COMP5349 – Cloud Computing

Week 11: Consistency in Cloud Storage and DB Services

Dr. Ying Zhou
School of Computer Science

Outline

- **Consistency in Cloud Storage and DB Services**
  - **Individual customized solutions**
- **Paxos Algorithm**

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Cloud Storage and DB Services

- Run on a large fleet of nodes
- Data are partitioned and replicated
  - In GFS, a large file is split into many fix sized chunks, each chunk is replicated on three different nodes (by default)
  - In Bigtable, a large table is split into many tablets, the actual tablet files (logs and SSTables) are replicated by GFS
  - Windows Azure Storage adopts similar layered design, with storage optimization using erasure coding
  - In Dynamo, there is no table concept, the whole database is a collection of Key/Value pairs. Each key/value pair is stored on a specific node based on its key's hash value and is replicated in two other subsequent nodes (by default).
  - In Aurora, the database layer consists of MySQL or Postgres instances organized following master/slave replication, the actual data is partitioned into 10GB segments and each is replicated 6 times by underlying storage service

How to keep replicas consistent

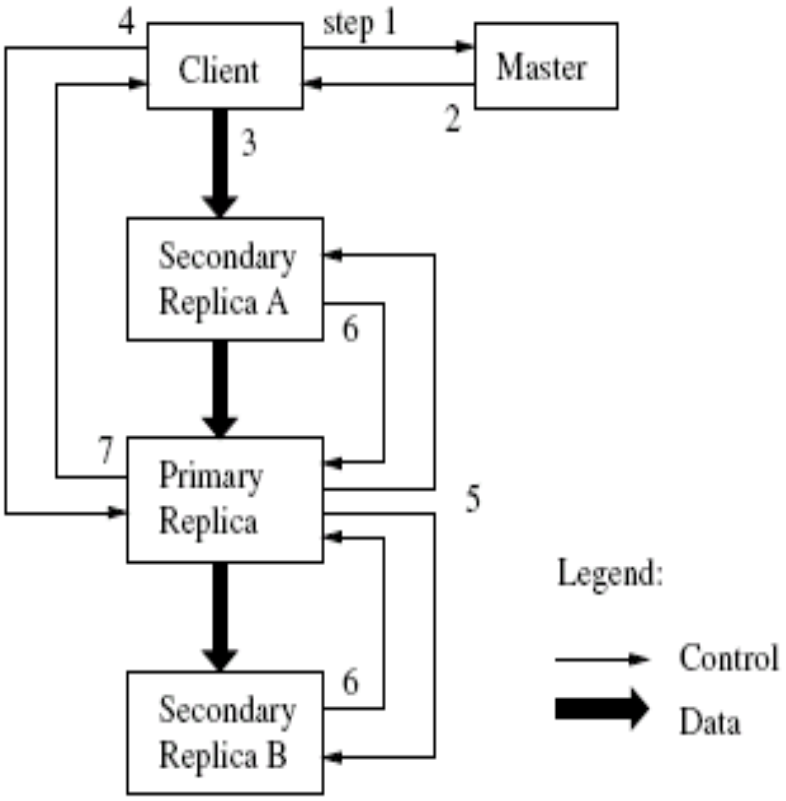
- There are two ways of looking at consistency.
  - From client point of view
  - From server point of view
- From client point of view
  - There are many processes performing read/write operations on a distributed storage/db system
  - *Strong consistency*: after a process completes the write operation, subsequent read should return the updated value
  - *Weak consistency*: after a process completes the write operation, some read may return old value
  - *Eventual consistency*: if no new write are made to the object, eventually all reads will return the latest updated value

https://www.allthingsdistributed.com/2008/12/eventually_consistent.html

How to keep replicas consistent

- Consistency level observed by the client depends on how system (server side) handles write and read
  - To ensure strong consistency, do we need to keep all replicas the same after a write request is complete?

Revisit: Write and Read in GFS



1. The client asks the master which chunkserver holds the current lease for the chunk and the locations of the other replicas.

   Grant a new lease if no one holds one

2. The master replies with the identity of the primary and the locations of the other (*secondary*) replicas

   Cached in the client

3. The client pushes the data to all the replicas

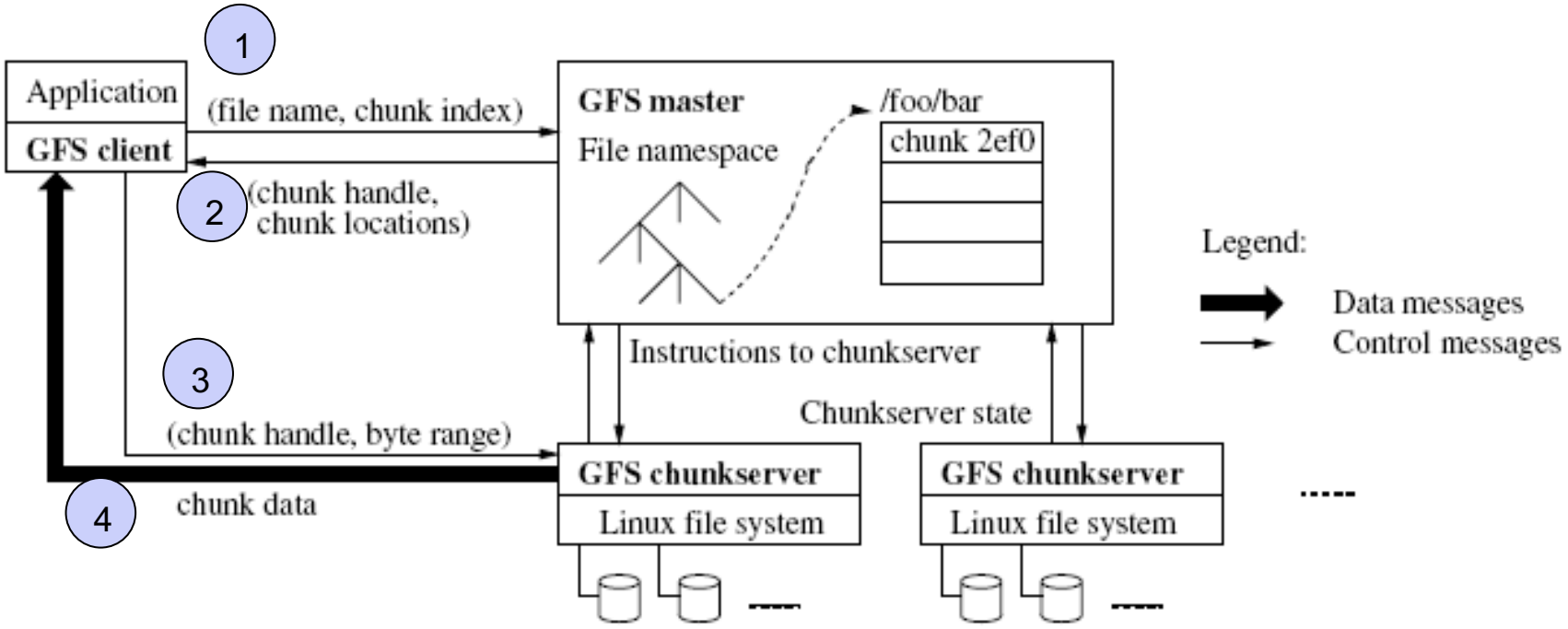
4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization.

5. The Primary forwards the write request to all secondary replicas.

6. Secondaries signal completion.

7. Primary replies to client. Errors handled by retrying.

Revisit: GFS write and read



- Client translates file name and byte offset to chunk index.
- Sends request to master.
- Master replies with chunk handle and location of replicas.
- Client caches this info.
- Client sends request to a close replica, specifying chunk handle and byte range.
- The chunk server sends chunk data to the client

GFS: Read and Write

- Read request only contacts a single replica
- Write request contacts all replicas and the acknowledgement is sent after all replicas have received the data and acted on it
  - Eager propagation
- GFS only achieves weak consistency
  - It does not have roll back mechanism, some replica may fail to write the data in the disk after receiving the data, leaving data in inconsistency state among replica
  - Simple retry may create duplicate records in certain replica

Revisit: Write and Read in Bigtable

- Each tablet is served by a single tablet server
  - Both read and write are managed by this server
- Write path
  - The latest write content is kept in the memory of the tablet server
  - The write request is also persisted as commit log (GFS files)
- Bigtable has strong consistency

Revisit: Dynamo/Aurora Read and Write

- Dynamo/Aurora uses quorum based technique
  - Obtain a minimum number of votes before carrying out an operation
  - In the context of replicated storage/database system, a minimum number of replicas should be responded before returning the write and read results back to the client
  - Replica number(**N**), Write(**W**) and read(**R**) quorum, the rules for strong consistency include
    - W > N/2
    - W+R > N
    - Typical value combinations:
      - N = 3, W = 2, R = 2
      - N = 6, W = 4, R = 3 (Amazon Aurora)
  - In a typical setting, quorum members are replica, Dynamo uses sloppy quorum which allows node without no data copy to participate in the read/write operation
    - Dynamo achieves eventual consistency

In summary

- Read/Write operations in replicated database systems can be implemented in various ways to achieve different levels of consistency
- How many replicas to contact for read and write operations?
- How many responses to wait before responding to the client?
- E.g.
  - Contact N replica, wait for N responses
    - GFS write
  - Contact 1 replica, wait for 1 response
    - GFS read
  - Contact N replica, wait for W or R responses (W< N, R<N)
    - Amazon Dynamo and Aurora

In Summary (cont'd)

- Who is coordinating the communication among replicas?
  - Always the same node for the any read/write request
    - Example?
  - One node per data partition
    - Example?
  - Any replica
    - Example
- Typical communication pattern during write
  - Coordinator to all others
    - Are you OK to perform this write?
  - If coordinator receives enough responses, send another message to all others
    - Do it now

Outline

- Consistency in Cloud Storage and DB Services
- **Paxos Algorithm**
  - **Basic Algorithm**
  - Running Multiple Paxos in Replicated System

Paxos Protocol

- Proposed by Leslie Lamport in 1998
  and in 2001 refined (or: explained in plain English ;)
- Central Question:
  " Assume a collection of processes that can propose values. A consensus algorithm ensures that a single one among the proposed values is chosen."

