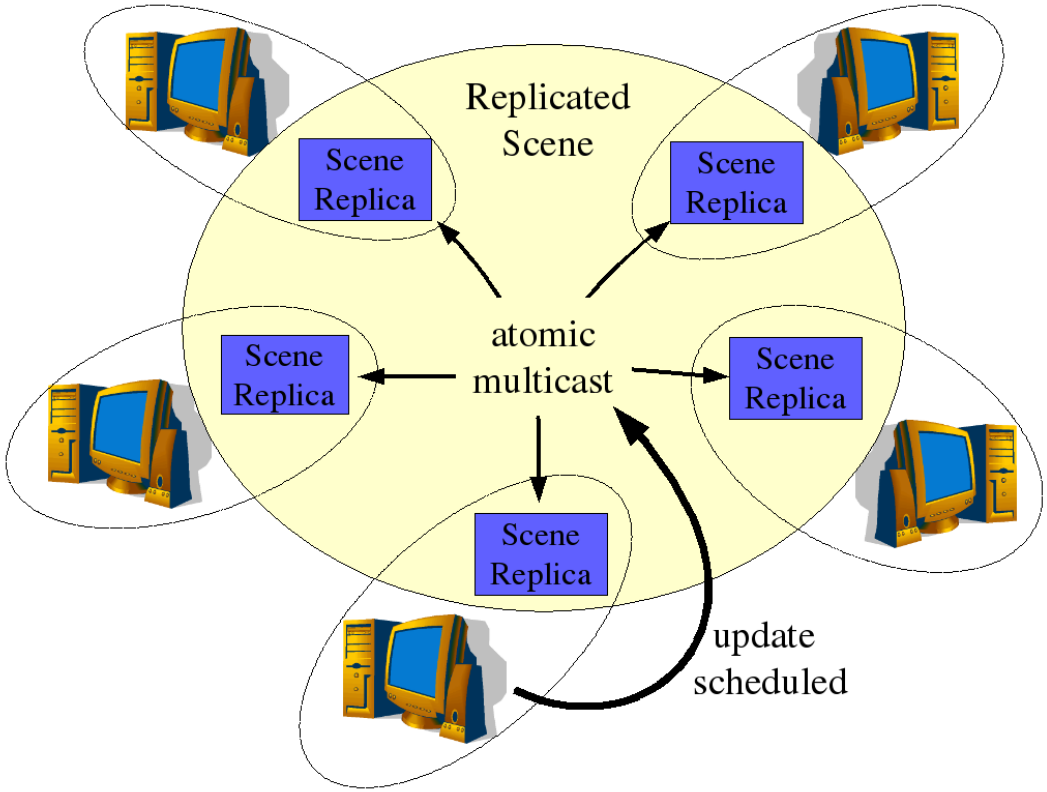
The Paxos Consensus Algorithm



https://en.wikipedia.org/wiki/File:Active_Replica

Quick review

Algorithm name	Type	DS and Failure Models
Centralized strategy	Mutual Exclusion	Asynchronous, **only inactive nodes can crash**, reliable messages
Token-based ring algorithm	Mutual Exclusion	Asynchronous, **no nodes failures**, reliable messages
Ricart & Agrawala	Mutual Exclusion	Asynchronous, **no nodes failures**, reliable messages
Changs & Roberts (Ring-based)	Leader Election	Asynchronous, **no failures during election**, reliable messages
Bully algorithm	Leader Election	Synchronous, process failures, reliable messages
Dolev-Strong algorithm	Consensus (a.k.a. agreement)	Synchronous, *f* failures with f+1 rounds, reliable messages

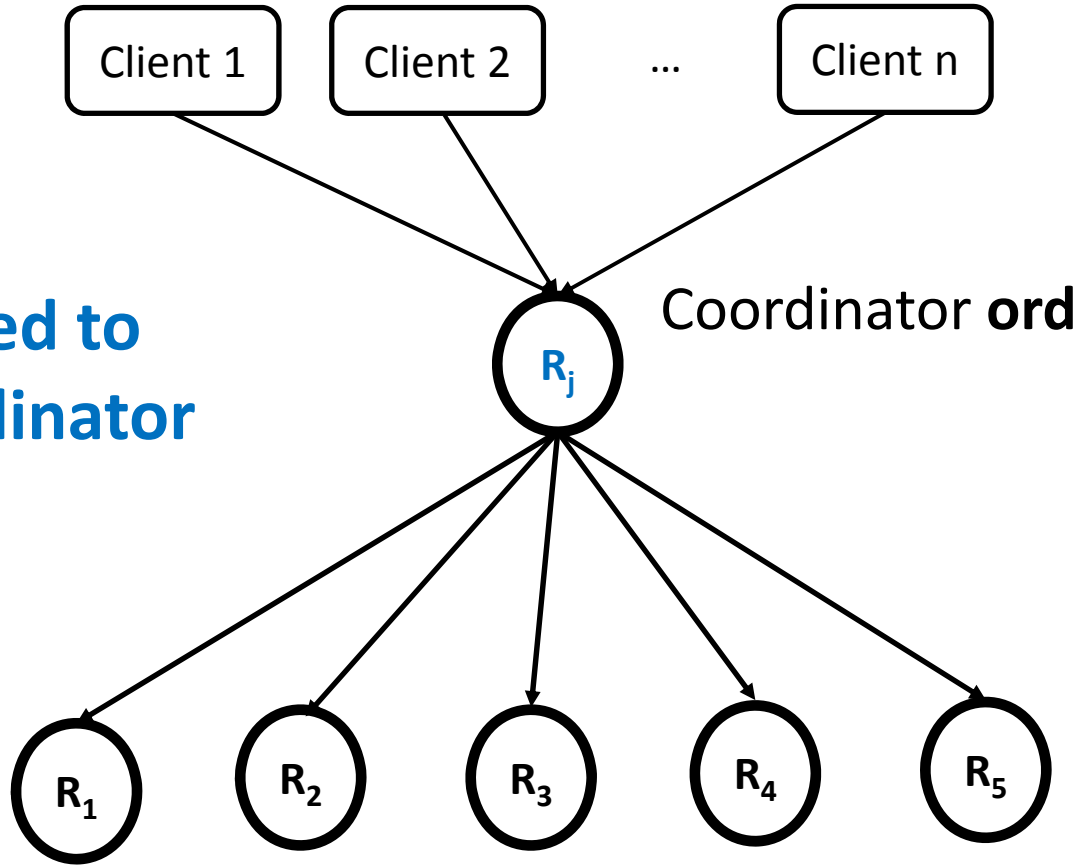
Consensus problems

- Desire **all nodes to agree** on a value after one or more node **proposed a value**
- Also known as **problems of agreement**
- **Challenges**
  - Reach consensus, even in the presence of **node failures** and **unreliable networking**

Consensus problem examples

- Two armies **agree** consistently to attack or to retreat (thought experiment)
- **Mutual exclusion**: Processes **agree** on who can enter CS
- **Leader election**: Processes **agree** on who is elected
- Replicated state machines **agree** on commands to execute
- Multi-primary replication **agree** on order of updates at replicas
- Totally ordered multicast: Processes **agree** on the order of messages delivered
- ATM and bank's servers **agree** what should happen to bank account balance when withdrawing money
- Transaction managers **agree** to commit or abort (e.g., two-phase commit)
- Flight computers **agree** to "abort" (reboot) or "proceed"
- Reactor safety systems **agree** on position of control rods

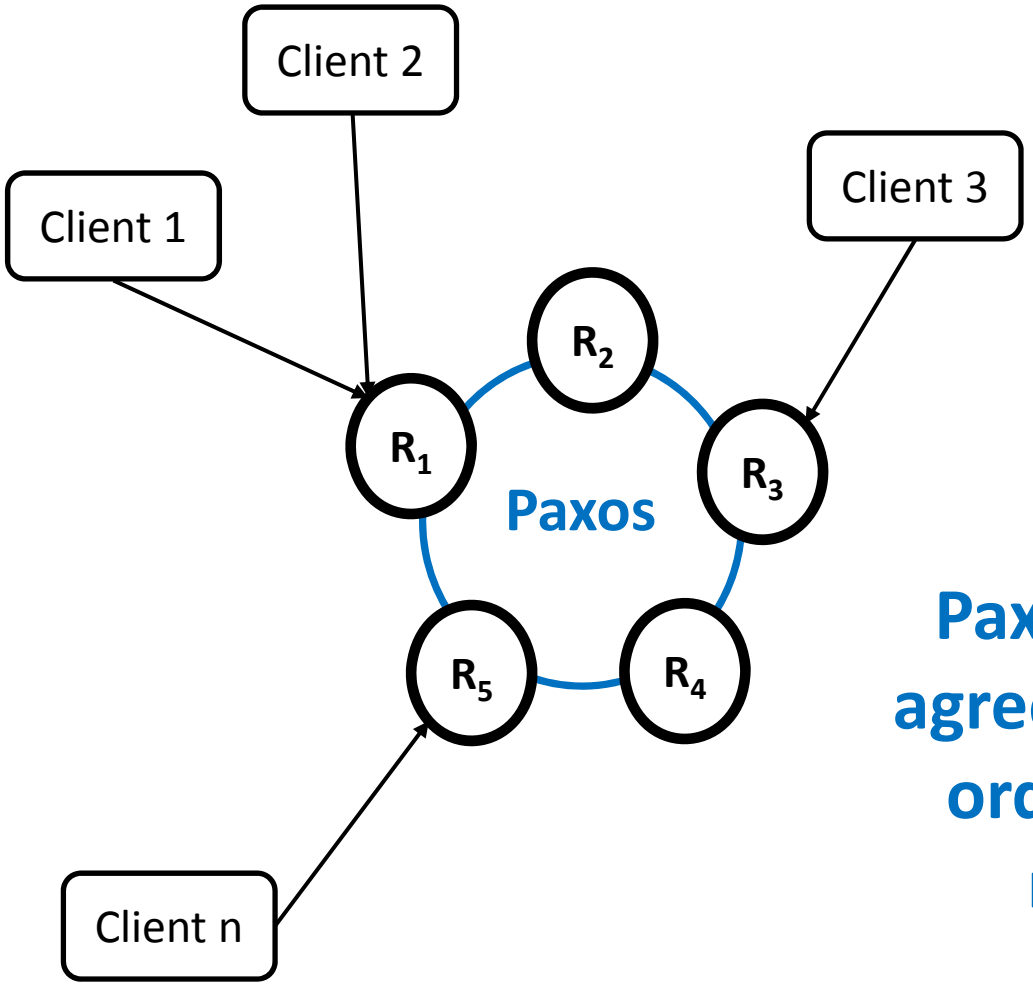
Replication



Paxos used to elect coordinator

Coordinator **orders** updates

Replication

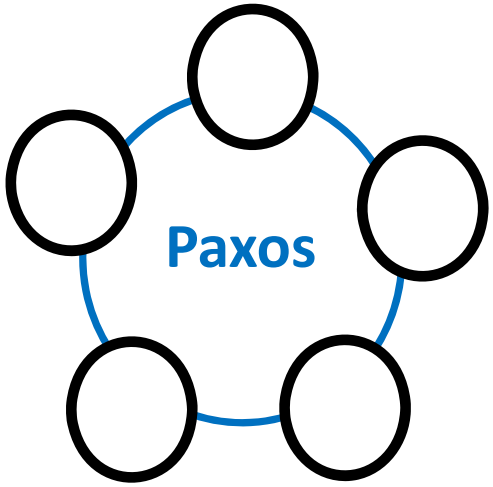


Paxos used to agree on update order among replicas

Coordination services
(Paxos inside)

Highly-available and persistent, a.k.a. distributed lock service

Chubby / ZooKeeper*

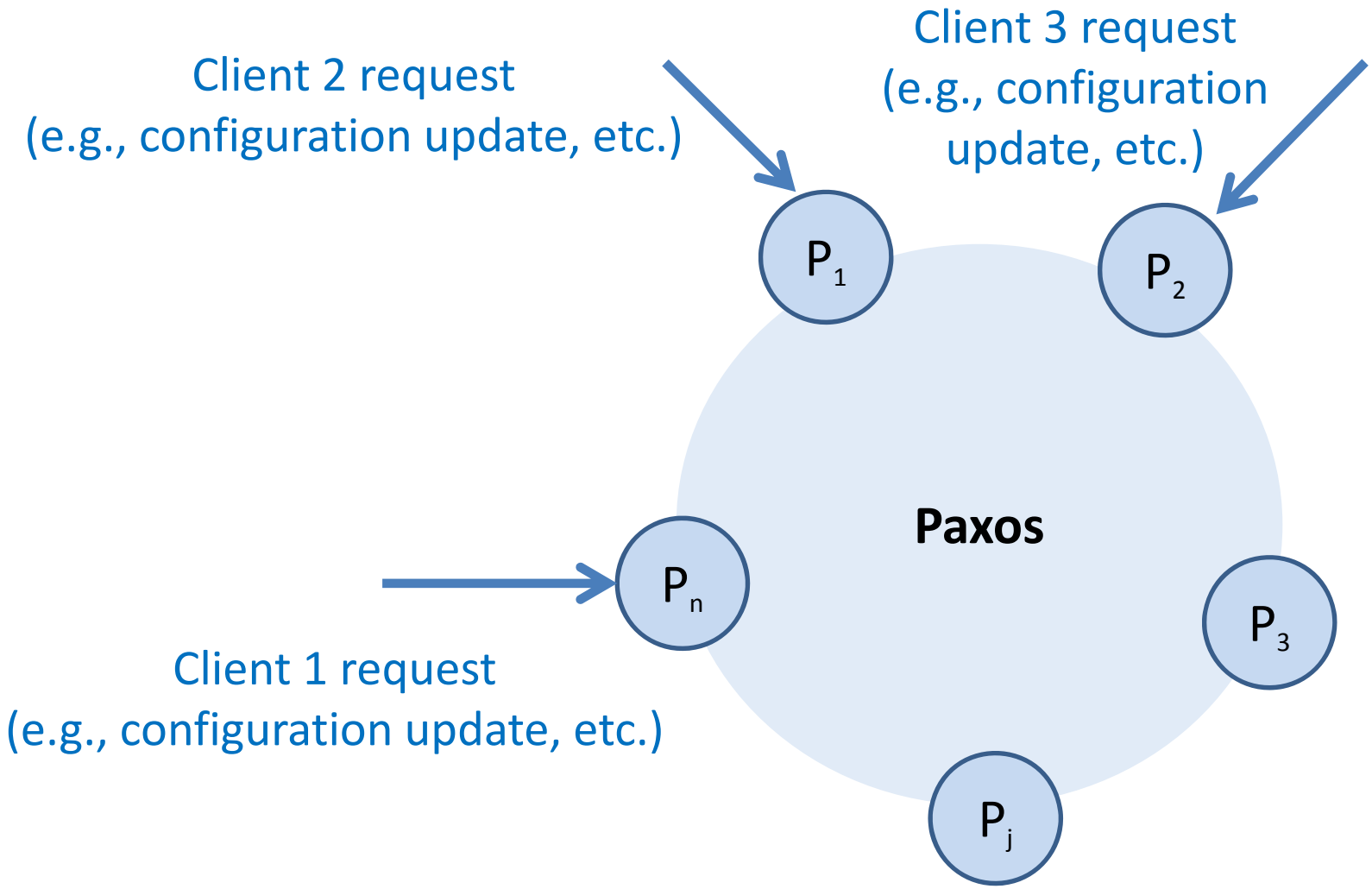


Use in Bigtable/HBase

- Ensure **at most one active** Bigtable **master** at any time
- Store **bootstrap** location of **data** (root tablet)
- **Discover** tablet servers (manage their lifetime)
- Store **configuration information**

*ZooKeeper uses ZAB (ZK. Atomic Broadcast, not Paxos)

As coordination service

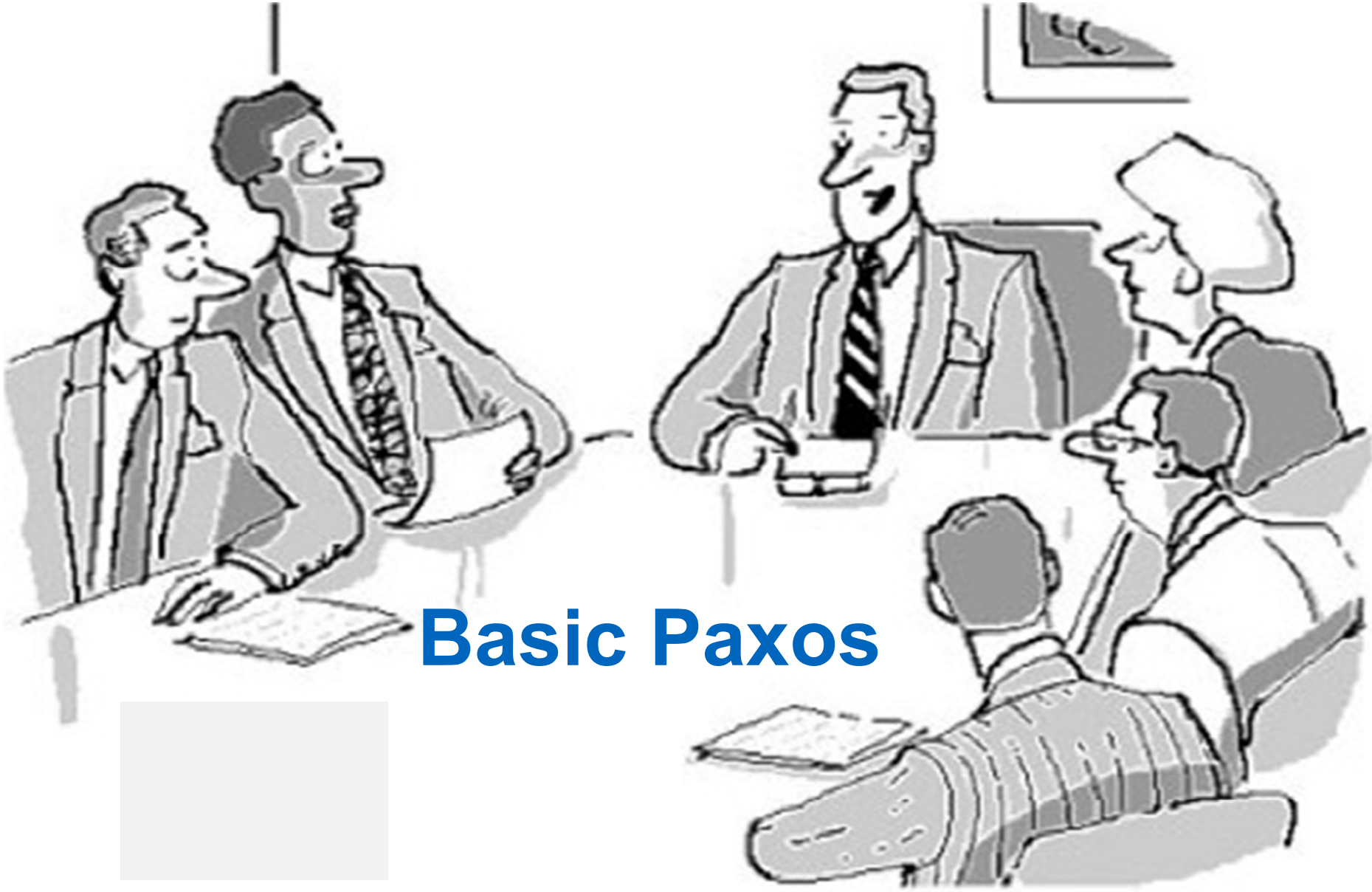


Consensus problem

- One or more node may propose some value. ***How do we get a collection of nodes to agree on exactly one of the proposed values?***

- Requirements for consensus:
  - **Agreement**: Every correct node (not failing) must agree on the same value.
  - **Validity**: Only proposed values can be decided. If a node decides on some value, *V*, then some node must have proposed *V*.
  - **Integrity**: Each node can decide a value at most once.
  - **Termination**: All correct nodes eventually decide on a value.

