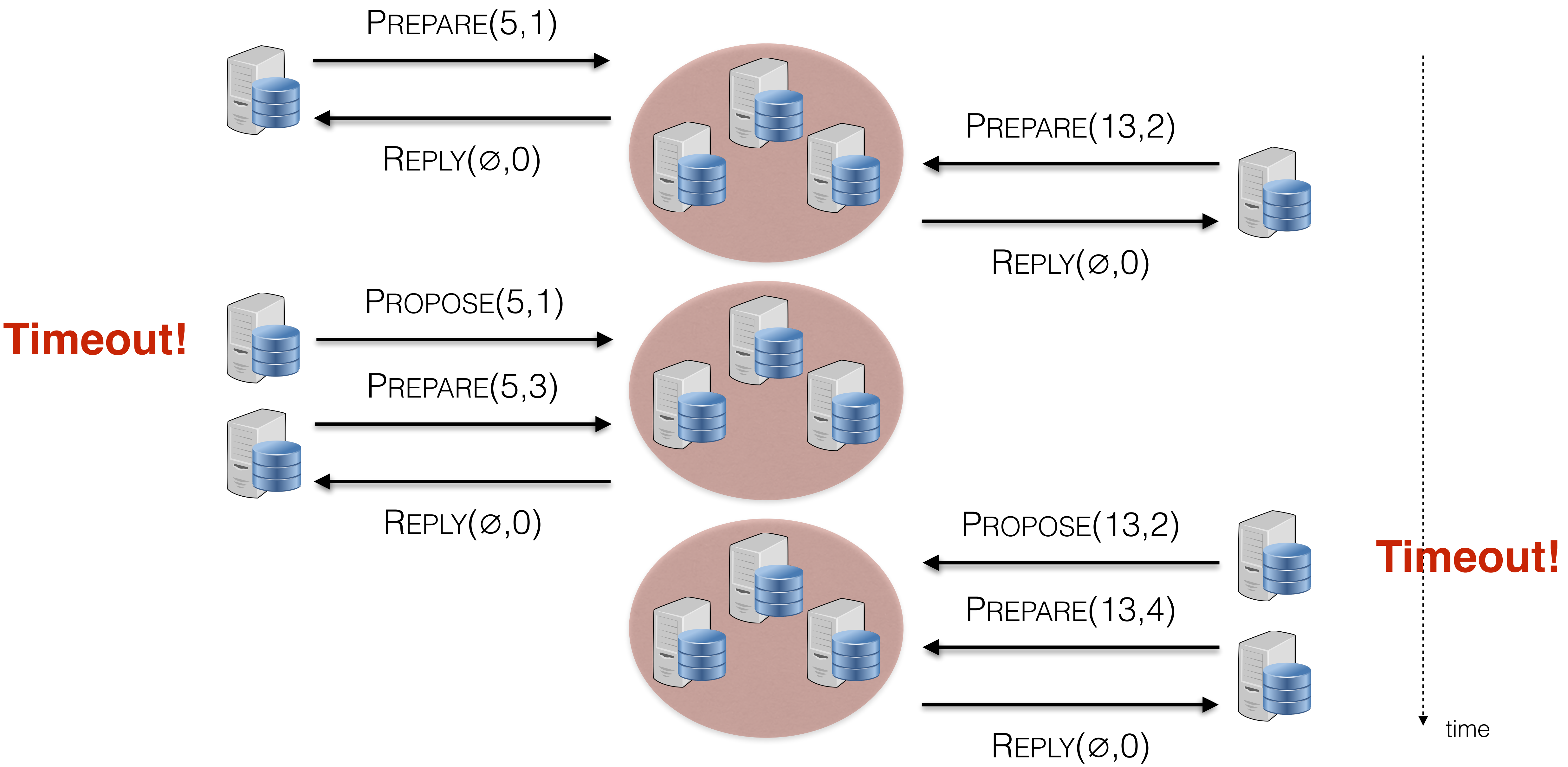
PROPOSE(13,2)

Timeout!

time

No Liveness Guarantees

- PAXOS only guarantees that if a value is chosen, the other nodes can only choose the same value
- PAXOS does not guarantee that a value is chosen!



PREPARE(5,1)

REPLY(∅,0)

PREPARE(13,2)

REPLY(∅,0)

Timeout!

PROPOSE(5,1)

PREPARE(5,3)

REPLY(∅,0)

PROPOSE(13,2)

Timeout!

PREPARE(13,4)

REPLY(∅,0)

time

Correctness vs. Termination

- In asynchronous systems, we cannot guarantee termination and correctness at the same time
- PAXOS is correct, so termination is not guaranteed
- PAXOS cannot guarantee that a consensus is reached in a finite number of steps
- In practice, PAXOS can be optimized to reduce probability of no termination
- For example, the acceptors could send NAK if they do not accept a prepare message or a proposal (this optimization increases the message complexity)
- PAXOS is used in Apache’s Zookeeper and Google’s Chubby