Paxos – Preliminaries

- Distributed System with multiple nodes
- Shared, global state
  - In database sense: data replication
- Processes are concurrent, asynchronous, and error-prone
  - Each node can 'propose' a new value for a data item
    - In database sense: update-everywhere
  - Nodes communicate via asynchronous messages
    - Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost, but they are not corrupted.
  - Nodes can fail
    - But after failure, will eventually restart
    - Can remember *some* information if restarted after failure

Requirements for Correct Consensus

Safety Requirements:

- Only a *value that has been proposed* may be chosen
- Only a *single value* is chosen, and
- A process never *learns* that a value has been chosen unless *it actually has been*.
  - Note: no restriction on *what* value is actually chosen as long as it satisfies these three safety requirements

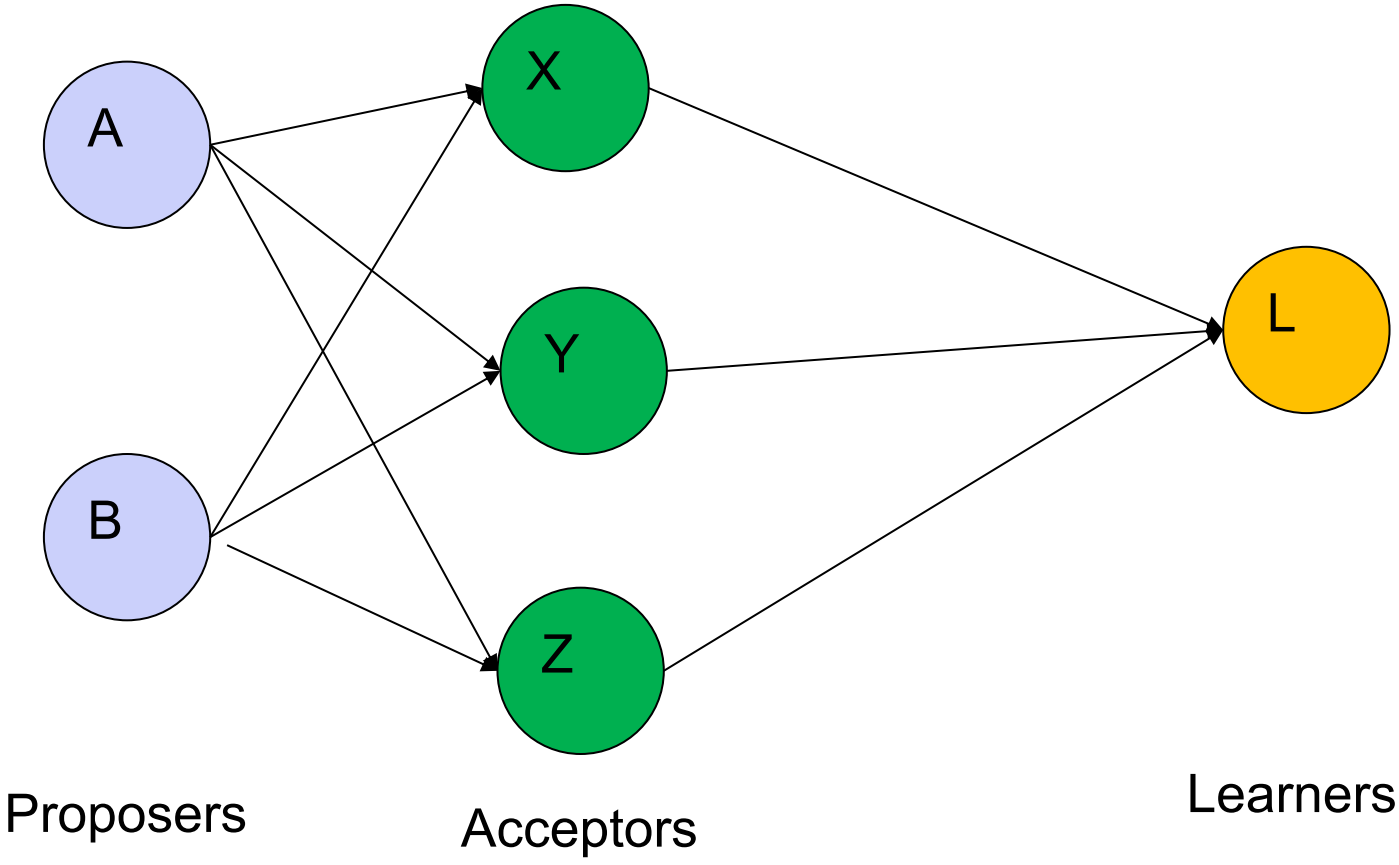
Liveness Requirements:

- Some proposed value is eventually chosen.
- If a value has been chosen, a node can eventually learn the value.

Paxos: Terminology

- Paxos is a distributed consensus algorithm
- Three roles for the participants ('agents') of the algorithm:
  - **Proposers**
  - **Acceptors**
  - **Learners**
  - A specific node might play any number (one, two or three) of these roles simultaneously for different data items; the role names are just a tool for better understanding the algorithm.
- A **proposer** sends a proposed value to a set of **acceptor**s. An acceptor may accept the proposed value. The value is *chosen* when a *large enough set of acceptors* have accepted it.