Basic Paxos



"Then we are agreed nine to one that we will say our previous vote was unanimous!"

Brief history of Paxos

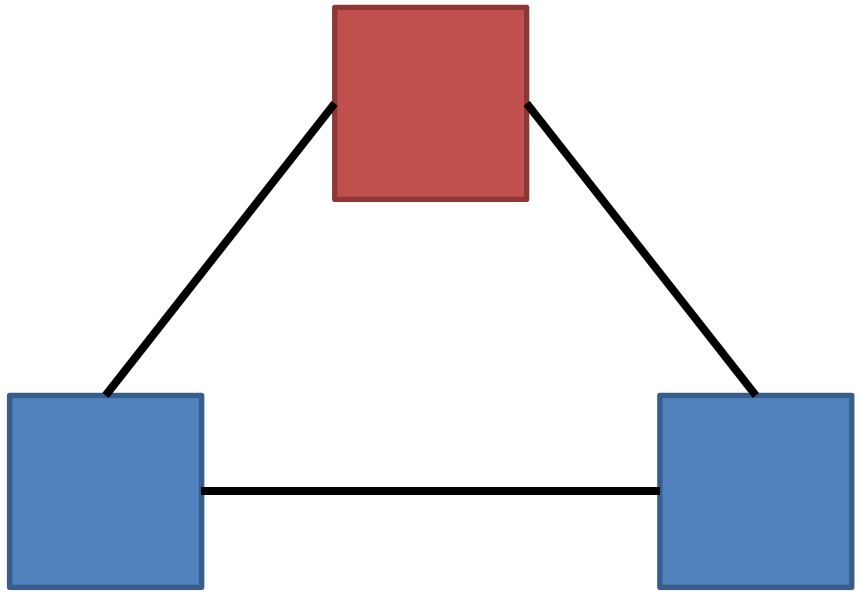
- Solves **consensus** in **asynchronous systems**, with **node** and **network omission failures***
  - Better than any algorithms we've seen so far!
  - But… (FLP, *cf.* next)*
- Original paper: "*The Part-Time Parliament*," by Lamport (submitted in 1990, published in 1998?!)
- Fictional legislative **consensus system** for the Greek island of Paxos
- Rewritten into "plain English" as "*Paxos Made Simple*"
- Many variations of the algorithm now exist in the Paxos family (cf. "Paxos family" slide)

*Paxos in light of FLP result

- In **asynchronous systems**, with even **one node crashing**, there is no guarantee to reach consensus.
- Paxos guarantees **safety,** but **not liveness**
- *"Conditions that could prevent progress are difficult to provoke."*
- Paxos **attempts to make progress** even during periods when some bounded number of nodes are unresponsive

Assumptions for Basic Paxos

- Nodes
- Network
- Number of nodes



Nodes

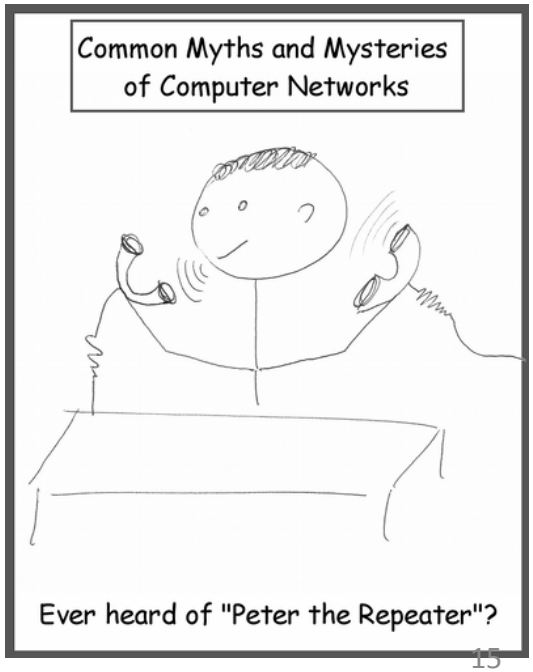
- Operate at **arbitrary speed** (i.e., asynchronous model)
- May experience **failures** (i.e., fail-stop faults)
- Nodes **may** re-join protocol after failures (i.e., following a **crash-recovery failure model;** remember their previous state)

- Do not collude, lie, or otherwise attempt to subvert the protocol (i.e., **no Byzantine failures**)

Network

- Nodes can send messages to any other node via unicast
- Messages are **sent asynchronously** (i.e., may take arbitrarily long)
- Messages may be **lost**, **reordered,** or **duplicated**
- Network may **partition**

- Messages are **delivered without corruption** (i.e., **no Byzantine failures**, cf. *Byzantine Paxos*)



Number of nodes

- In general, Paxos is functional given **2*f*+1** nodes (i.e., majority)
- Despite the **simultaneous failure** of any ***f*** nodes
- E.g., **5** nodes, **resilient against 2 failing (e.g., 5 replicas in Chubby)**

TOWARDS UNDERSTANDING PAXOS

Part 2

Roles in Paxos

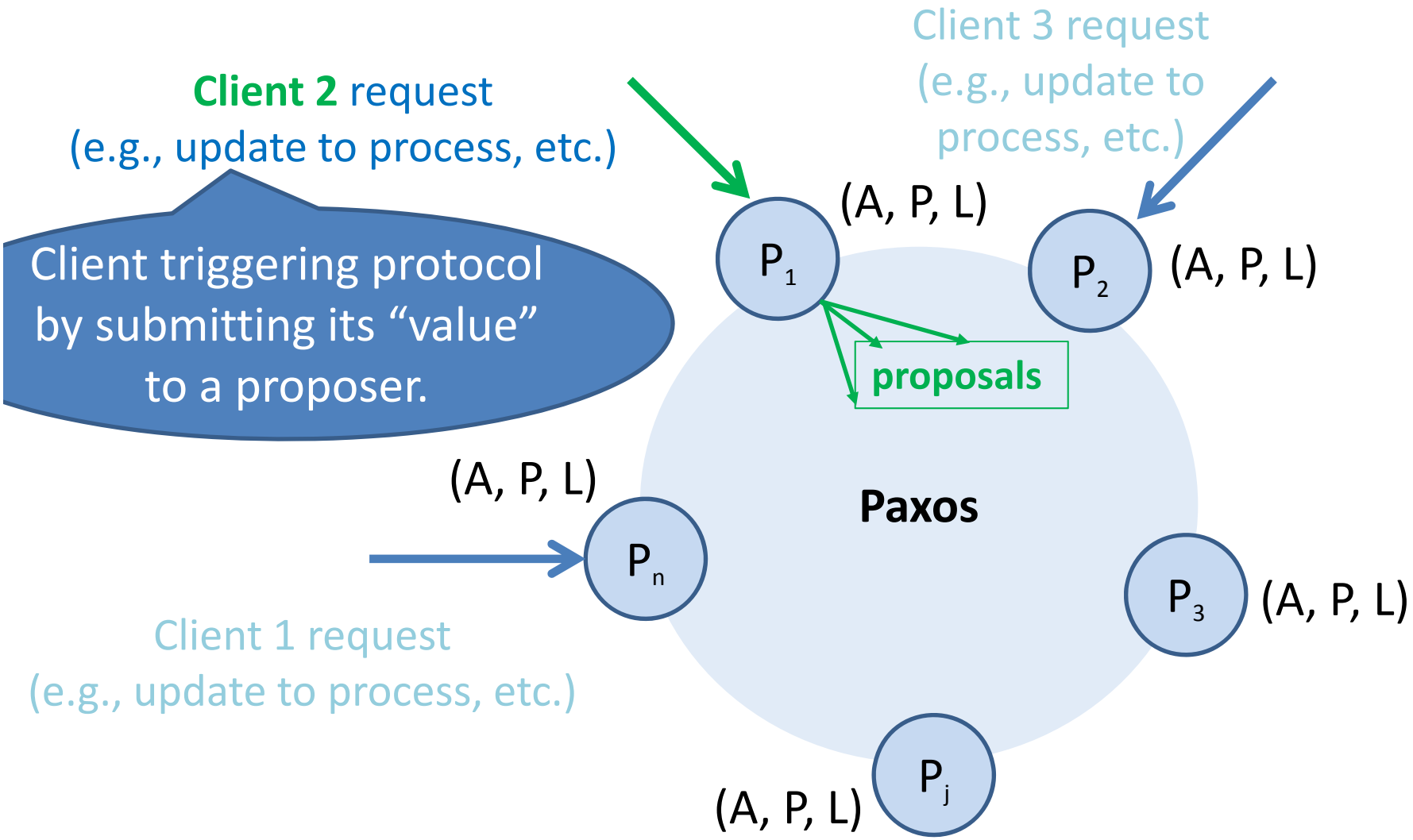
Like client & server roles in client-server

- Client triggers protocol
- Express protocol in terms of **roles**
  - **P**roposer
  - **A**cceptor
  - **L**earner
  - Leader (one of P)
- **Single process** may play **one or more roles** at the same time
- Has no effect on protocol correctness
- **Roles** are **commonly coalesced** to improve latency, number of messages exchanged, etc.

Paxos informal, high-level overview

- A **proposer** starts a new proposal by sending a ***proposal*** (message) to **acceptors**
- **Acceptors** send a ***promise*** back if they can accept the proposal
- If a **proposer** collects a **majority** of ***promises***, it sets a value to the proposal and sends it to a majority of **acceptors**
- **Acceptors** send the decided value to all **learners** if they can accept the value
- **Learners** learn the decided value when they collect a **majority** of messages from **acceptors** for a given value

Paxos in context: Roles, proposals, value

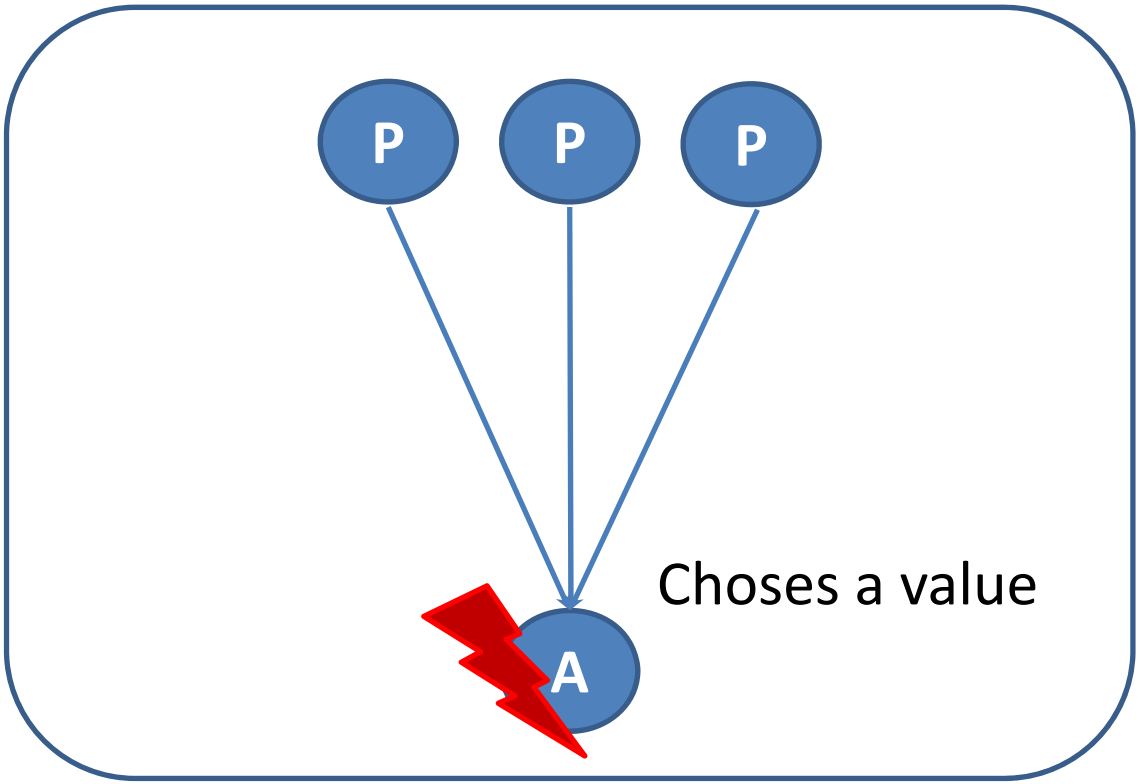


Corner cases addressed by Paxos

- *What if multiple proposers make competing proposals simultaneously?*
- *What if a proposer proposes a different values than an already decided value?*
- *What if there is a network partition?*
- *What if a proposer crashes in the middle of soliciting promises?*
- *What if a proposer crashes after collecting a majority of promises but before setting a value to its proposal?*
- …

Strawman 1

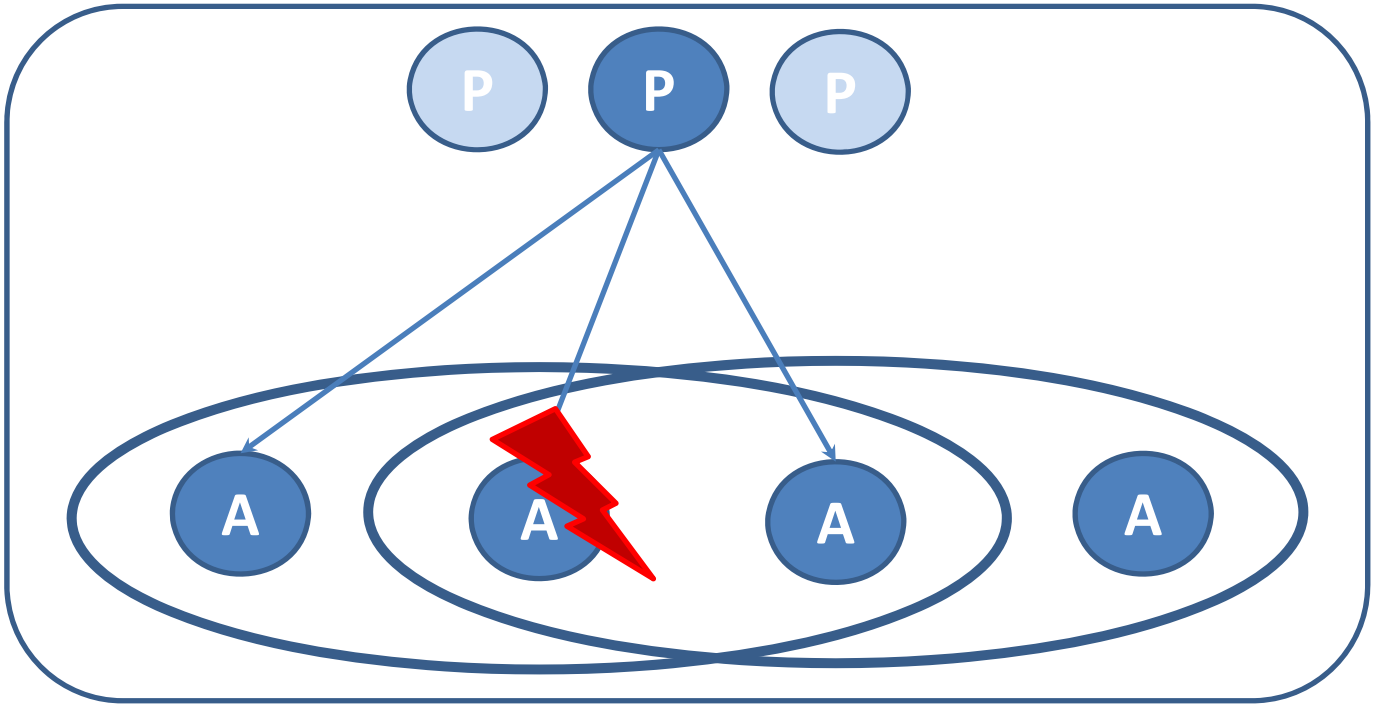
(Multiple proposers, single acceptor)



We need multiple acceptors to tolerate faults.