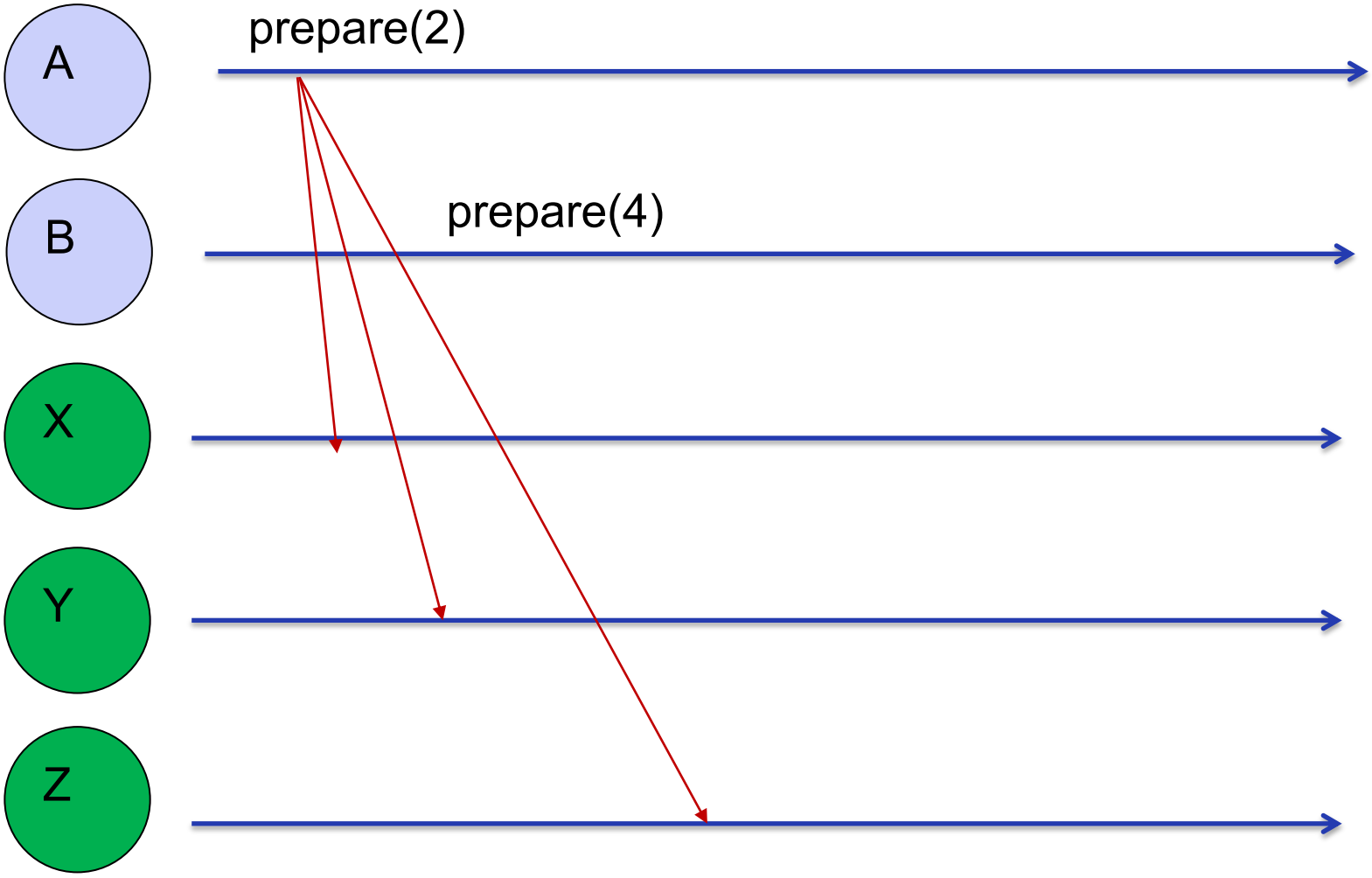
Paxos Example: agents



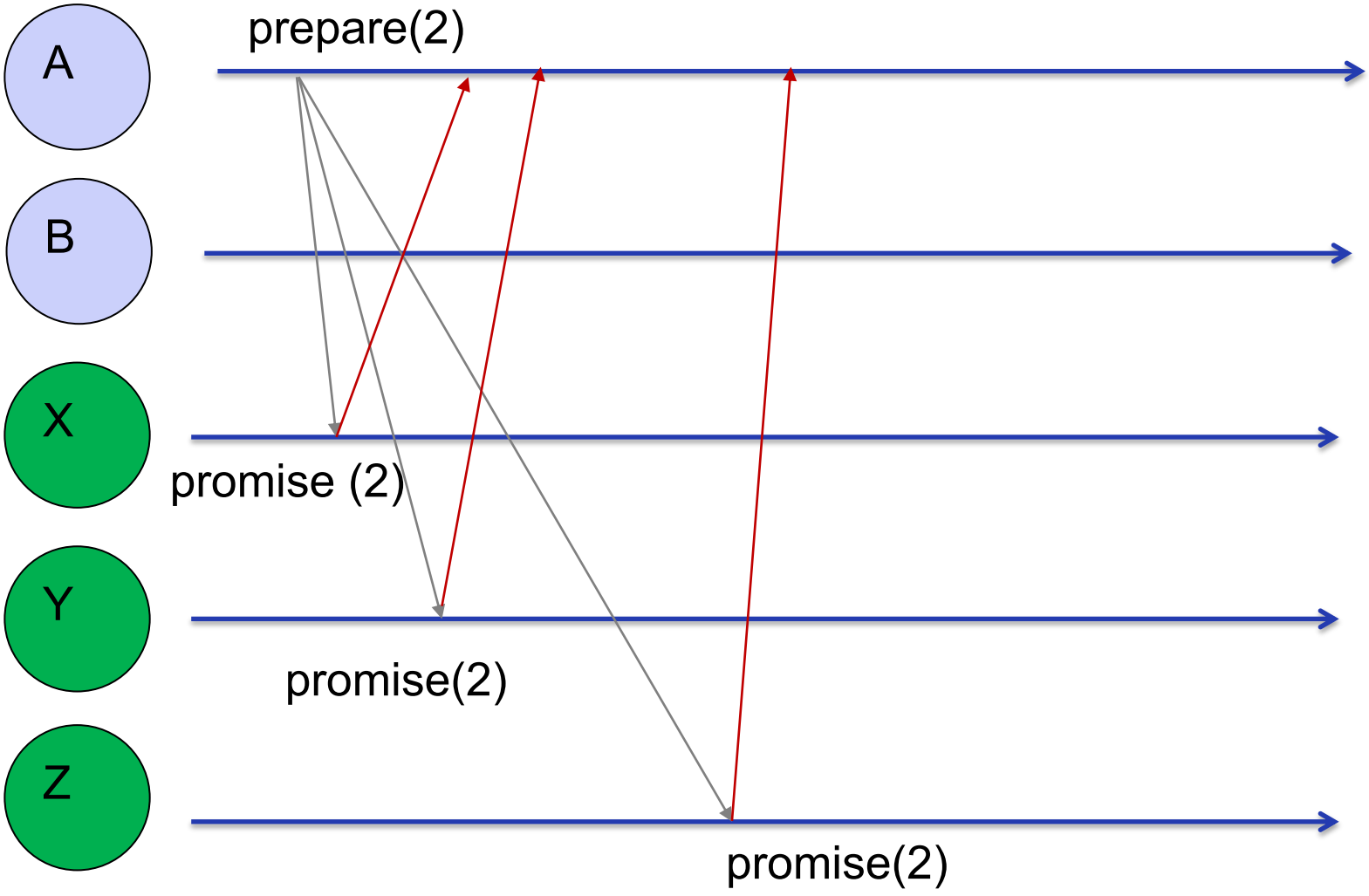
Paxos Phase 1: Prepare & Promise

- A **proposer** creates a proposal identified with a number *n* and sends a *prepare request* with this number to some quorum of *A* acceptors
  - e.g. via a broadcast to some majority of the acceptors
  - *n* must be greater than any previous proposal number seen by proposer
- If an **acceptor** receives a *prepare request* with number *n* greater than that of any *prepare request* it saw,
  - **it responds the request with a promise not to accept any more proposals numbered less than *n*,** and
  - includes the highest-numbered proposal less than *n* (if any) that it has accepted before (typically from a concurrent proposal); *nil* otherwise
- Acceptors can ignore any prepare request with a number *n* smaller or equal to another request that it already saw

Phase 1: prepare request



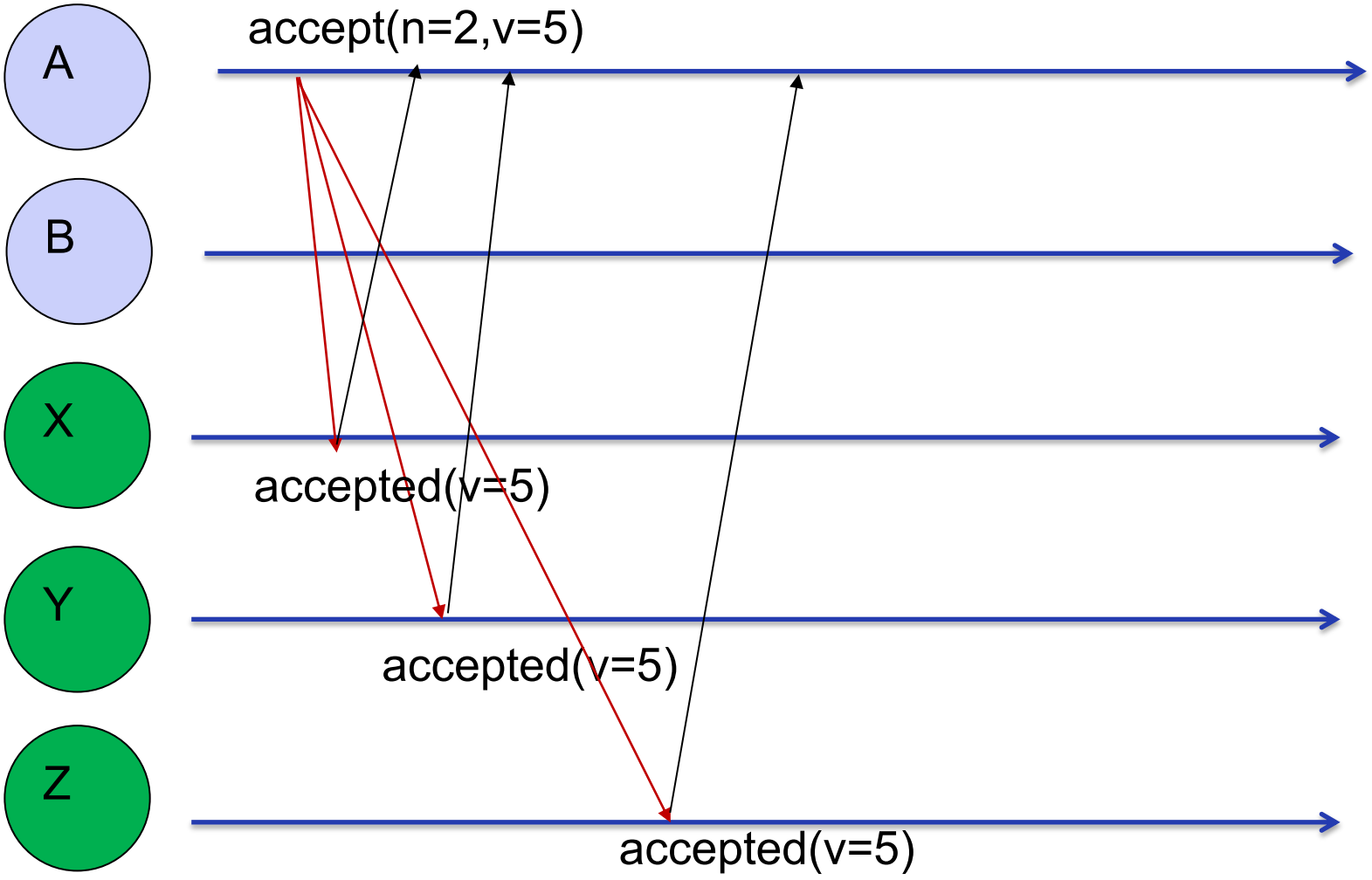
Phase 1: Acceptor Response



Paxos Phase 2: Accept

- If the **proposer** receives a response to its prepare requests (numbered n) from a *majority of acceptors*, then it sends an *accept request* to each of those acceptors for a proposal numbered *n* with a value *v*, *where v is the value of the highest-numbered proposal among the responses*, or is any value if the responses reported no proposals.
- (b) If an **acceptor** receives an accept request for a proposal numbered *n*, it accepts the proposal unless it has already responded to a prepare request having a number greater than *n*.
  - Only higher-numbered proposals overwrite lower-numbered ones

Paxos Phase 2: Accept Request

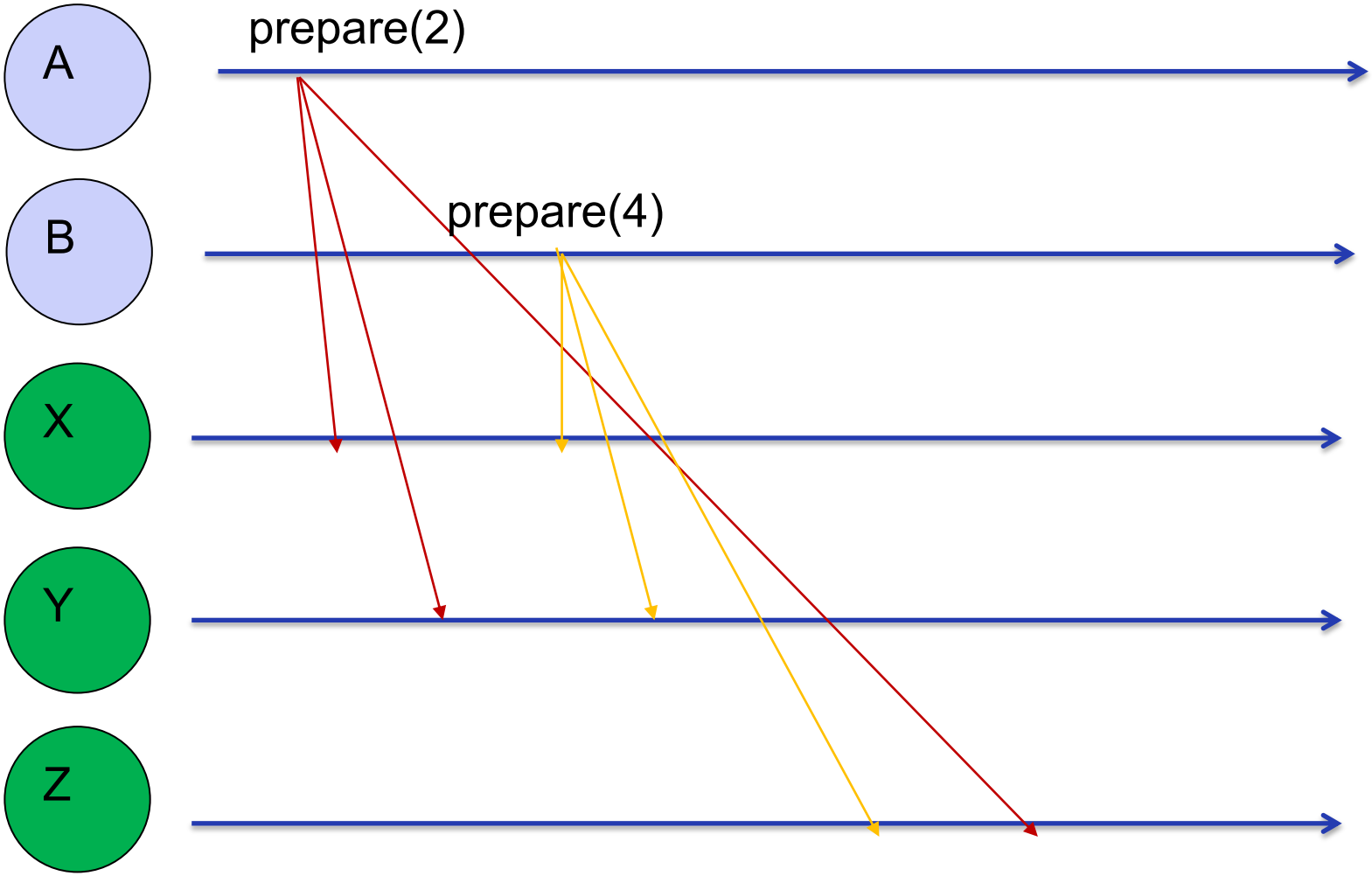


The chosen value is 5

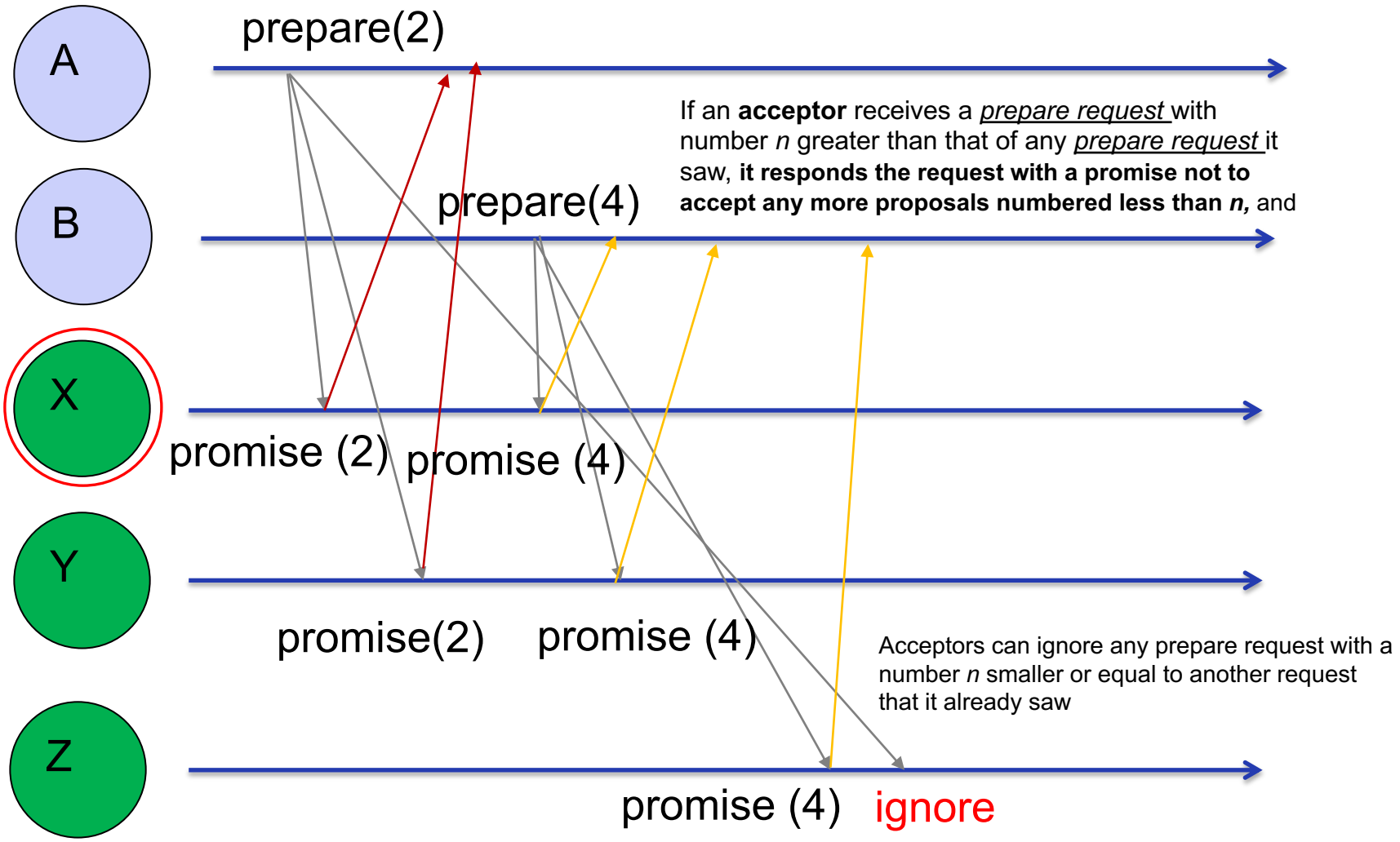
A concurrent requests case

- Paxos is designed with concurrent requests in mind and it is also designed to tolerate message loss (request, response)
  - Request may arrive in different orders
  - Request or response may loss
  - Request may be ignored