Strawman 2
(Multiple proposers, multiple acceptors)

- Proposer sends proposal to a **majority of acceptors** or more
- If a majority of acceptors choses a particular proposal (value) then that value is considered chosen



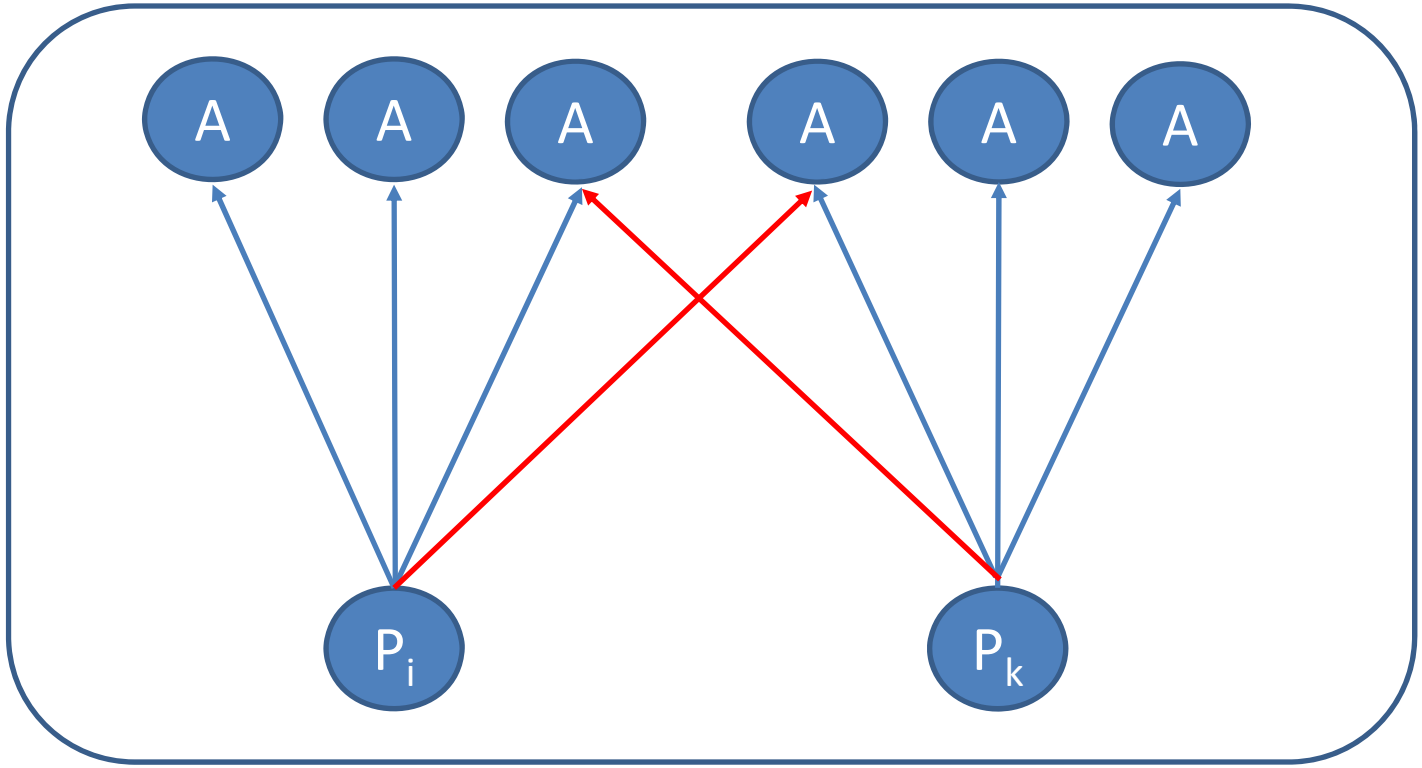
Strawman 2
(Multiple proposers, multiple acceptors)

- Proposer sends proposal to a **majority of acceptors** or more
- If a majority of acceptors choses a particular proposal (value) then that value is considered chosen
- ***But what proposal to chose?***
- Each acceptor **accepts the first proposal** it receives and rejects the rest
- A proposer who receives a majority of replies (i.e., votes) from acceptors chooses its own value

  → Only one proposal will get a majority, so only one value will be chosen
- Proposer sends the chosen value to everyone

Strawman 2

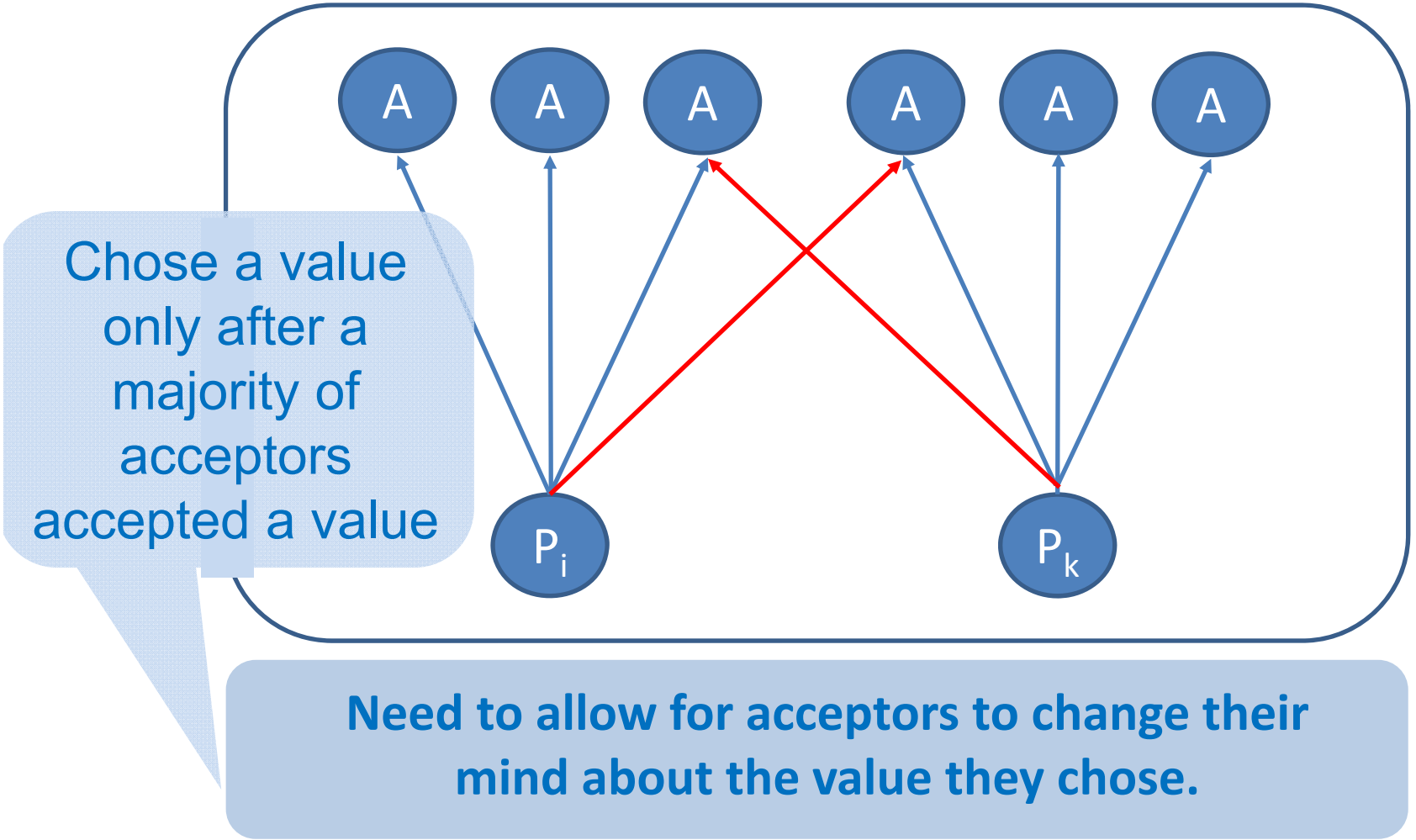
(Multiple proposers, multiple acceptors)



Multiple proposers send proposals concurrently, no one collects a majority!

Strawman 2

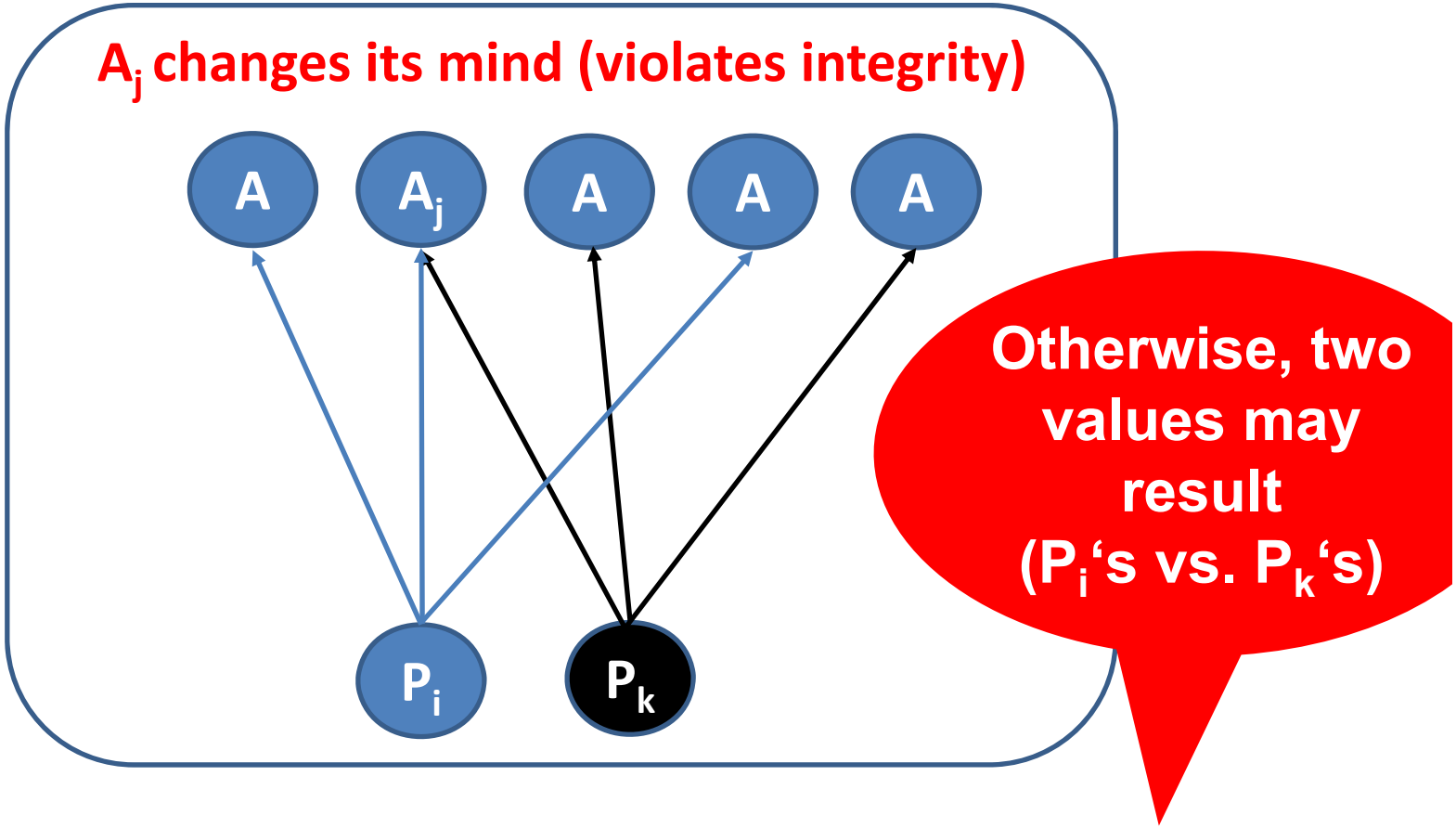
(Multiple proposers, multiple acceptors)



Need to allow for acceptors to change their mind about the value they chose.

Strawman 3

(Acceptors change their mind, accept any proposal)



Integrity: Each node can decide a value at most once.

Strawman 3

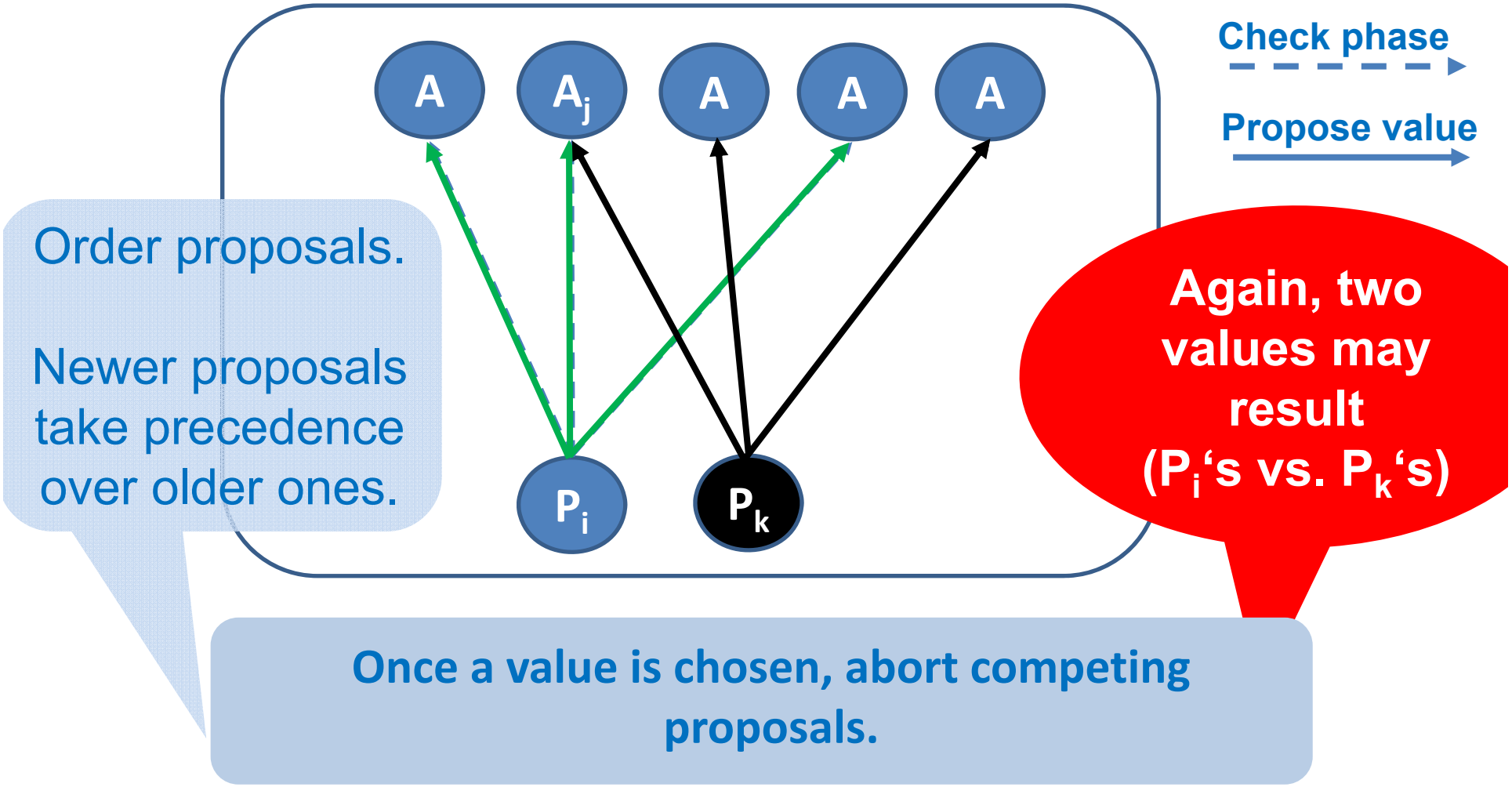
(Acceptors change their mind, accept any proposal)

- A proposer can't just propose a value

- Proposer needs to **first check with a majority of acceptors** whether a value has already been chosen
- If yes, proposer must adopt this value and propose it to a majority of …
- If no, proposer is free to propose its value

- **Requires a two-phase protocol**: Check, act accordingly

Strawman 4


A
A_j
A
A
A
P_i
P_k
Check phase
Propose value
Order proposals.
Newer proposals take precedence over older ones.
Again, two values may result (P_i's vs. P_k's)
Once a value is chosen, abort competing proposals.

Lessons learned from Strawmen 1-4

- Require multiple acceptors to tolerate faults
- Require votes on a value from a majority of acceptors to break ties
- Need to allow for acceptors to change their mind about the value they chose to establish a majority
- Require a two-phased protocol to check for prior decisions
- Require means to order proposals to ensure single value results

The Paxos way

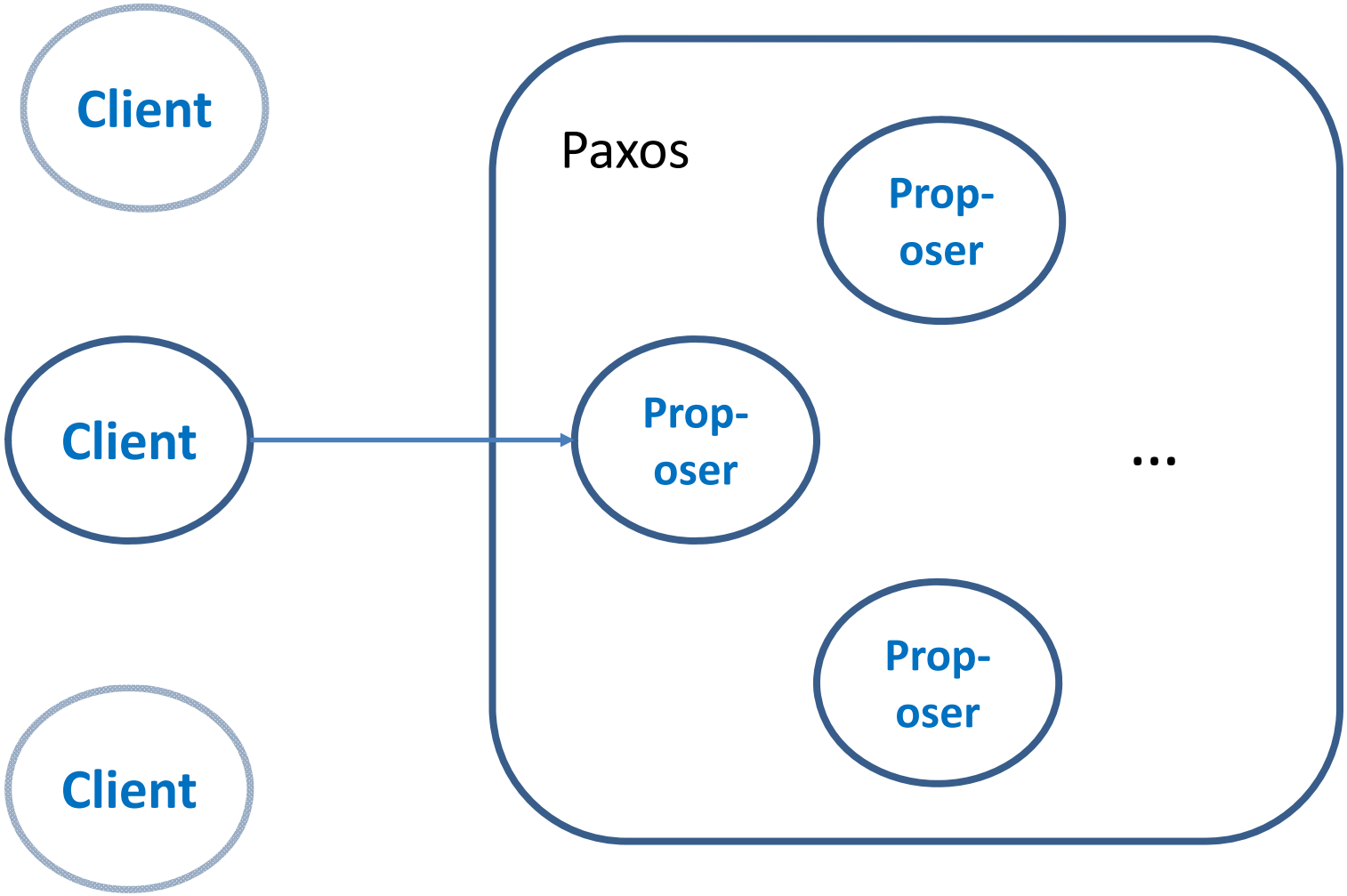
Preview

- Each acceptor may be able to accept multiple proposals **based on ordering**
- But, each acceptor must follow two rules:
  - If it **promised** to one proposal with a **P# *n***, it can only **accept** a **newer proposal** with a **P# *m***, where **m > n**.
  - If it promises to a newer proposal, it will ask the proposer to **set the value of the newer proposal** to the **same value as an older accepted proposal**.