Phase 1: prepare request



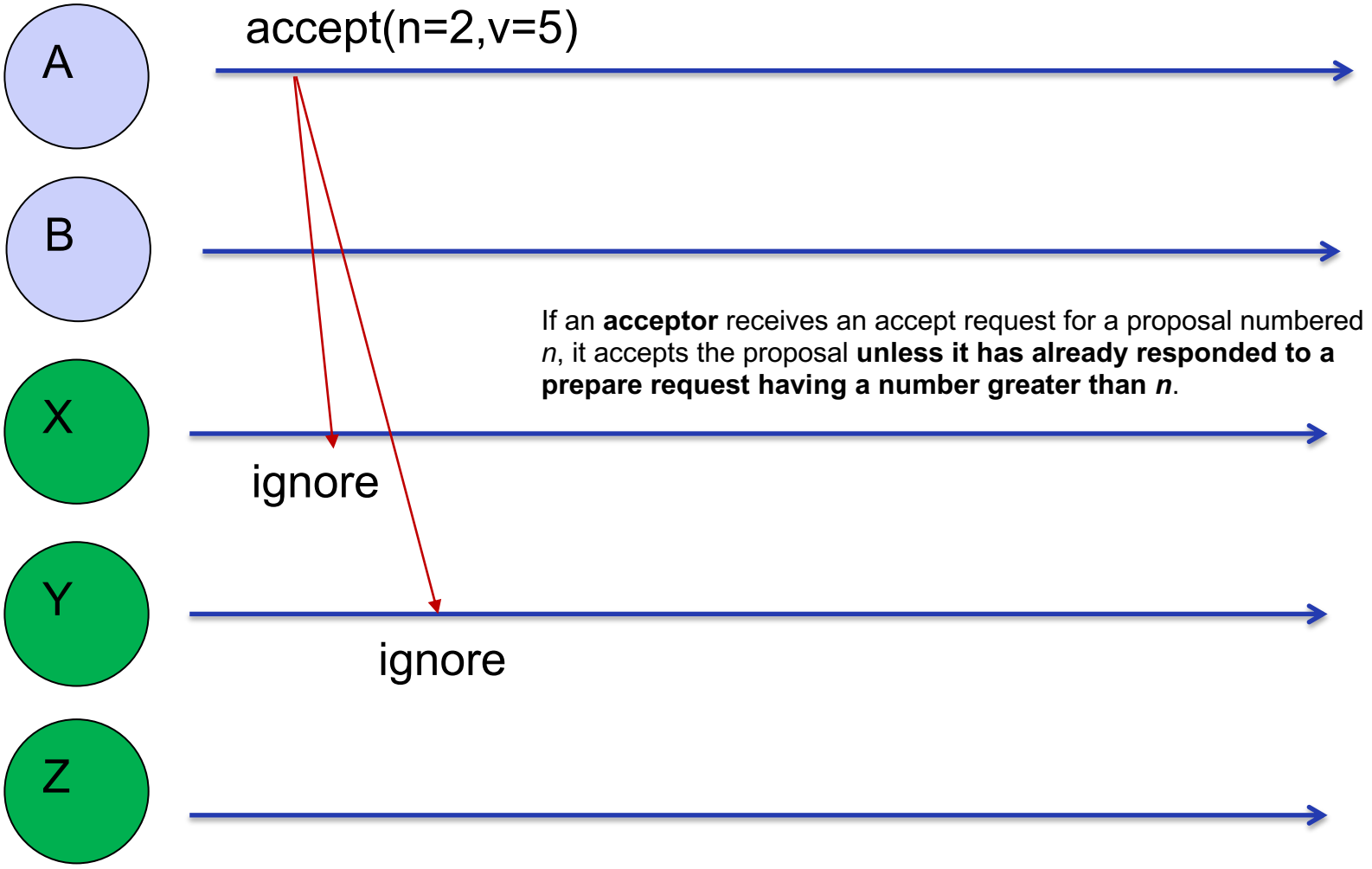
Phase 1: Acceptor Response



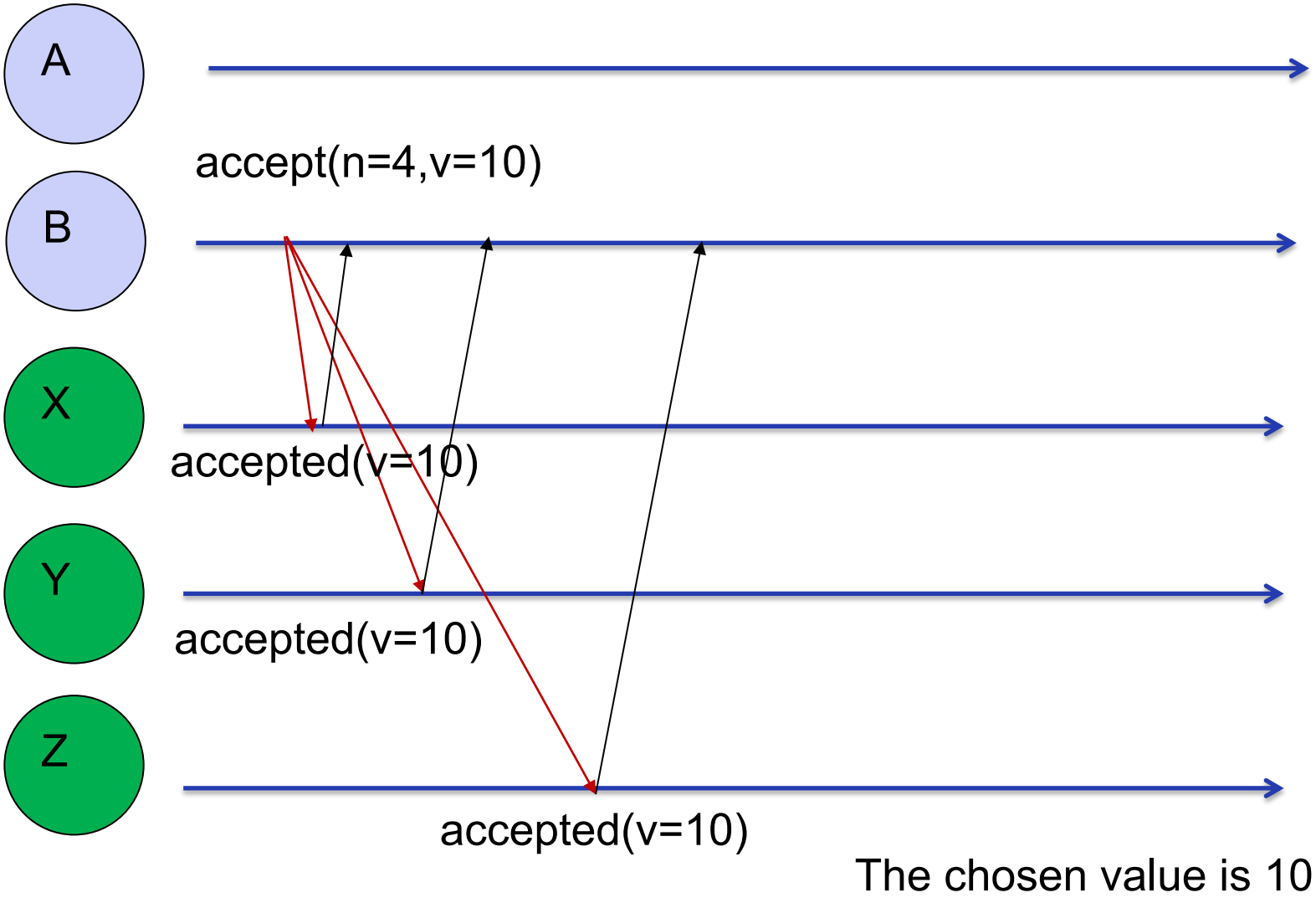
The situation after phase 1

- Both proposers receive responses from a majority of the acceptors
  - Proposer A receivers two responses from X and Y
  - Proposer B receives three responses from X, Y and Z
- Some acceptors made more than one promises
  - Both X and Y made two promises to two different proposers
  - But they are not conflicting, one subsumes the other
  - Promise *not to accept any proposal with number less than 4* **overwrites** promise *not to accept any propose with number less than 2.*

Example: Phase 2



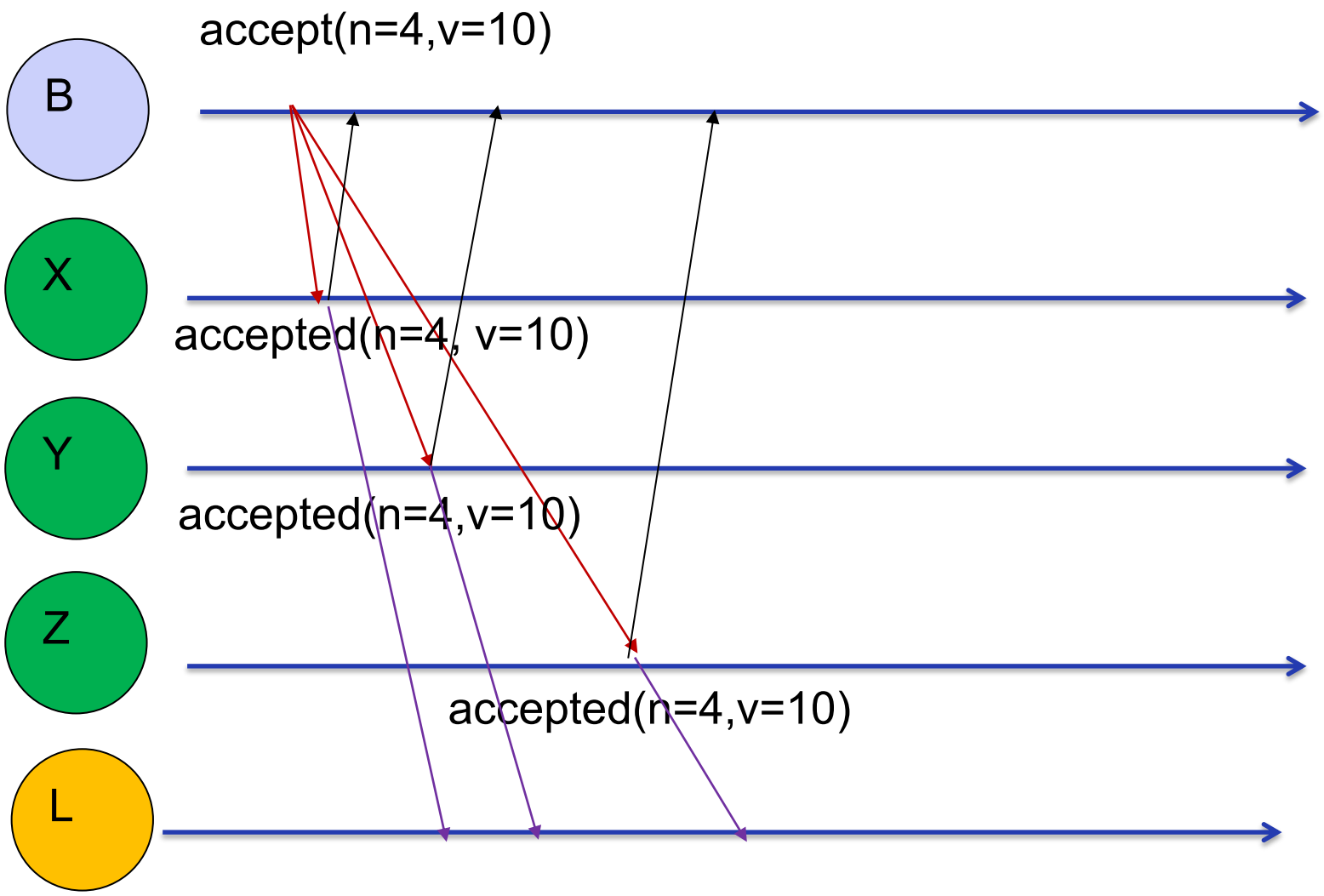
Example: Phase 2



Paxos: Definition of Chosen

- A value is chosen at proposal number *n* **iff** the majority of acceptor nodes accept that value in phase 2 of the algorithm
- Learning a chosen value
  - Easy solution: each acceptor, whenever it accepts a proposal, respond to all learners by sending the proposal.

Learning a chosen value



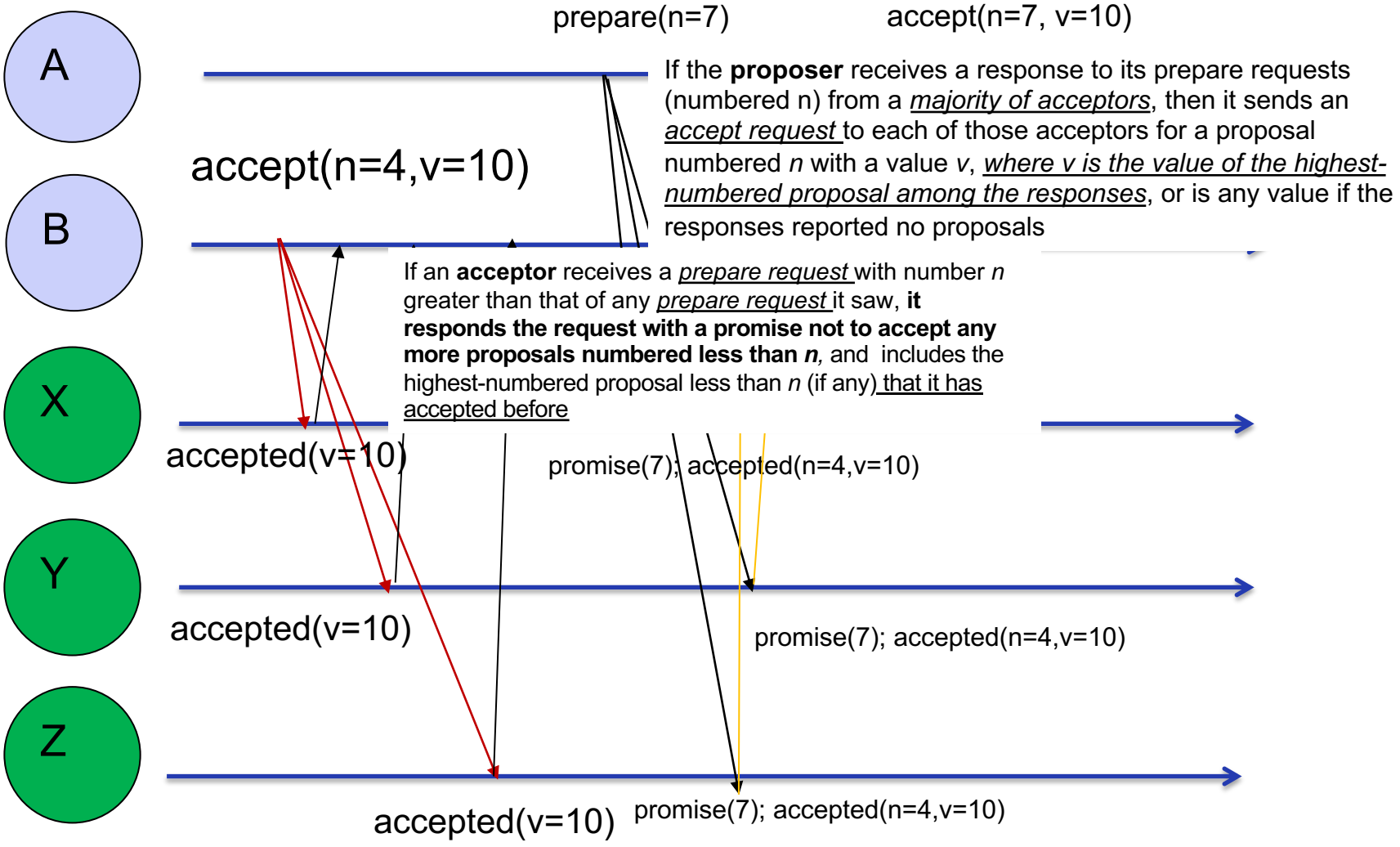
Learning a chosen value: optimization

- The simple way would need *n* m* messages where *n* is the number of acceptors and *m* is the number of learners
- To avoid unnecessary message passing, it is possible to have a single *distinguished learner*. All acceptors will send accept message to this learner, which then informs other learner of the value chosen
- Message loss scenario:
  - All messages from acceptors to learner may have lost
  - A learner can be more proactive by polling all acceptors what value has been chosen, but again, some message may get lost and no majority is formed
  - A learner need to ask the proposer to create another round of proposing to learn the value chosen

Implications

- A proposer can make multiple proposals, only a single value will be chosen in a *single instance* of Paxos algorithm
- Sometimes, it would be a good idea for a proposer to abandon a proposal and this won't affect the correctness
  - e.g in the above example, if proposer A found out that B has started another proposal with high number, it does not have to continue with the second phase

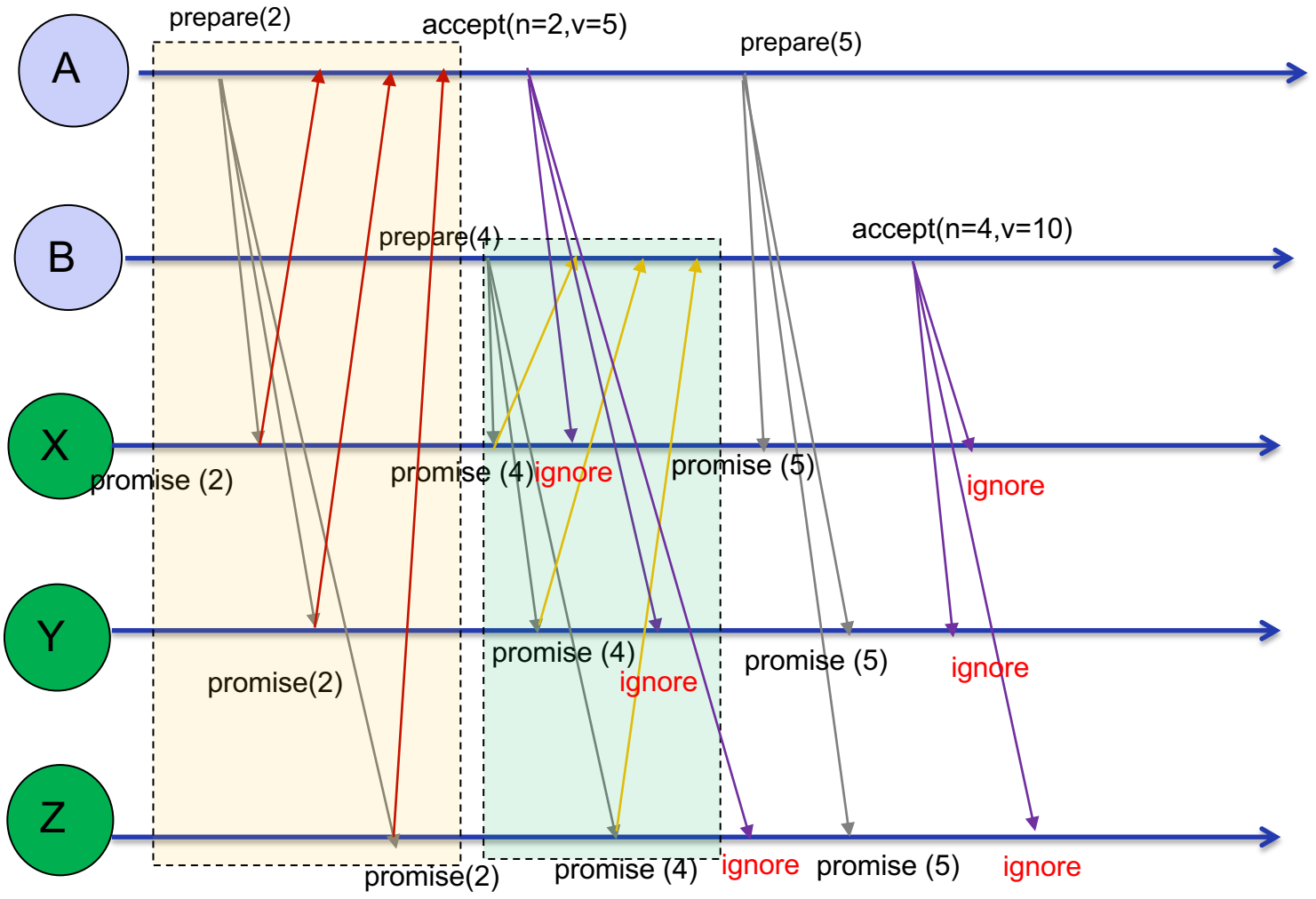
Multiple Proposals Example



Progress

- In a system with two proposers, it is possible that each keeps issuing a sequence of proposals with increasing numbers, each in between the other proposer's phase 1 and phase 2
  - Proposer ***p*** completes phase 1 for a proposal number n1.
  - Another proposer ***q*** then completes phase 1 for a proposal number n2 > n1.
  - Proposer ***p***'s phase 2 accept requests for a proposal numbered n1 are ignored because the acceptors have all promised not to accept any new proposal numbered less than n2.
  - proposer ***p*** then begins and completes phase 1 for a new proposal number n3 > n2, causing the second phase 2 accept requests of proposer q to be ignored.
- To guarantee progress, a ***distinguished proposer*** must be selected as the only one to try issuing proposals.
  - Like the coordinator in a replicated system

Non-progress situation



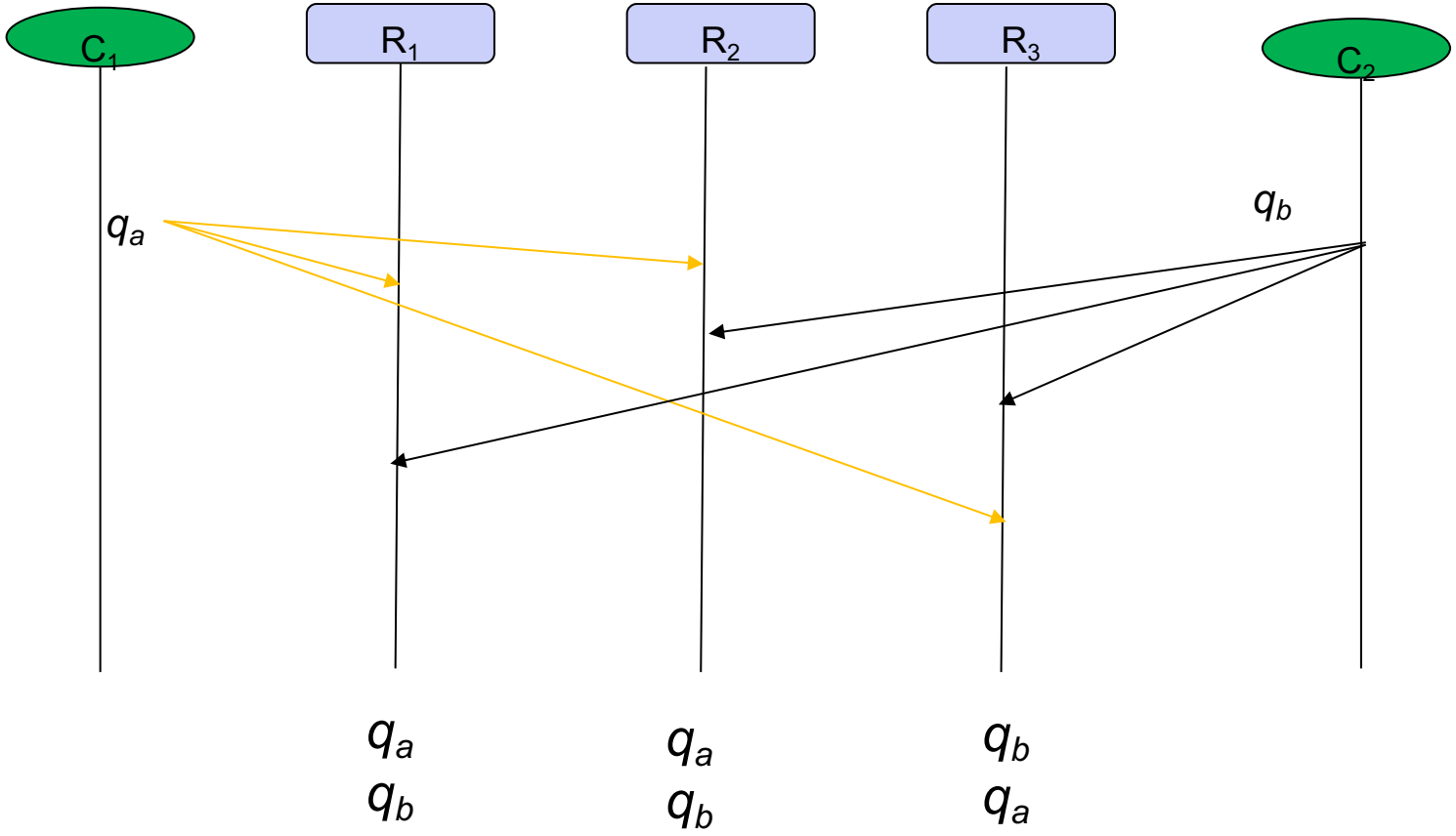
Outline

- Consistency in Cloud Storage and DB Services
- **Paxos Algorithm**
  - Basic Algorithm
  - **Running Multiple Paxos in Replicated System**