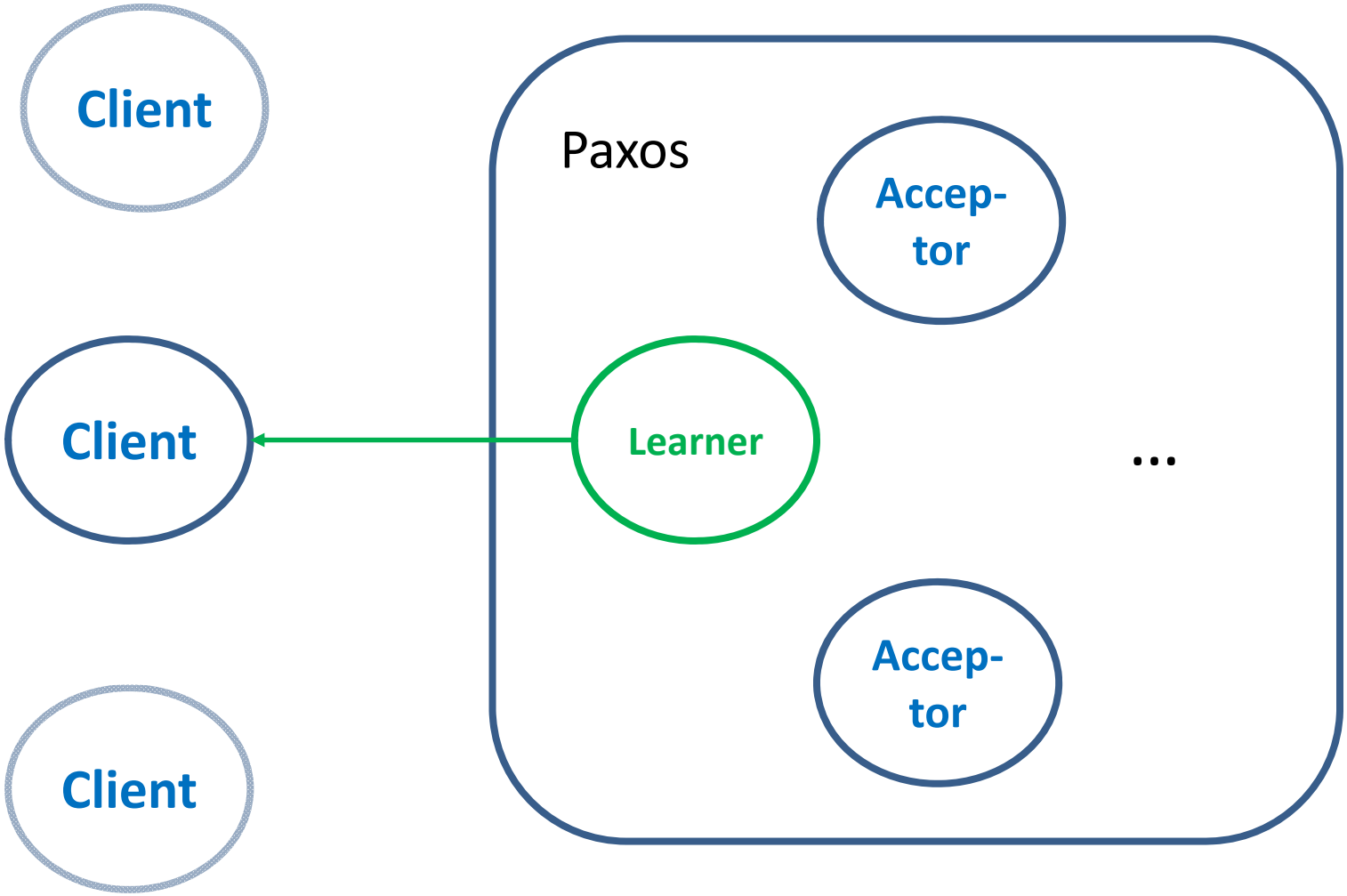
BASIC PAXOS IN DETAIL

Part 3

Clients



Clients



Clients

- Processes which interact with the Paxos protocol: **Do not participate in protocol**
- Clients send **requests to proposer** and wait for **response from learner**
- Basic Paxos **decides which request is accepted** (in multi-paxos, a sequence is decided)
- The nature of requests depends on how Paxos is used

Role of proposers in Paxos

- Convince acceptors to agree on **client requests**

- Paxos refers to **client request** simply as **values**
- Proposer issues proposals for a value to ultimately obtain agreement from Paxos
- There could be several different client requests (values) that are concurrently being proposed by different proposers
- One instance of Paxos determines one value that all correct nodes agree on

- Proposers need to initially check whether any other value has been decided on by acceptors (first phase of protocol)

Proposal number & agreed value

- **Messages** sent by proposer

  prepare (P# (N)) First phase

  acceptReq (P# (N), Value (V)) Second phase

  where ***P#*** refers to the **proposal number** (cf. below)

- Attempt to define an agreed value ***V*** via **proposals**
- Proposals may or may not be accepted by **acceptors**
- **Proposers** assign monotonically increasing numbers to each proposal (called the **P#**), e.g., 1,3,4,8,…*
- **Value *V*** represents the information that is to be agreed on (i.e., input from the client, client request)

*Proposal number

- **Must be unique** (required)
  - No two proposers can have the same number.
- Should be larger than any previous proposal number (by any proposer in instance of protocol) (**desirable**)
  - If this is not the case, proposer would find out by having its proposal rejected (proposer would have to re-try)
- Use a globally unique node/process identifier in lower-order bits and an incrementing counter in high-order bits
  - E.g., (logical clock, unique node/process identifier)-pair
  - E.g., (2345 1921681215555)

Acceptors

- **Messages** sent by acceptor
  - **promise** (*P#*, old P# (*N'*), old Value (*V'*)) — First phase
  - **accepted** (*P#*, *V*) — Second phase
- Acceptors decide which proposal to accept and memorize its value (i.e., write it to stable storage)
- Any proposal must be sent to a **quorum of acceptors (e.g., 3 out 5 acceptors)**
  - Usually, proposal is sent to every acceptor
- A proposal **must** be accepted by a **quorum** in order to pass
- Note: Paxos provides **consensus**, but **internally it only requires majority** from acceptors!
  - This makes this step more fault-tolerant than strict unanimity
  - I.e., tolerates failures

Quorum

- A quorum is a subset of acceptors such that two quorums share at least one member.
- Using quorum ensures there exists at least one acceptor who can decide between two proposals **(safety)**
- Typically, any **majority of participating acceptors**
- **Example**, given **acceptors** {A,B,C,D}:
  - **Majority quorum** would be any three **acceptors** (or more):

    {A,B,C}, {A,C,D}, {A,B,D}, {B,C,D}, {A,B,C,D}
  - **Weighted quorum**, if w_A = 2, else 1, weight 3 needed:

    {A,B}, {A,C}, {A,D}, {B,C,D}, {A,B,C,D}

Learner

- **Messages** sent by learner

  clientRes (Value (V))
- A **learner** acts as a storage point for decisions made by acceptors
- Paxos converts a decision made by only a **quorum** of acceptors into a consensus for **learners** (for Paxos)

- Because the decision has already been made by acceptors, it is easier for all learners to agree

Safety and liveness properties

- **Non-triviality**: Only proposed values can be learned
- **Safety**: At most one value can be learned (i.e., two different **learners** cannot learn different values)
- **Liveness**: If value *V* has been proposed, then eventually **learner** *L* will learn *some* value (if sufficient processes remain non-faulty)

- Paxos **ensures safety** property holds, regardless of the pattern of failures (subject to 2*f*+1 requirement)
- Paxos may **get stuck in a livelock** (then, no progress**; liveness property is not guaranteed**)

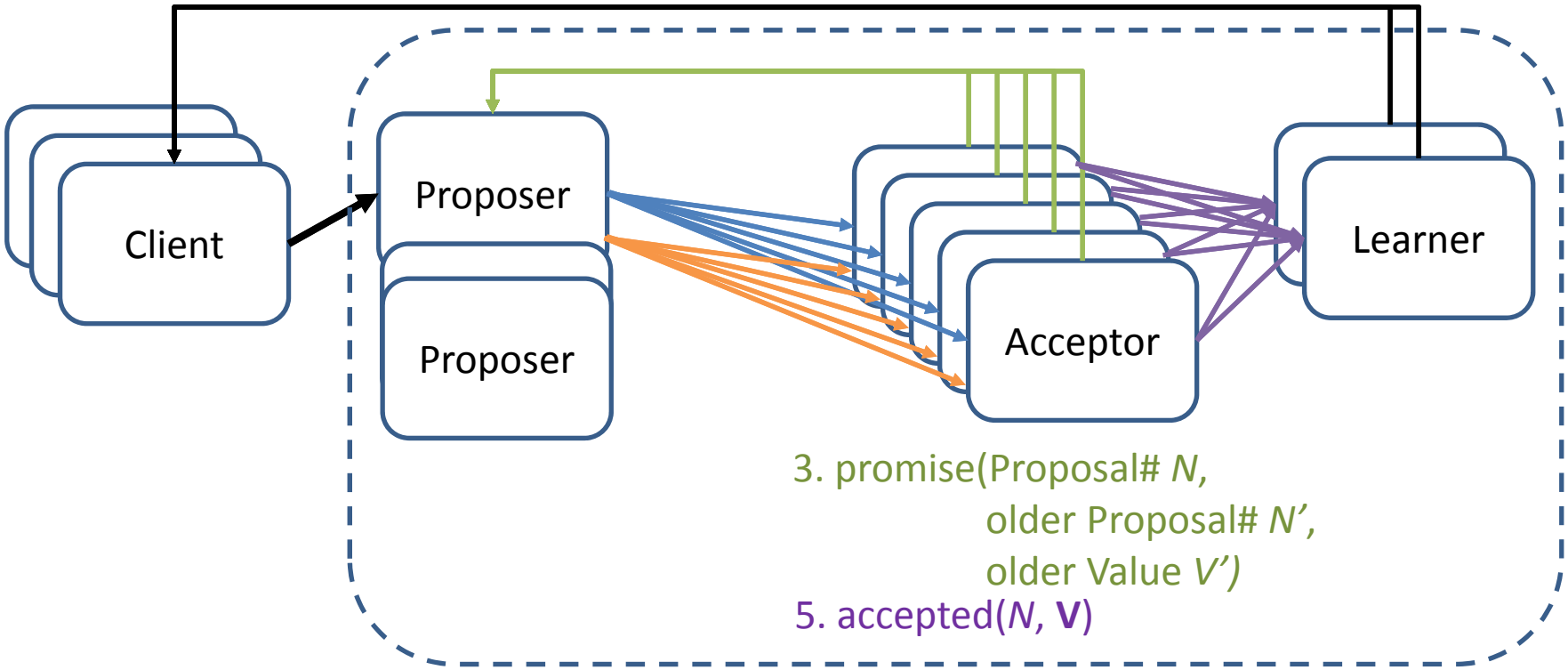
Summary of Paxos roles

1. clientReq (Value *V*)

2. prepare (Proposal# *N*)

6. clientRes (Decided Value *D*)

4. acceptReq (*N*, *V*)



Basic Paxos

- Each protocol invokation **instance** decides on a **single** output **value**
- Protocol may **proceed** over several **rounds**
- A successful round has **two phases**
  - prepare (P) and promise (A) (first phase)
  - acceptRequest (P) and accepted (A) (second phase)

- Once learner collects a majority of accepts for a value it issues clientResponse (L) (“third phase”)

- **Proposer** needs to communicate with at least a **quorum of acceptors**

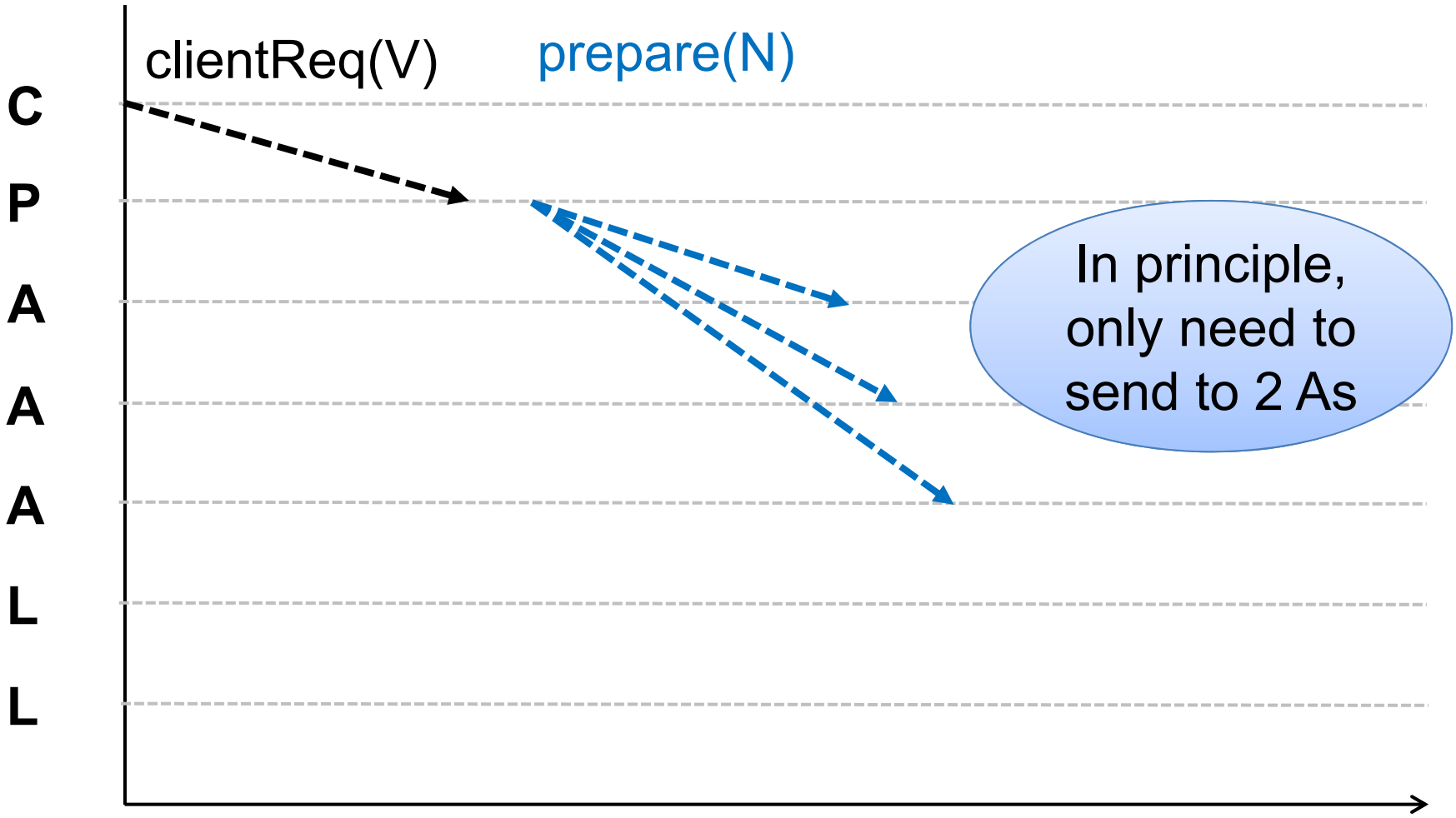
Phase 1a: Prepare (@ Proposers)

- **Proposer** creates proposal identified by P# *N*
- Any proposer could send proposals at any time
- **P# *N*** must be **greater than any previous P# *N'*** by this proposer
- **Proposer** sends a **prepare message** with P# *N* to quorum of **acceptors**:

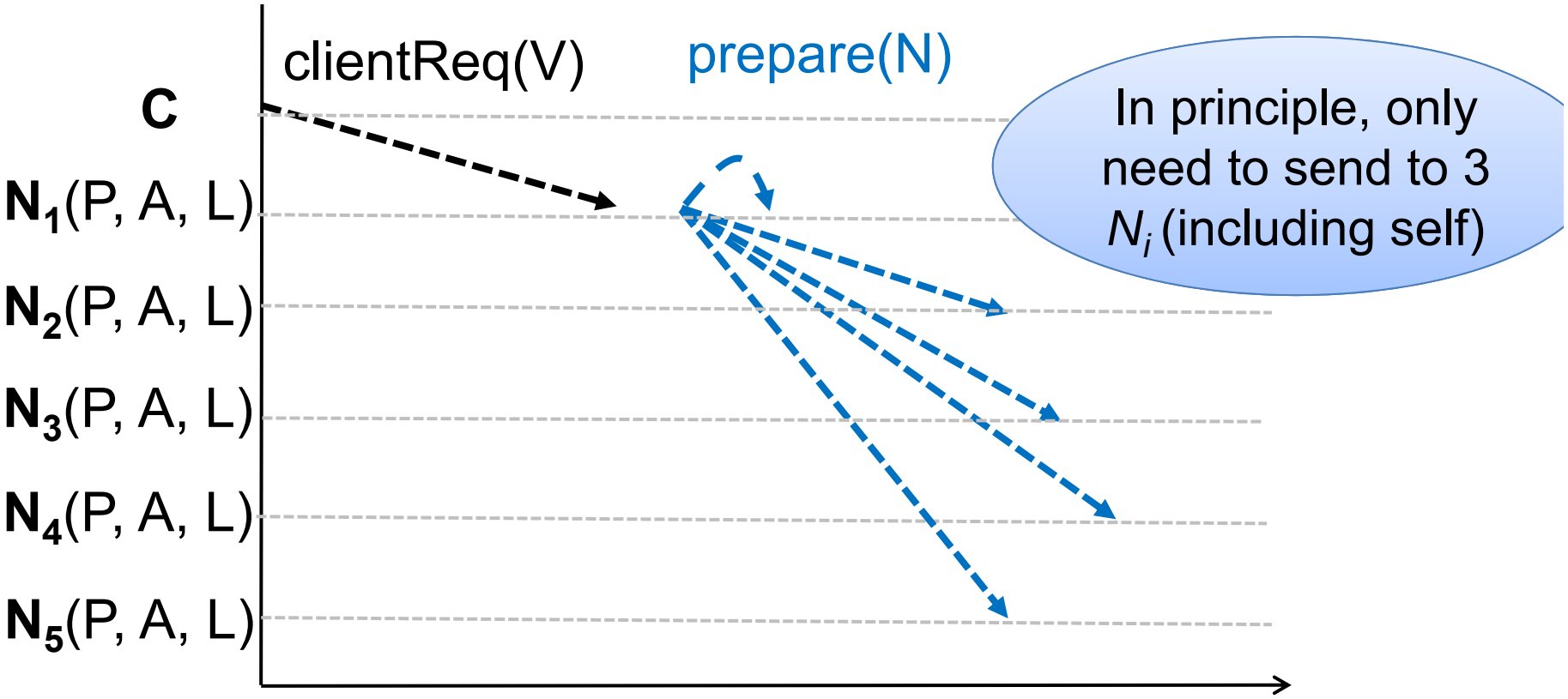
  prepare(N)
- **Proposer** decides who (among the acceptors) is in the quorum (could be every acceptor)

I'm not sure we have a quorum now that half our Board has been arrested.

Phase 1a: Prepare



Phase 1a: Prepare
(Roles coalesced)



Phase 1b: Promise (@ Acceptors)

- If ***N*** is larger than any previous ***N'*** received from **any proposer** by this acceptor, then the acceptor must return a **promise** to **ignore all future proposals having a number less than *N***

  promise(N, -, -)

- If the **acceptor** accepted a proposal at some point in the past, it must include the previous ***N'*** and previous value ***V'*** in its response to the **proposer**

  promise(N, N', V')

- Otherwise, the **acceptor** sends a negative reply (NACK)

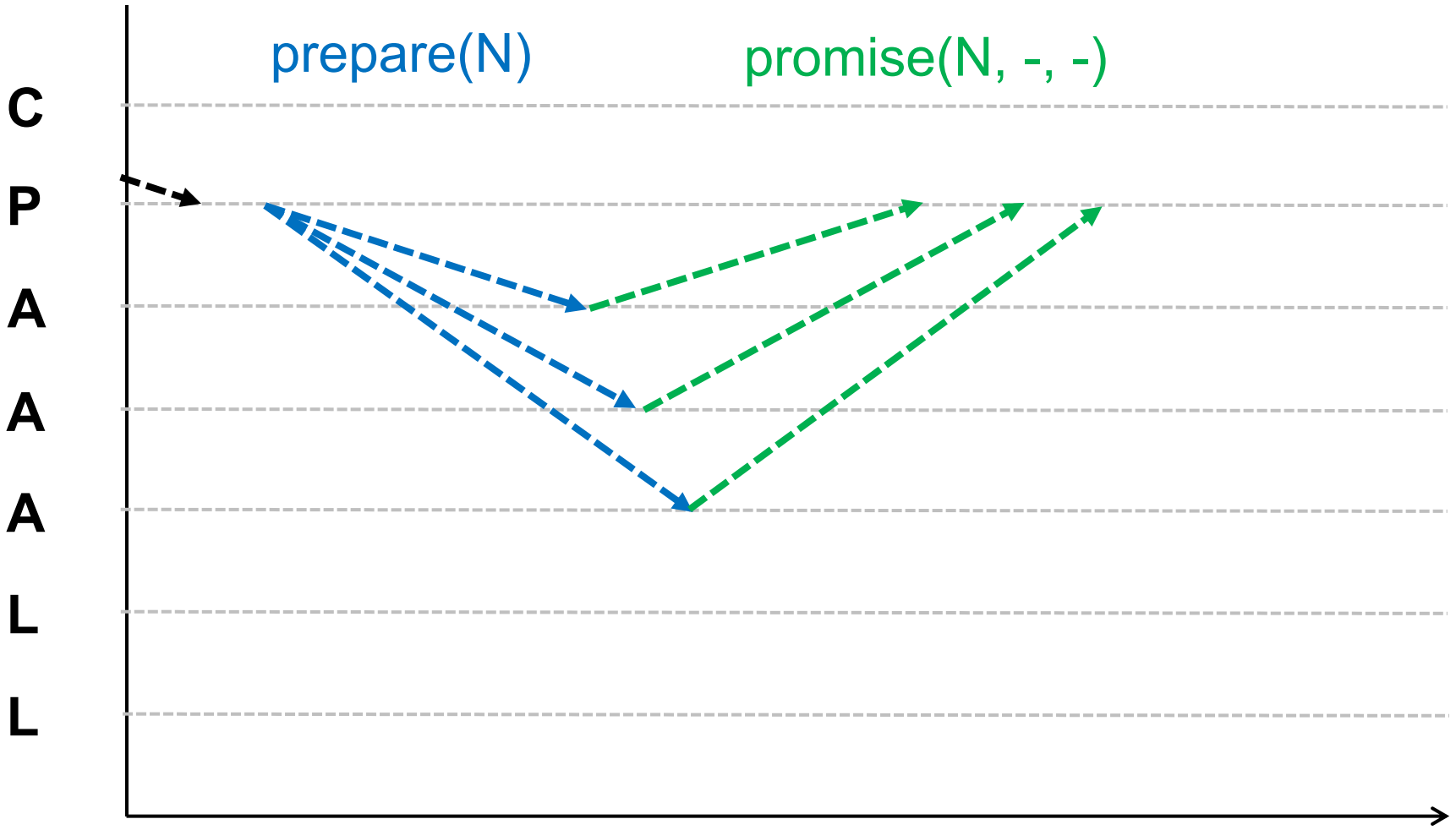
Phase 1b: Promise (@ Acceptors)

- If ***N*** is larger than any previous ***N'*** received from **any proposer** by this acceptor, then the acceptor must

```
// N larger than any P# previously received
IF N > max_PN
   // Yes, remember P#, reply with promise
   max_PN = N
   reply: promise(N,  -,  -)
ELSE
   // No
   do not reply (or reply with a NACK)
```

- Otherwise, the **acceptor** sends a negative reply (NACK)

Phase 1b: Prepare



Phase 1b: Promise (@ Acceptors)

Acceptor accepted proposal in the past

- If ***N*** is larger than any previous ***N'*** received from **any proposer** by this acceptor, then the acceptor must return a **promise** to **ignore all future proposals having a number less than *N***

  promise(N, -, -)

- If the **acceptor** accepted a proposal at some point in the past, it must include the previous ***N'*** and previous value ***V'*** in its response to the **proposer**

  promise(N, **N'**, **V'**)

- Otherwise, the **acceptor** sends a negative reply (NACK)

Phase 1b: Promise (@ Acceptors)

Acceptor accepted proposal in the past

- If N is larger than any previous N' received from any

```
// N larger than any P# previously received
// N > max_PN
IF N ≤ max_PN
   //!(N > max_PN)
   do not reply (or reply with a NACK)
ELSE
   //(N > max_PN), was proposal accepted before?
   IF (proposal_accepted == true)
        reply: promise(N, N', V')
   ELSE
        reply: promise(N,  -,  -)
```

Phase 2a: Accept Request (@Proposer)

- If a **proposer** receives **enough promises** from **acceptors** (a quorum), it needs to **set a value to its proposal**

- If any acceptor had previously accepted a proposal, then it would have sent its values to the **proposer**, who now must **set the value** of its **proposal** to the value associated with the **highest proposal number** reported by any **acceptor**

Phase 2a: Accept Request (@Proposer)

cont.'d.

- If none of the **acceptors** had accepted a proposal up to this point, then the **proposer** may **choose any value** for its proposal (i.e., the one from the client request)

- **Proposer** sends an **accept request message** to a **quorum of acceptors** with the chosen ***V*** (value) for its proposal

acceptReq(N, V)

- **Proposer** may choose **quorum** from all acceptors in the system, not just ones that promised back

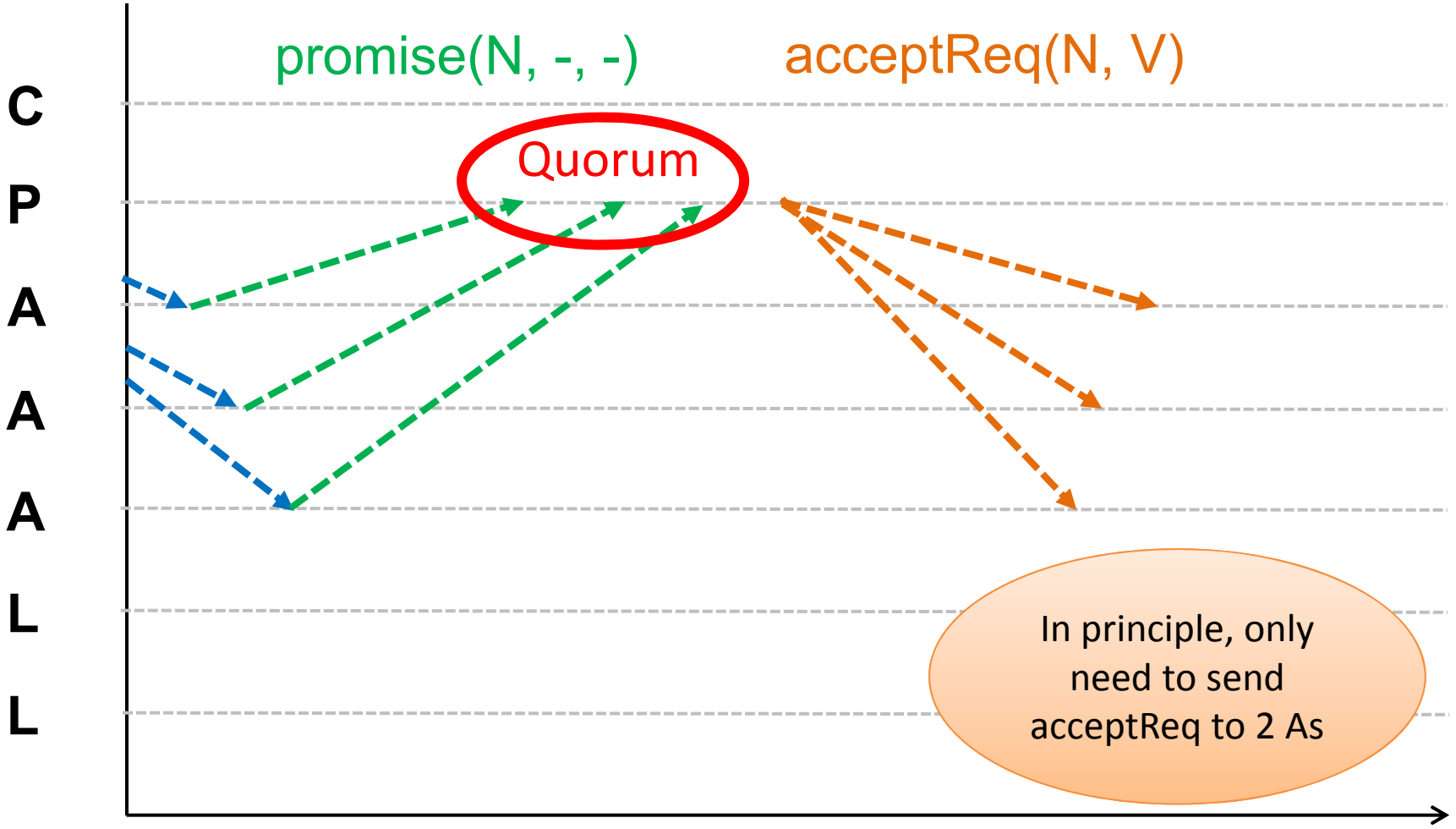
Phase 2a: Accept Request (@Proposer)

cont.'d.

```
Were promises from a majority of acceptors
received?
IF yes
   Did any reply contain accepted values
    (from other proposals)?
   IF yes
       //Take value from promise with highest PN
       V = accepted_VALUE
   ELSE
       // Use own value (client's input)
       V = OUR_VALUE
   reply acceptReq(N, V) to majority of acceptors
    (or more)
```

Phase 2a: Accept Request

Quorum size is 2



Phase 2b: Accepted (@Acceptor)

(Accept Request – acceptReq)

- If an **acceptor** receives an **acceptReq message** for a proposal number ***N***, it must accept it ***if and only if*** it has not already promised to only consider proposals having an identifier greater than ***N***

- In this case, it registers the corresponding value ***V*** and sends an **accepted message** to the **proposer** and **every learner**

  accepted(N, V)

Optional

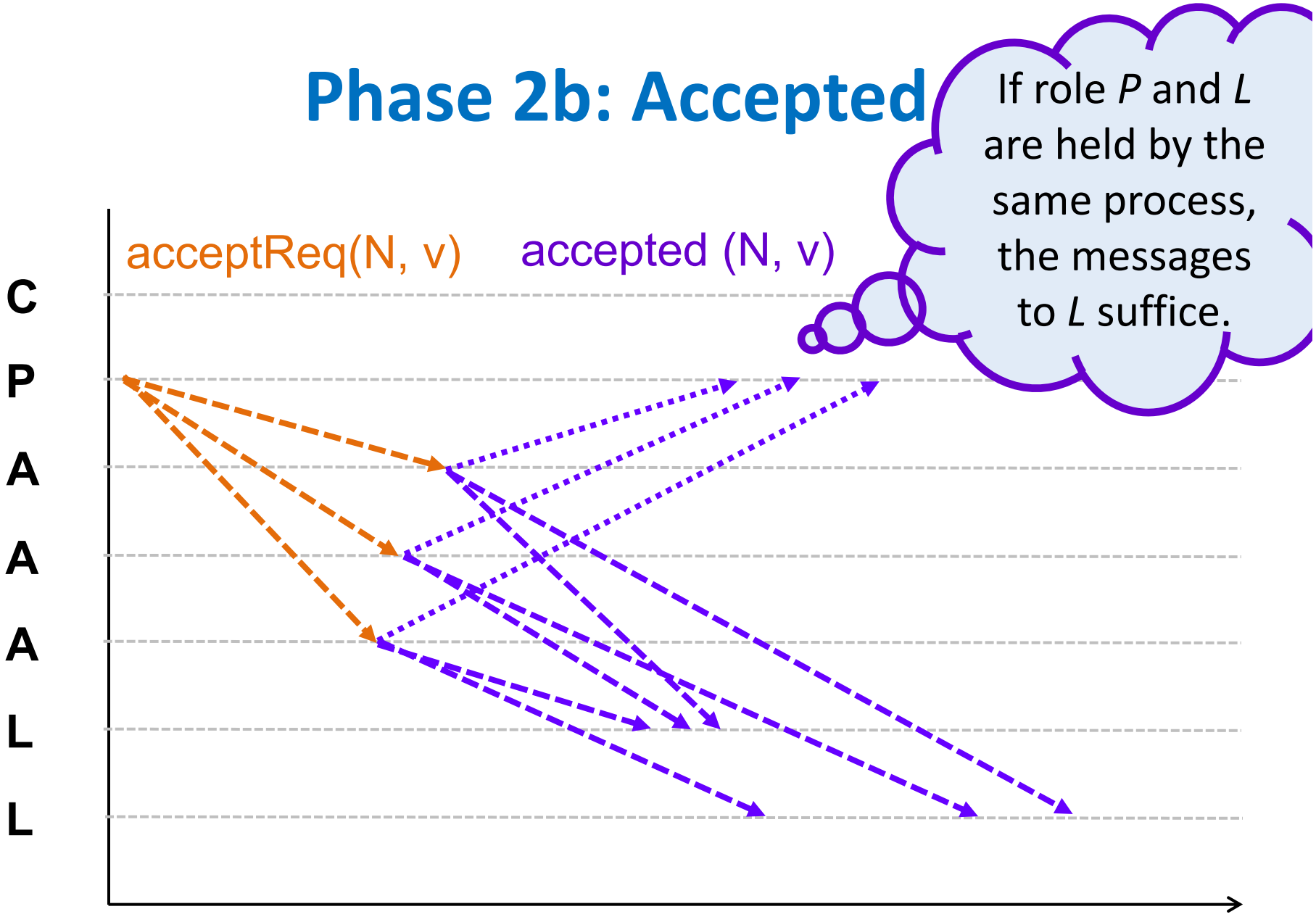
- Otherwise, it **ignores** the **acceptReq** and sends a **NACK** to the proposer

Phase 2b: Accepted (@Acceptor)

(Accept Request – acceptReq)

```
// Is P# N the largest seen so far?
IF (N == max_PN)
    // note that we accepted a proposal
    proposal_accepted = true
    // save the accepted proposal number
    accepted_ID = N
    // save the accepted proposal value
    accepted_VALUE = V
    reply: accepted(N, V) to proposer & learners
ELSE
    do not respond (or respond with a NACK)
```

Phase 2b: Accepted



Notes on Phase 2b

- An acceptor can accept **multiple proposals**

- **Paxos guarantees that acceptors ultimately agree on a single value**

Notes on Phase 2b

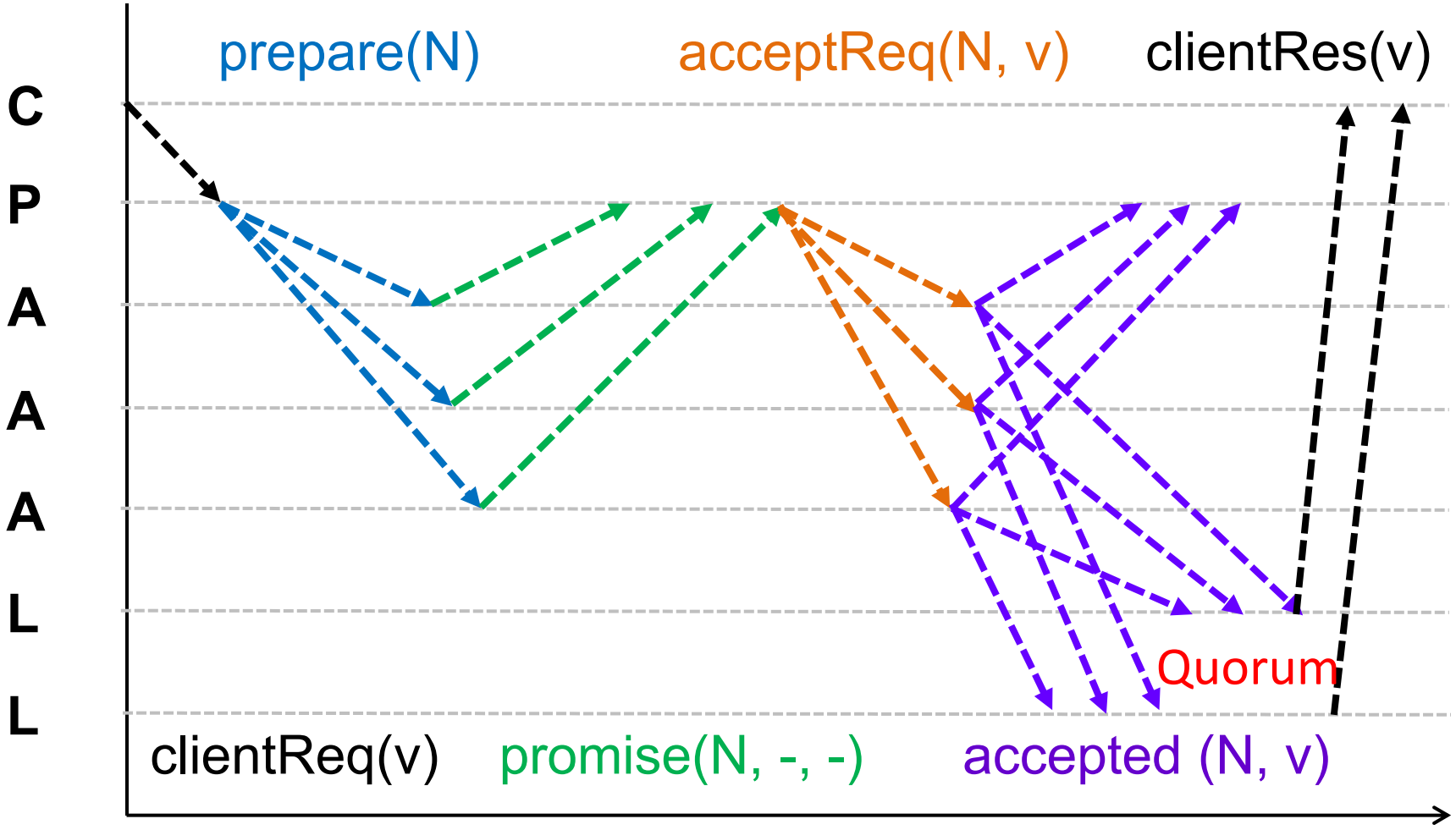
- Rounds fail when **multiple** proposers send **conflicting prepare messages** which invalidate **acceptReq** messages
- Rounds fail when proposer **does not receive a quorum of replies** (promise or accepted messages)

- In these cases, **another round must be started** with a higher proposal number

Phase 3: Decided (@Learner)

- A learner **learns** a decided value if it receives accept messages from a **quorum** of **acceptors**
- A quorum is required to avoid conflicting accepts from multiple concurrent proposals
- Once a learner has **learned** a decided value, it can execute the associate command and/or communicate with a client