Implementing a State Machine

- A single instance of Paxos algorithm only allows one value to be chosen
- Any real system would involve many data, replicated in a number of servers and the data can be updated by multiple clients
- Each replica server can be viewed as a deterministic state machine that performs client commands in some sequence
- The state machine has a current state; it performs a step by taking as input a command and producing an output and a new state.
- For strong consistency, the system need to ensure
  - All or majority of the replica servers execute the client commands
  - They execute the commands in the same order

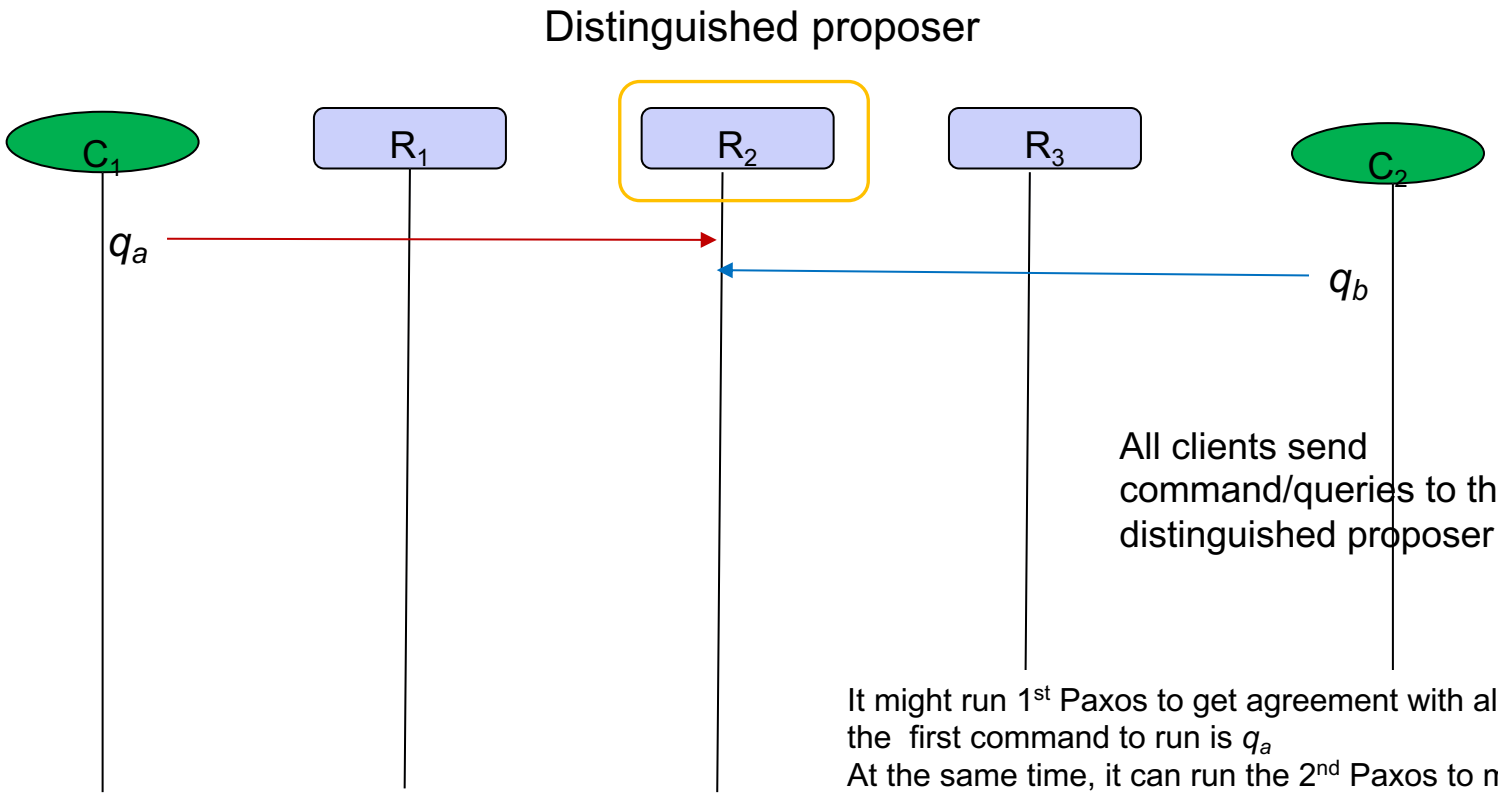
Replicated System



Multiple Paxos

- This would be implemented as infinite number of Paxos instance, correspond to the number of client commands a distributed system can receive
- Assuming the set of servers is fixed, each server should play all the roles (proposer, acceptor and learner)
  - All instances would have the same set of agents.
- A single server is elected as the distinguished proposer
- Each Paxos instance is numbered and represent the i^{th} command the system should run.
  - It runs a Paxos algorithm to chose what would be the actual command to run in the i^{th} step, the value of a proposal would be an actual client request, e.g. update row 1 of table 2 by setting column 3's value to 4 .

Multiple Paxos in replicated system



All clients send command/queries to the distinguished proposer

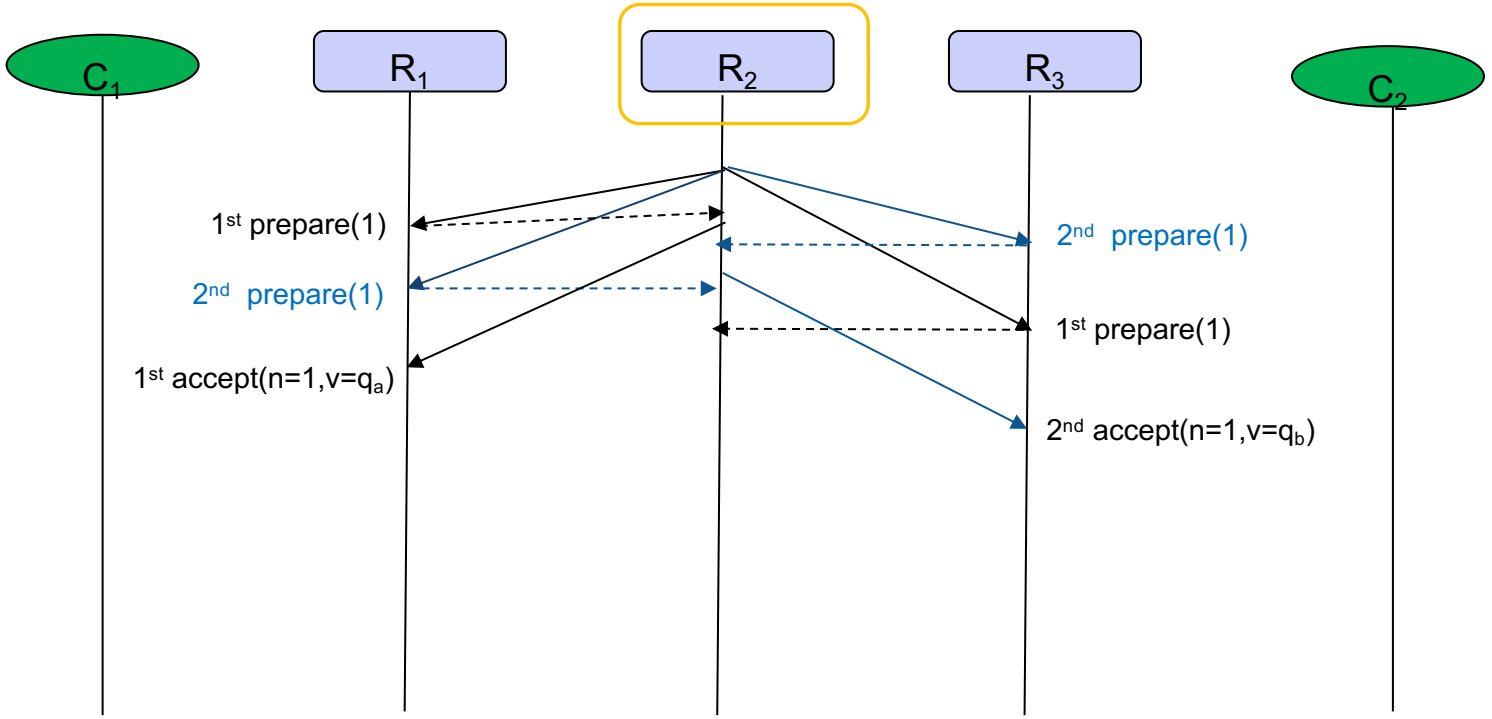
The distinguished proposer runs infinite number of Paxos instance, each is numbered and represent the i^{th} command/query the system should run

It might run 1st Paxos to get agreement with all replicas that the first command to run is q_a
At the same time, it can run the 2nd Paxos to make sure that all replicas know that the second command to run is q_b

Any replica that knows the chosen value of 1st and 2nd Paxos can go ahead to execute the queries

A replica may learn the value of 2nd Paxos before learning the 1st Paxos, in that case, it needs to wait until it learns the value of the 1st Paxos instance.