Basic Paxos protocol timeline



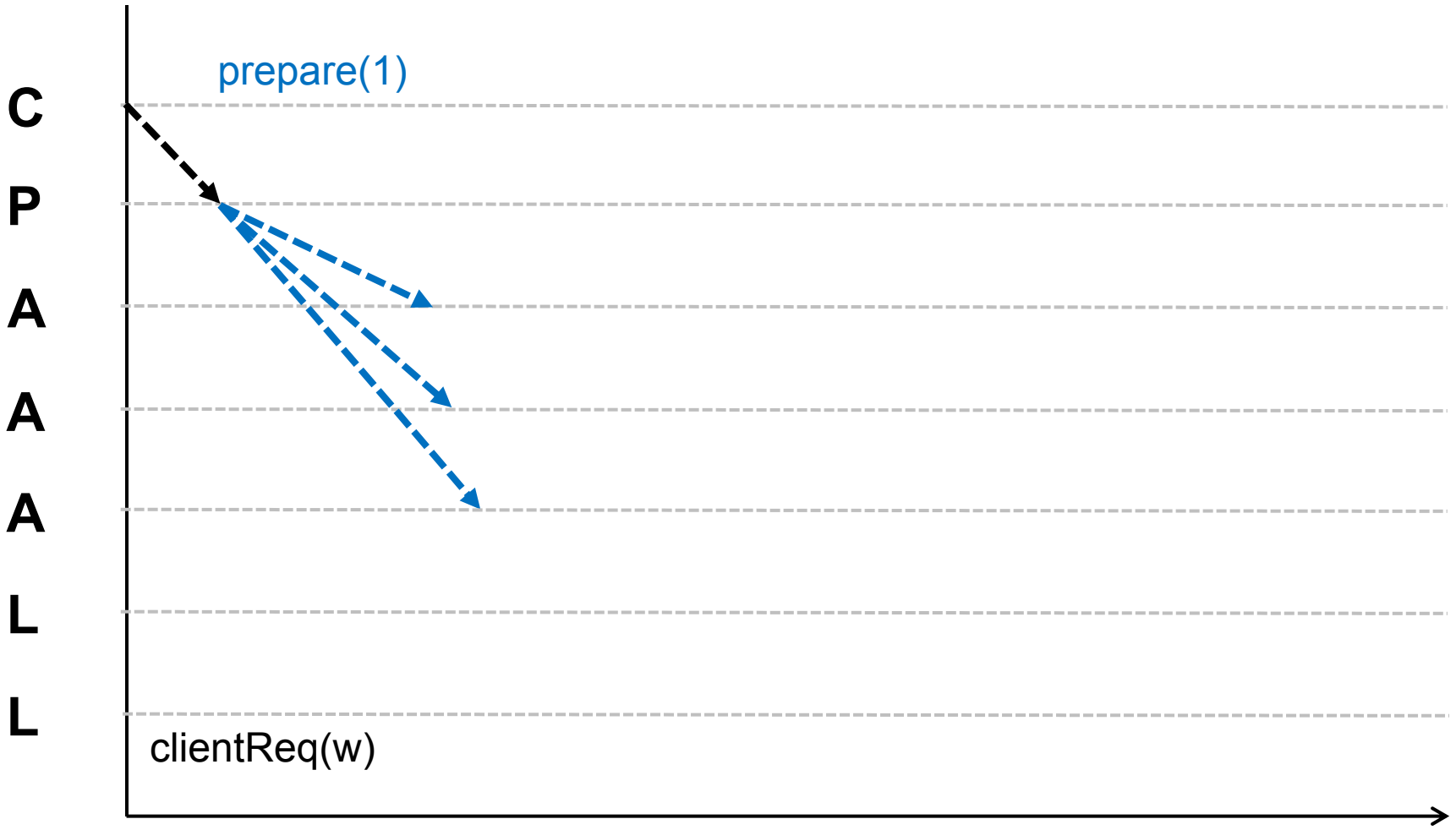
FAILURE HANDLING IN PAXOS

Part 4

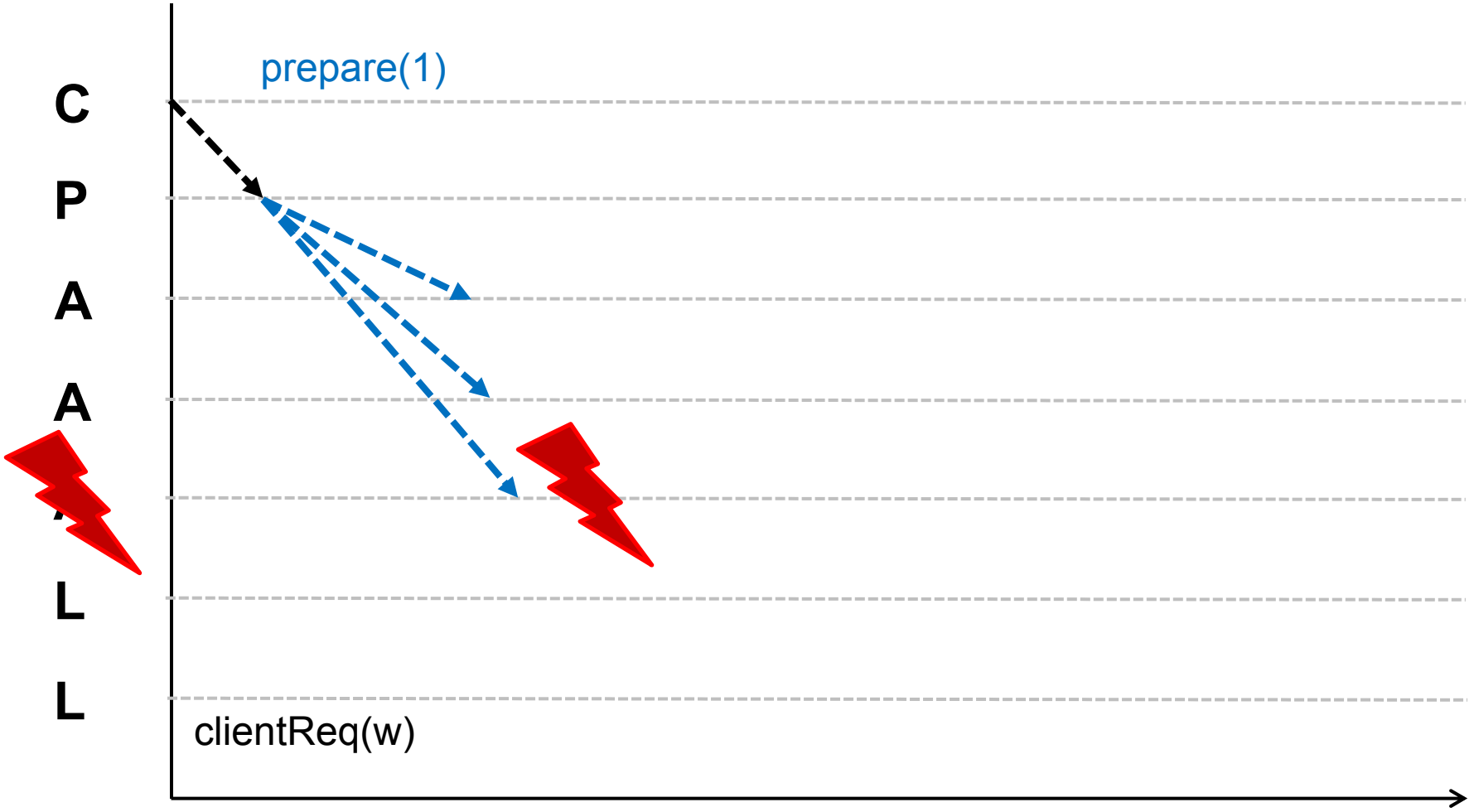
Failure scenarios

- Failure of **acceptor**
- Failure of redundant **learner**
- Failure of **proposer**
- Dueling **proposers**
- Communication failures

Basic Paxos: Failure of acceptor I

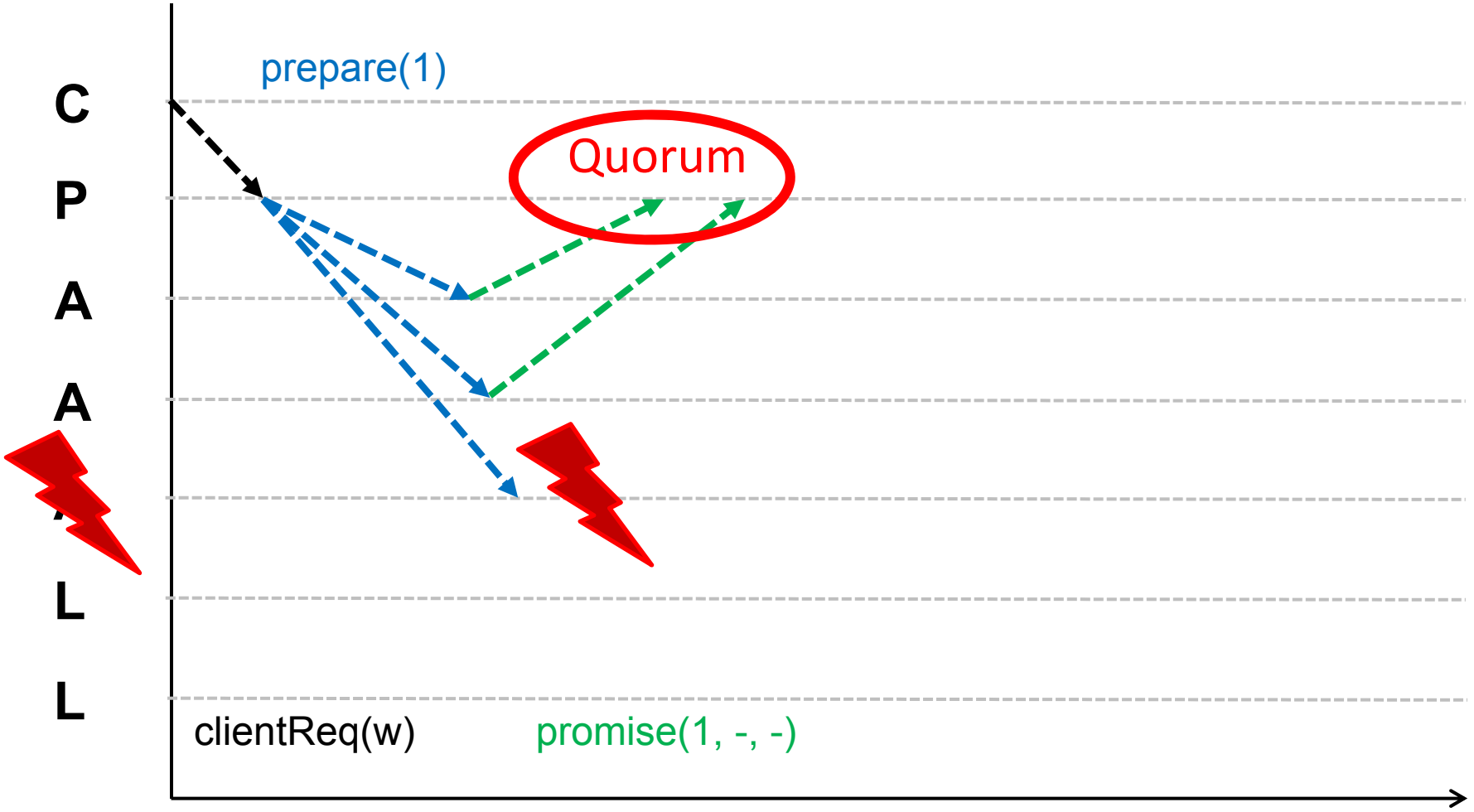


Basic Paxos: Failure of acceptor II

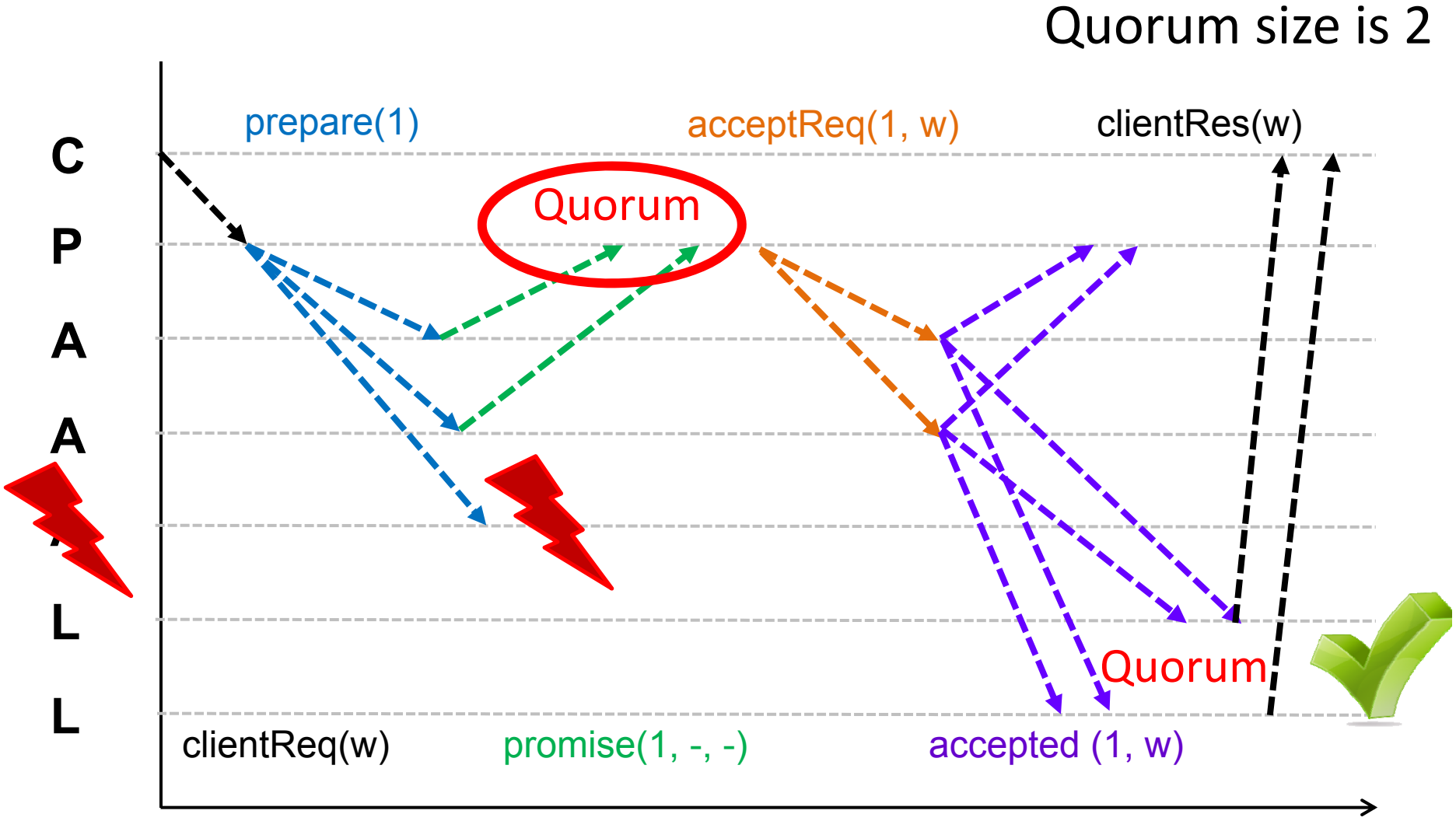


Basic Paxos: Failure of acceptor III

Quorum size is 2

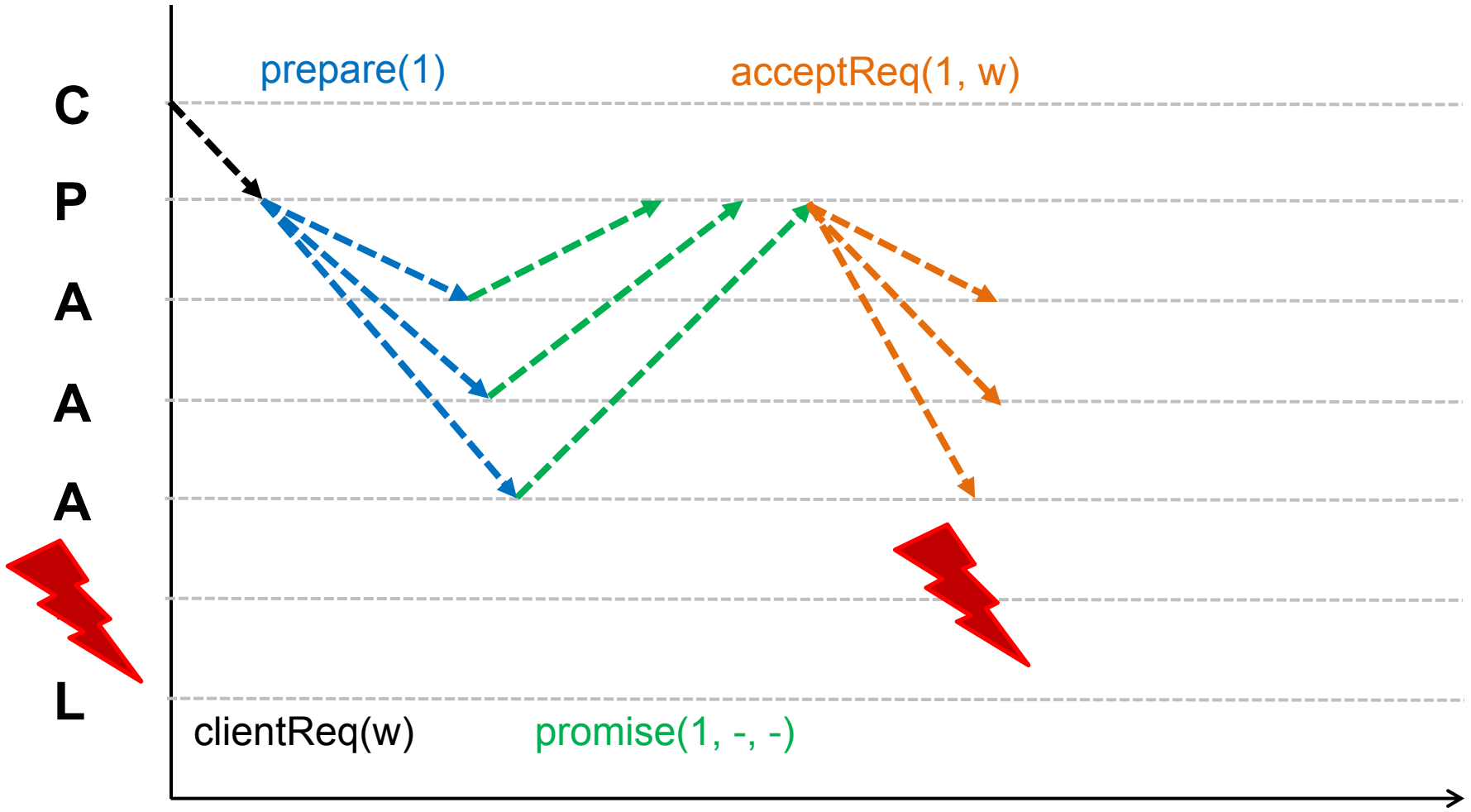


Basic Paxos: Failure of acceptor IV

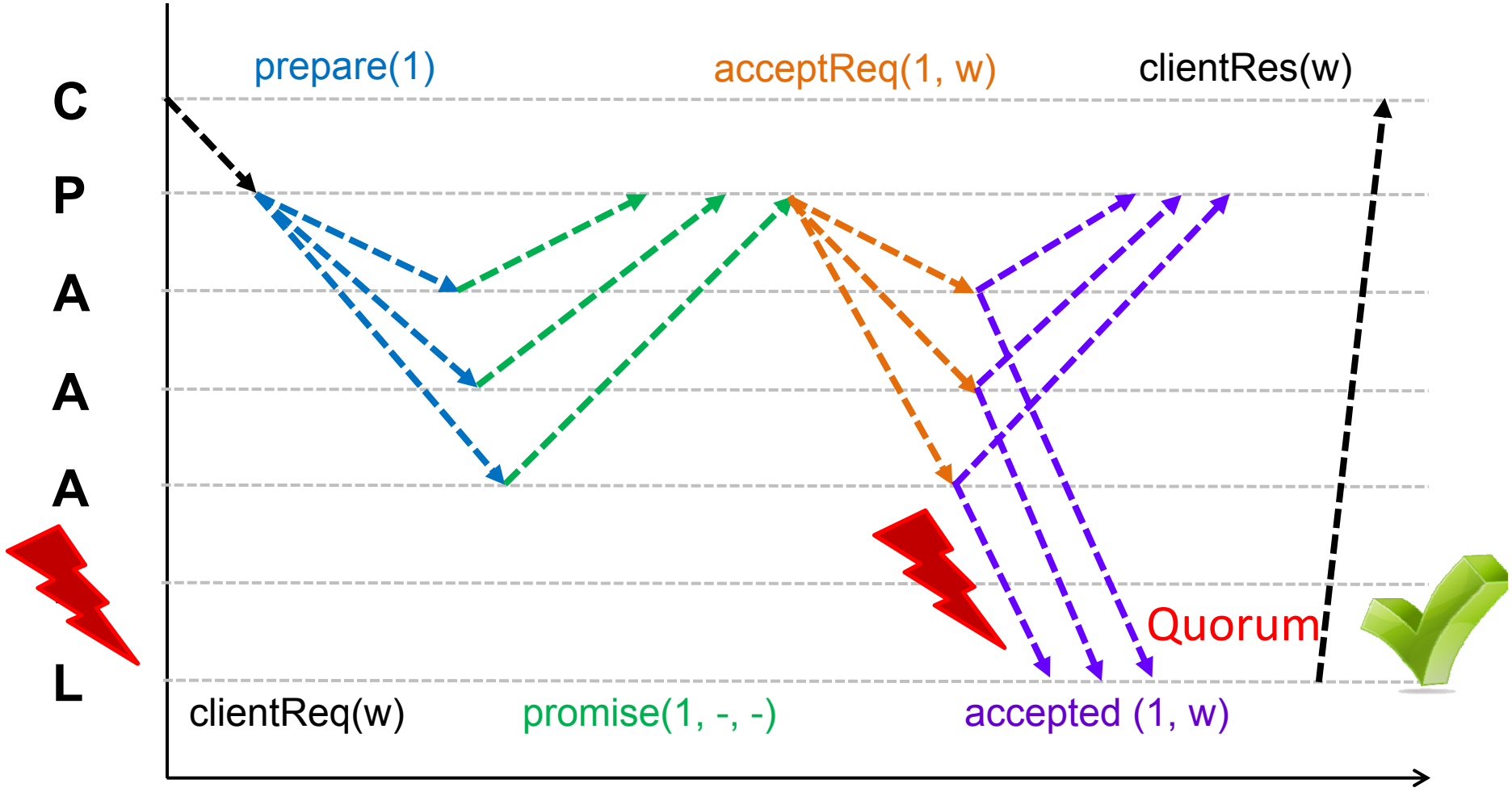


Basic Paxos: Failure of redundant learner

Failure of redundant learner I



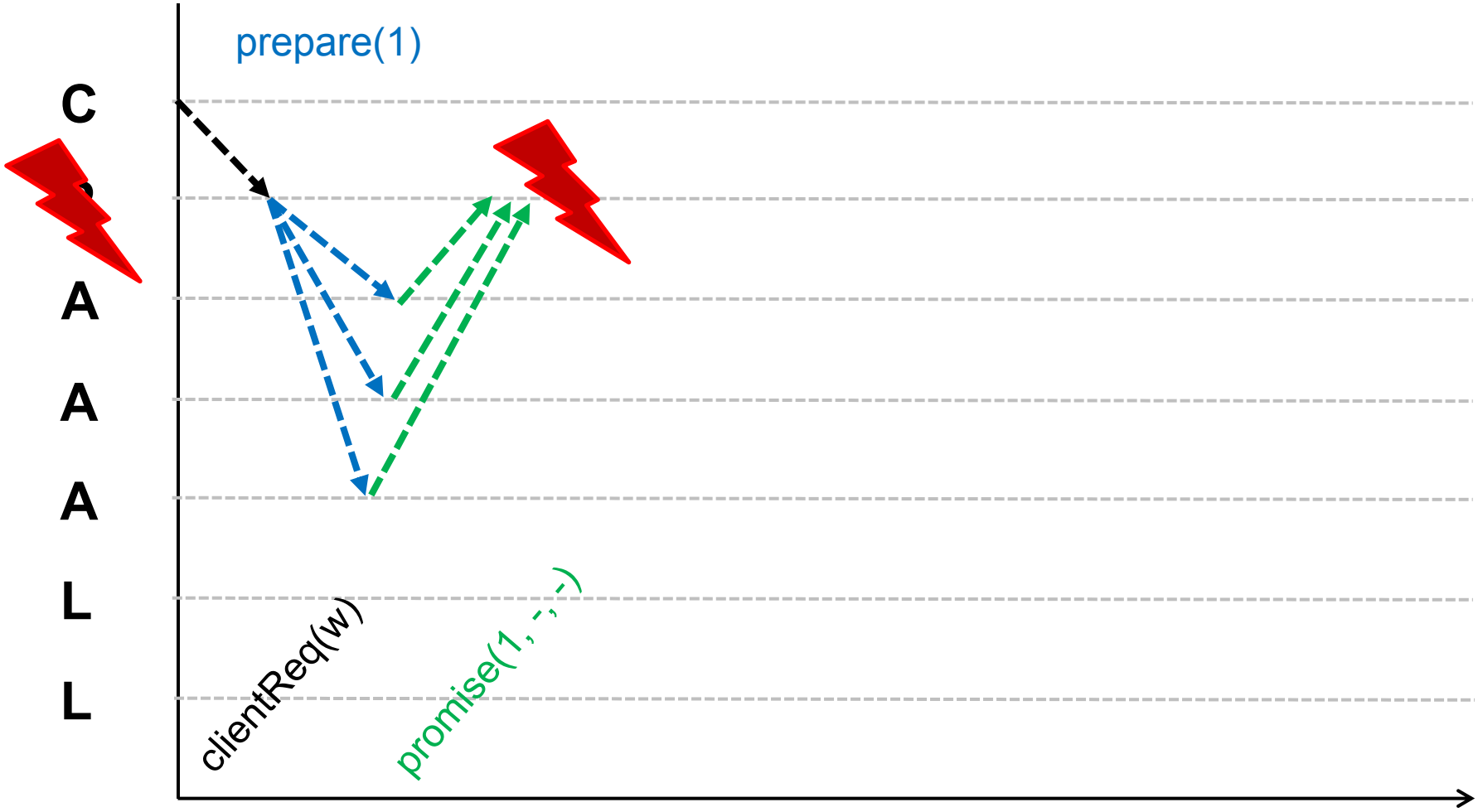
Failure of redundant learner II



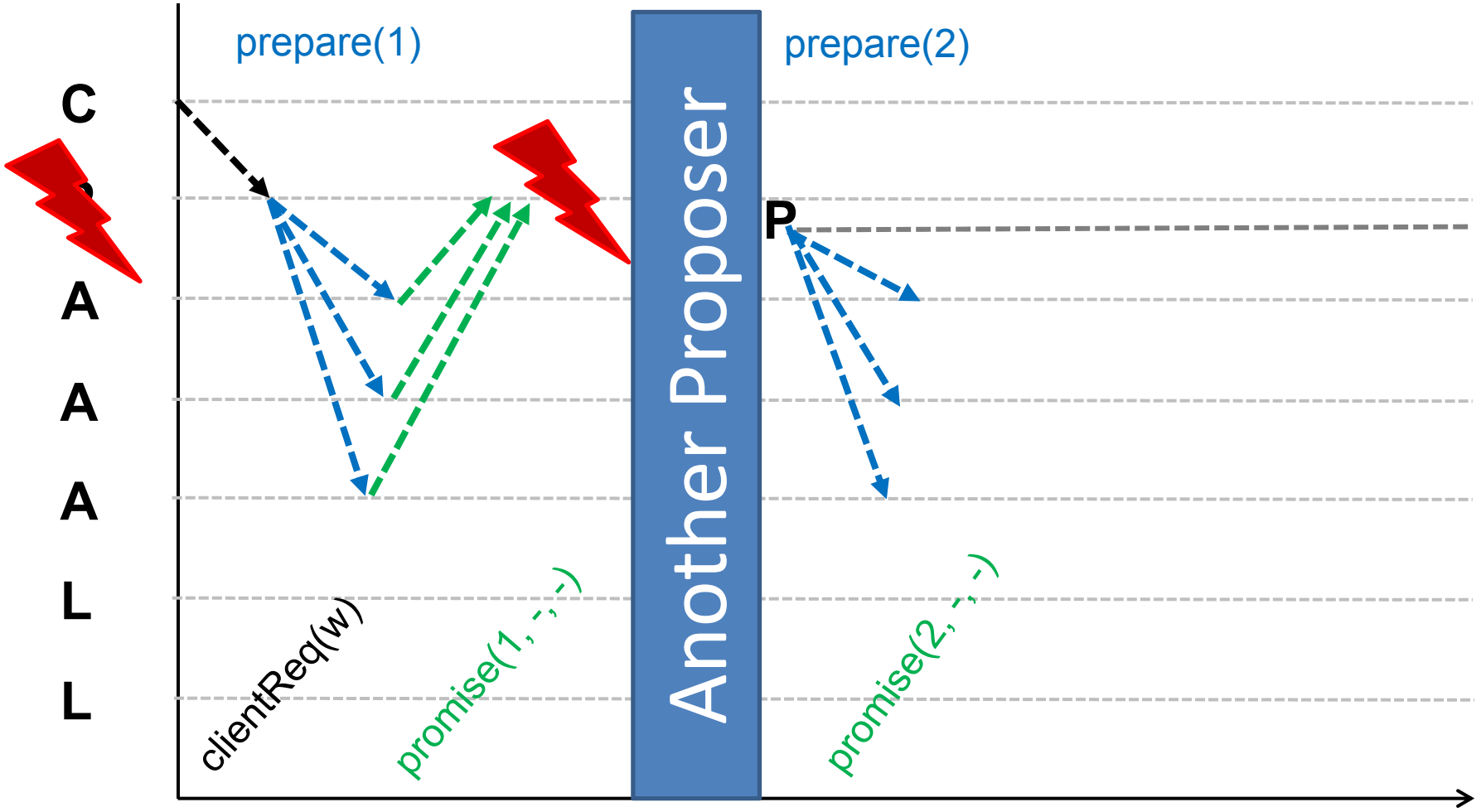
Basic Paxos: Failure of proposer

Basic Paxos: Failure of proposer I