Multiple Paxos at the same time



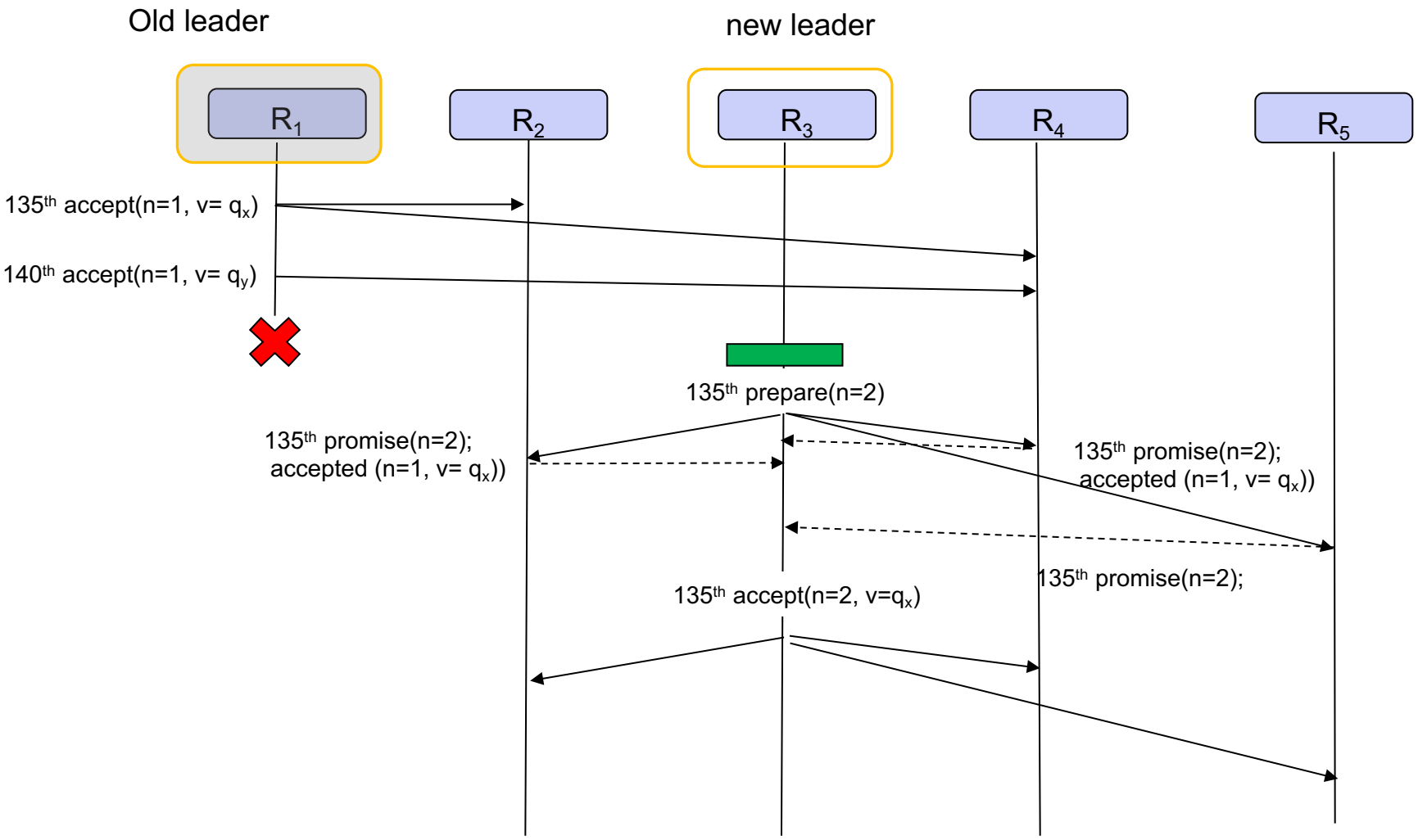
Leadership Changeover

- Suppose the previous leader has just failed and a new leader has been elected.
  - The new leader is one of the replicas, it should know most of the commands that have been chosen, but may have slightly different knowledge with other replicas
  - Suppose “it knows commands 1–134, 138, and 139—that is, the values chosen in instances 1–134, 138, and 139 of the consensus algorithm”
  - There is a knowledge gap 135-137 for the new leader, it could be caused by
    - Previous leader has not executed phase 2 of these instances
    - Previous leader has executed phase 2 and only some replica received the message
  - The new leader also does not know if any value has been chosen or been proposed in any Paxos instance after 139 by the old leader

New leader catching up

- Executes phase 1 of instances 135–137 and of all instances greater than 139
  - If some other replica receives the phase 2 request from the old leader in any instance, it will respond with those accepted value to the new leader
  - Otherwise, just a promise
- Suppose in instances 135 and 140, some replica has accepted values proposed by the old leader
  - They would respond their accepted values
- The new leader would execute the 2nd phase of these two instances to confirm the chosen value

New leader catching up (illustration)



The 140th instance is handled in slightly different way

New leader catching up (cont'd)

- Now the system has chosen value for 1-135; 138-140
  - All replicas call execute commands 1-135;
  - They cannot execute 138-140 commands because there is a gap, there is no value for 136 and 137
- The gap could be caused by
  - The old leader only executed the first phase of these two instances, or
  - None of the replica received second phase message
- The new leader has two options to fill in the gap
  - Assign the next two commands issued by the client as 136 and 137 value
    - But they are likely issued after all 138-140 commands
  - **Assign special no-op value to both.**
- The new leader has already executed the first phase those instances
  - Execute the second phase with special value no-op
  - Now all replicas can go ahead with 130-140 commands
- The new leader go ahead by assigning newly received commands to 141 and later instance

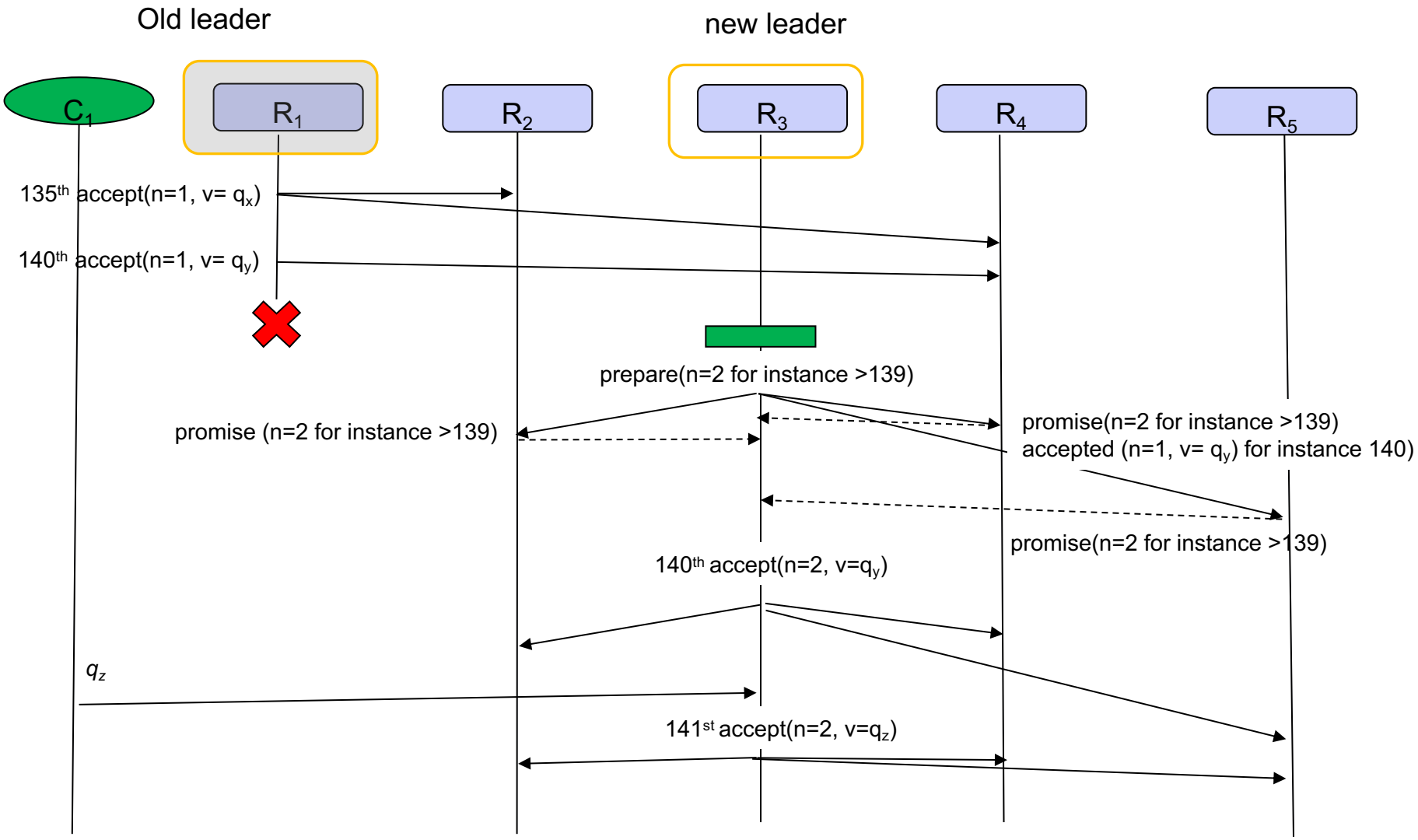
The need for running infinite instances

- A newly chosen leader executes phase 1 for infinitely many instances of the consensus algorithm.
  - E.g. In previous scenario, a new leader needs to "Executes phase 1 of instances 135–137 and of ***all instances greater than 139***"
- Any leader can send proposals for various commands concurrently
- It does not need to wait for a value chosen for 140th instance before sending out the request for 141st instance
- The new leader does not know how far ahead the old leader has been progressed
  - E.g. In the scenario, the new leader only has data up to **139th** instance, but some replica has accepted **140th** instance's value

Run infinite instances efficiently

- “Using the **same proposal number for all instances**, it can do this by sending a single reasonably short message to the other servers. In phase 1, an acceptor **responds with more than a simple OK** only if it has already received a phase 2 message from some proposer. (In the scenario, this was the case only for instances 140.) Thus, a server (acting as acceptor) can respond for all instances with a single reasonably short message. ”

Example of handling infinite instances



Any Practical Usage of Paxos?

Yes, for example:

- Google Chubby (OSDI2006)
  - Distributed lock system and meta-data repository
- Hadoop Zookeeper (DISC2009)
  - Open-source implementation of Chubby; uses own ZAP protocol that is inspired by/based on Paxos
- Google Spanner (OSDI2012)

References

- Leslie Lamport, “Paxos Made Simple”, ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)