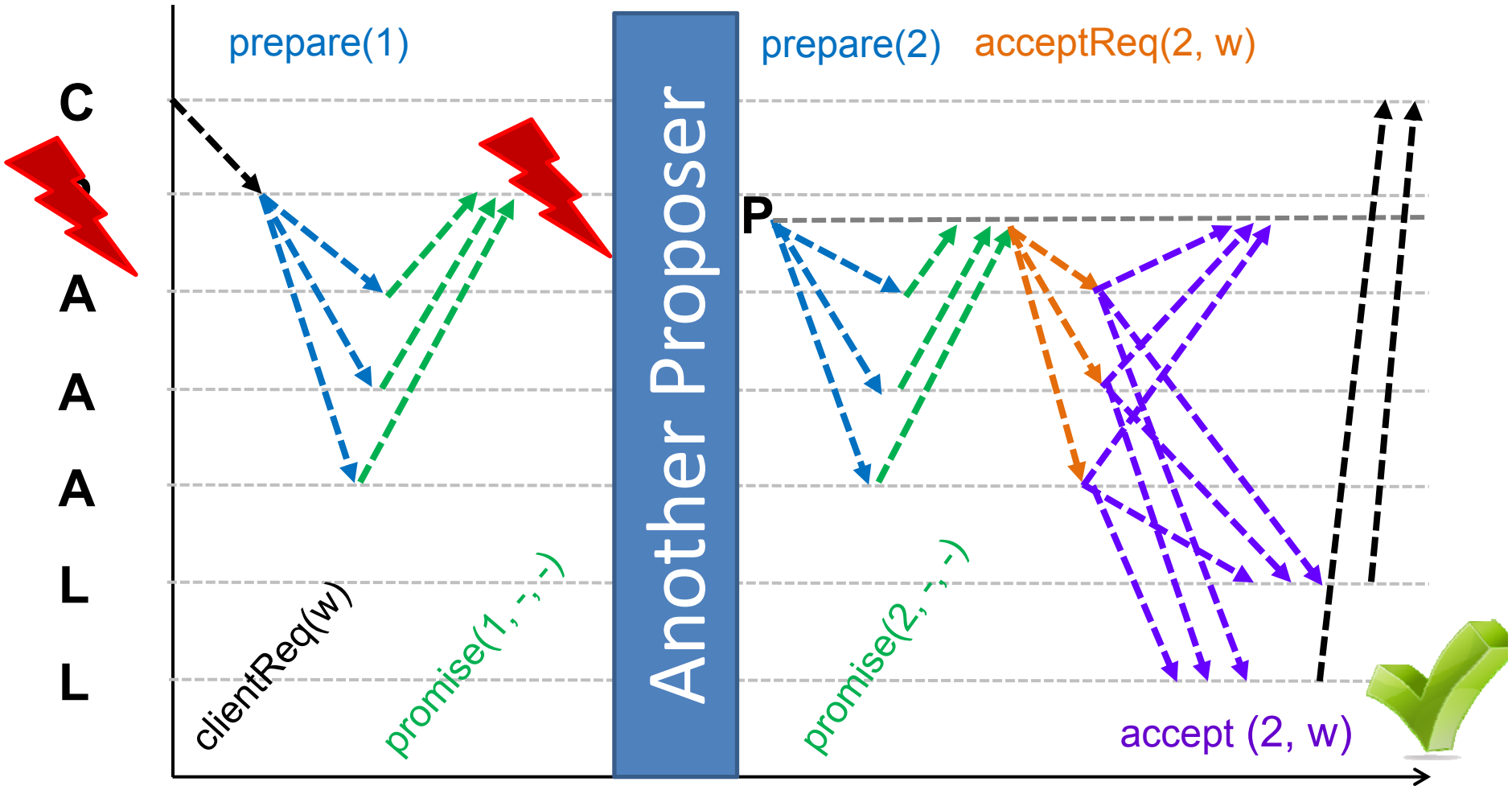
In prepare phase



Basic Paxos: Failure of proposer II



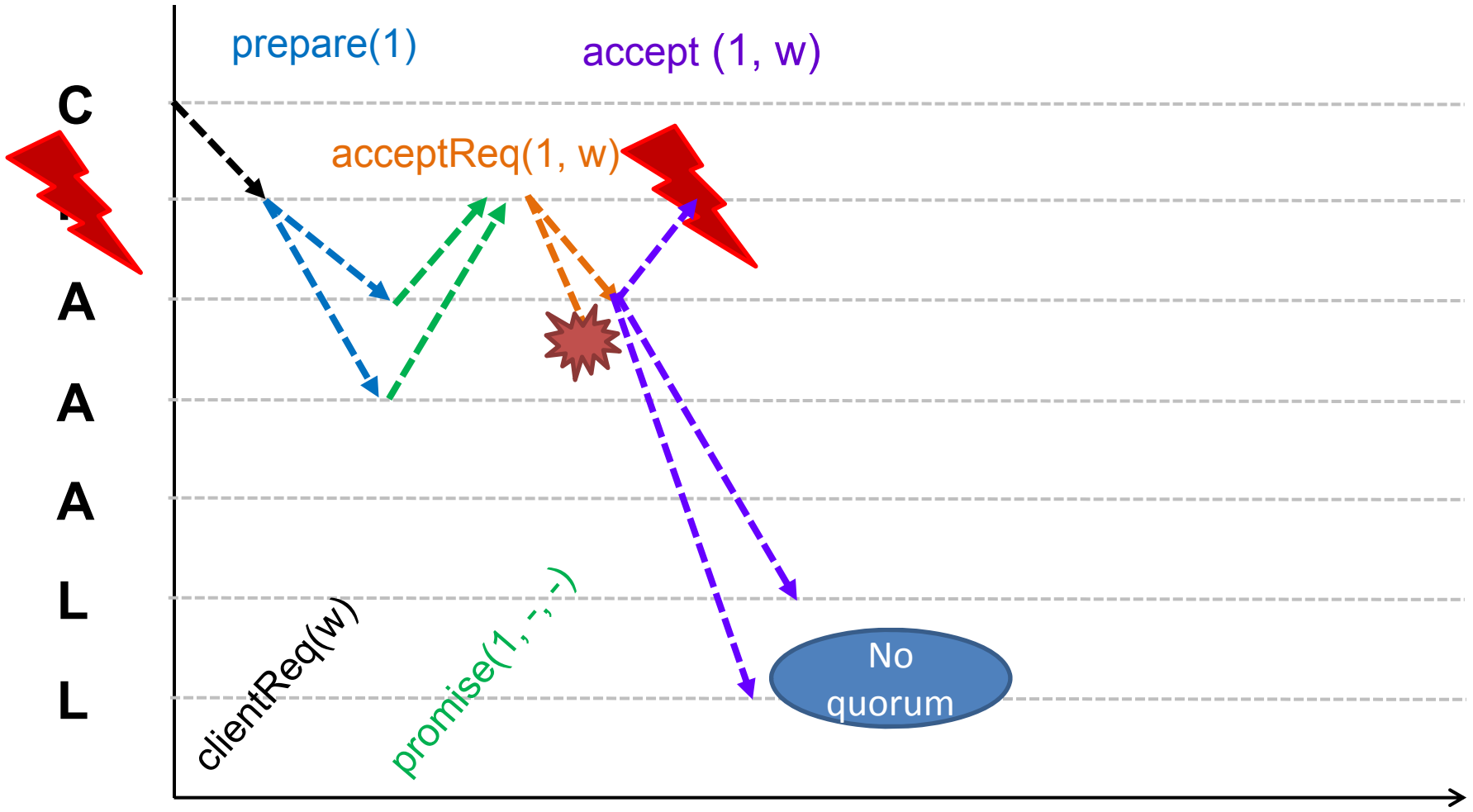
Basic Paxos: Failure of proposer III



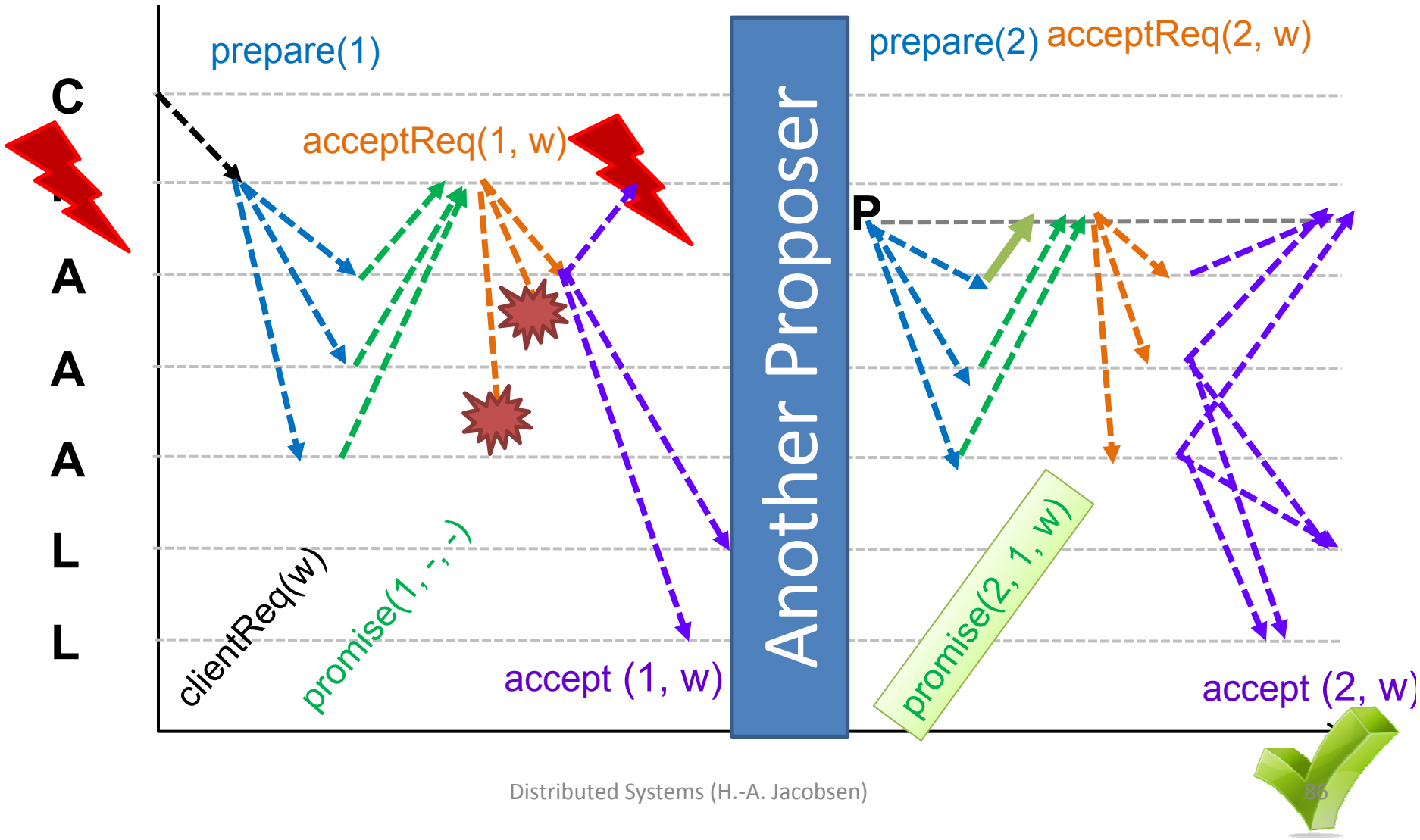
Basic Paxos: Failure between accepts

In accept phase

Basic Paxos: Failure in Phase 2



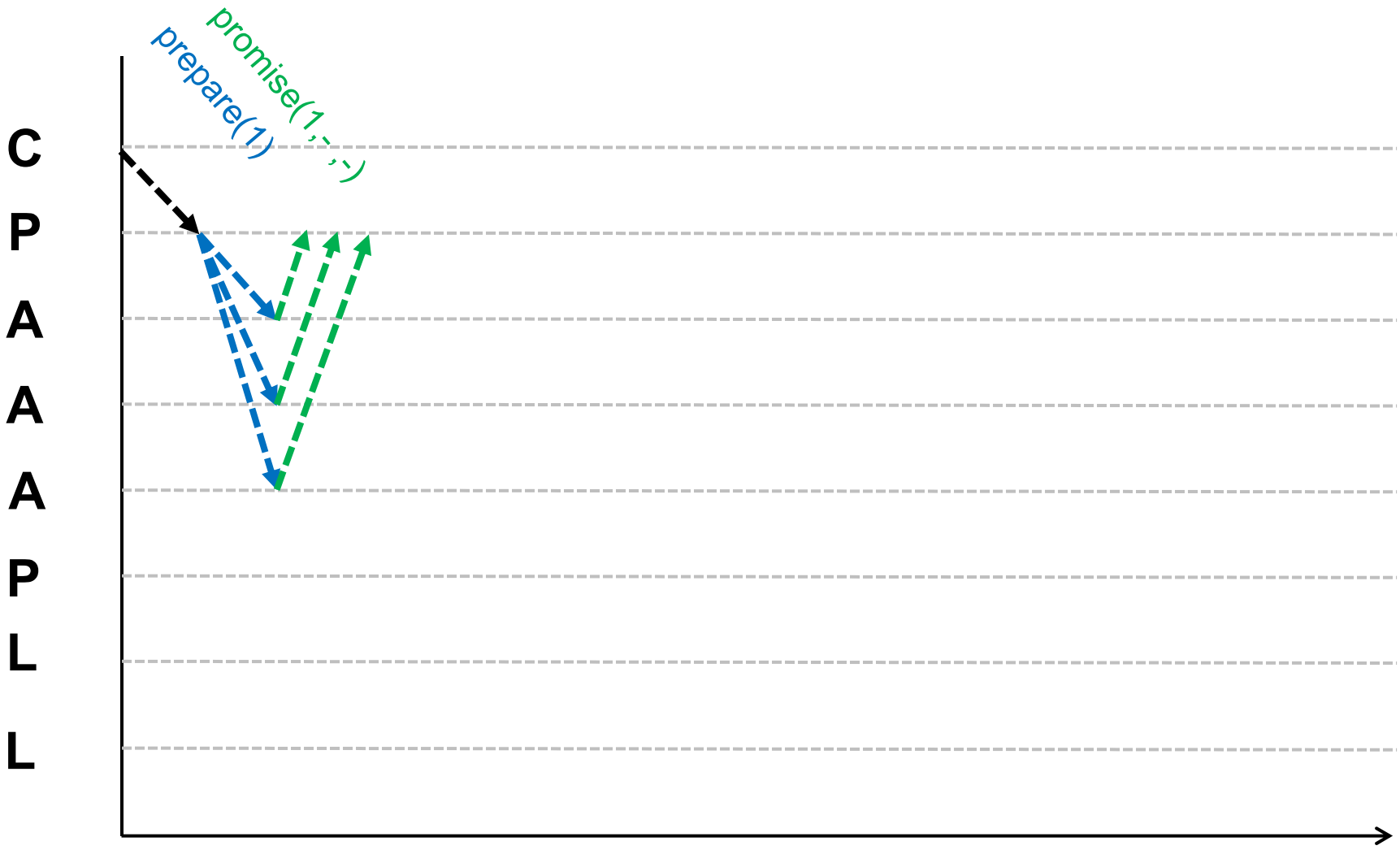
Basic Paxos: Failure in Phase 2



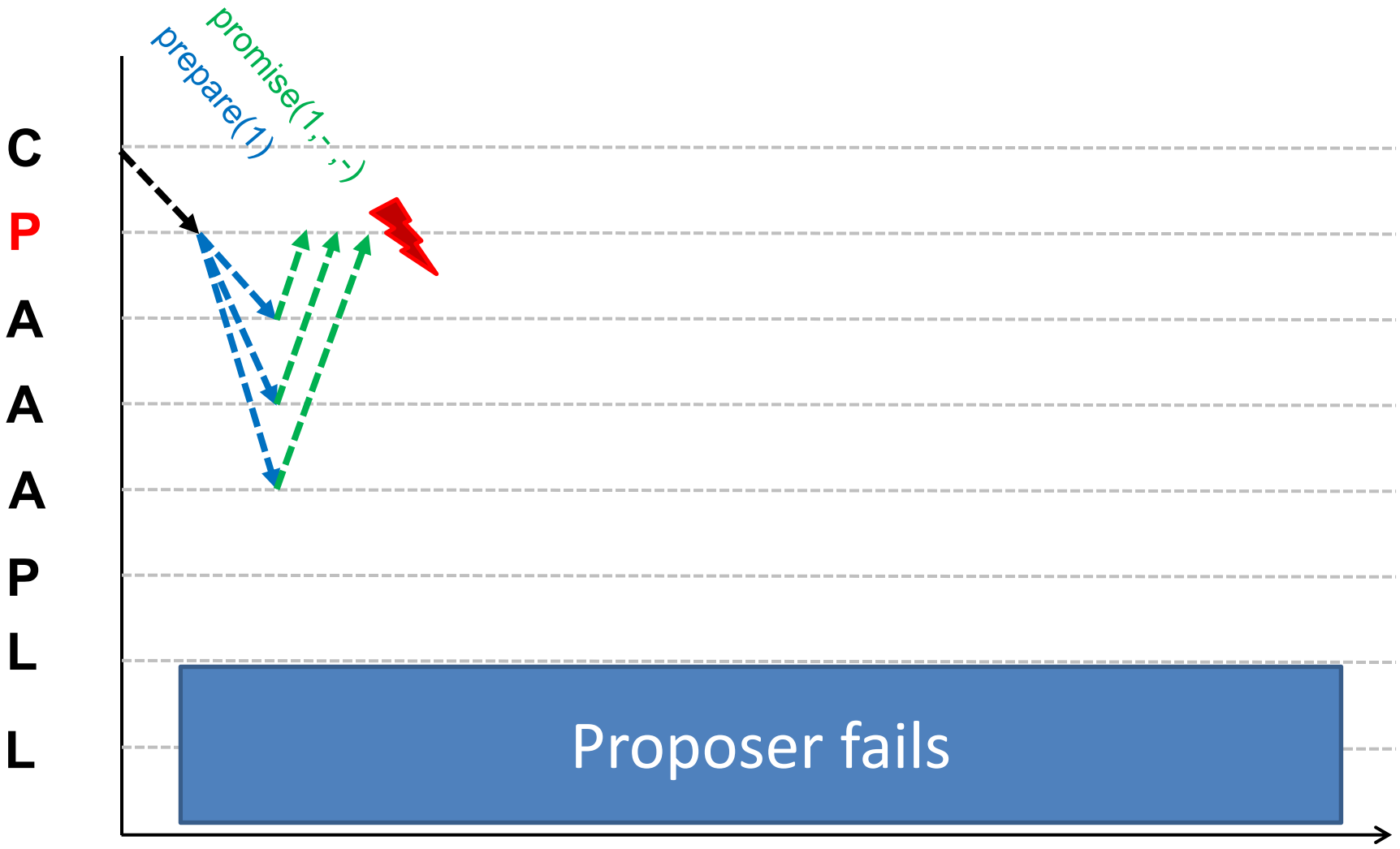
Basic Paxos: Dueling proposers

Dueling proposers

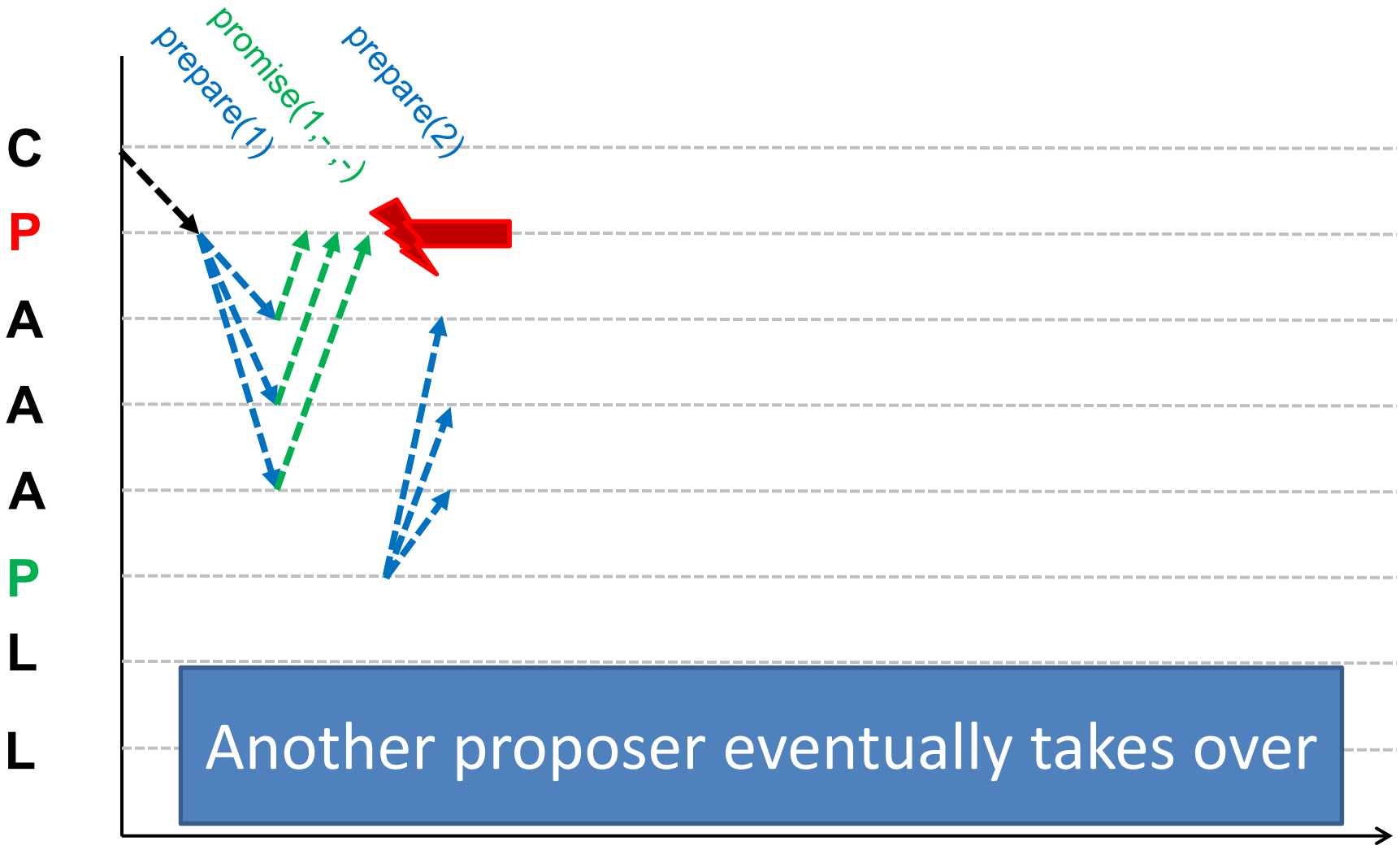
Dueling proposers



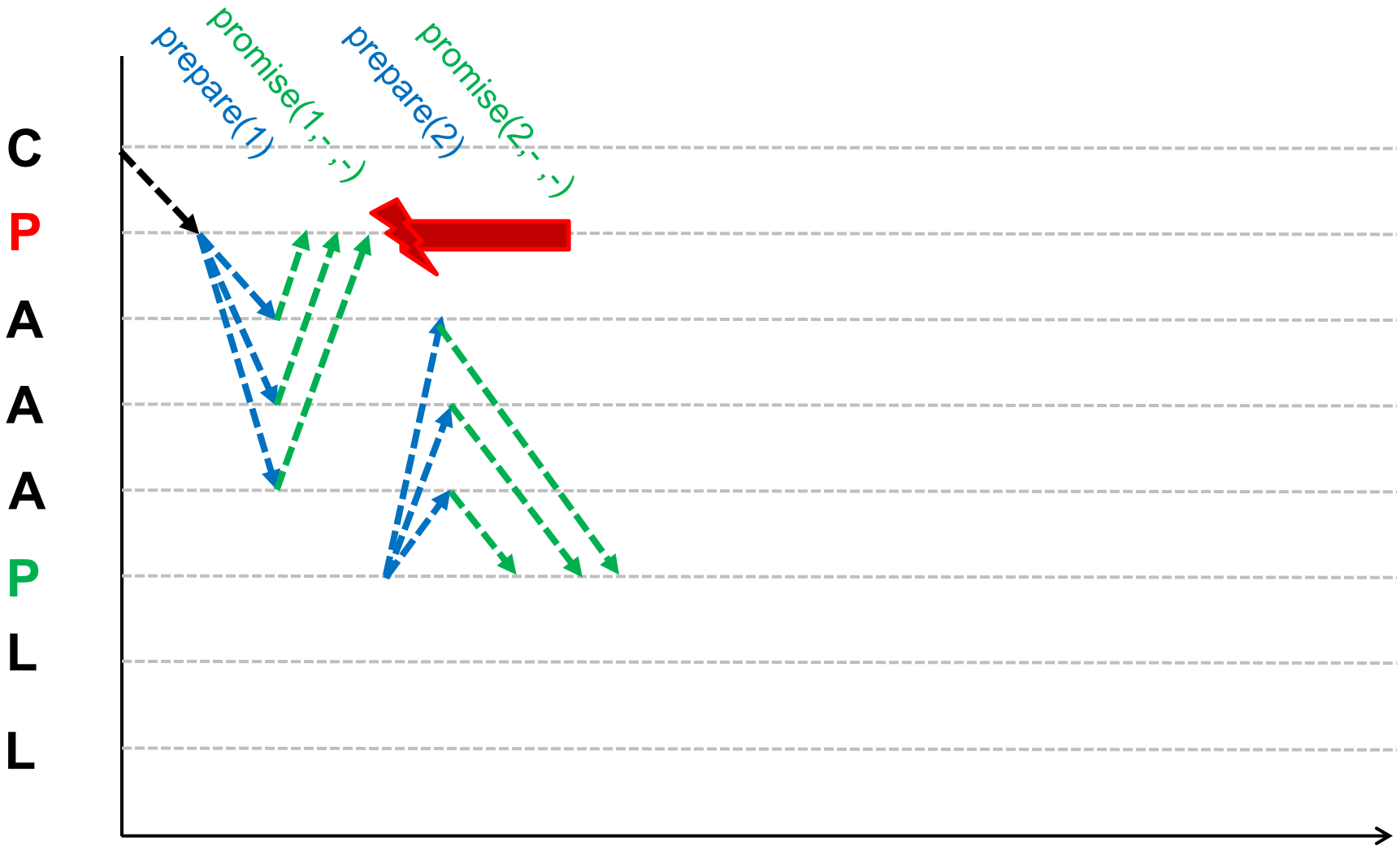
Dueling proposers



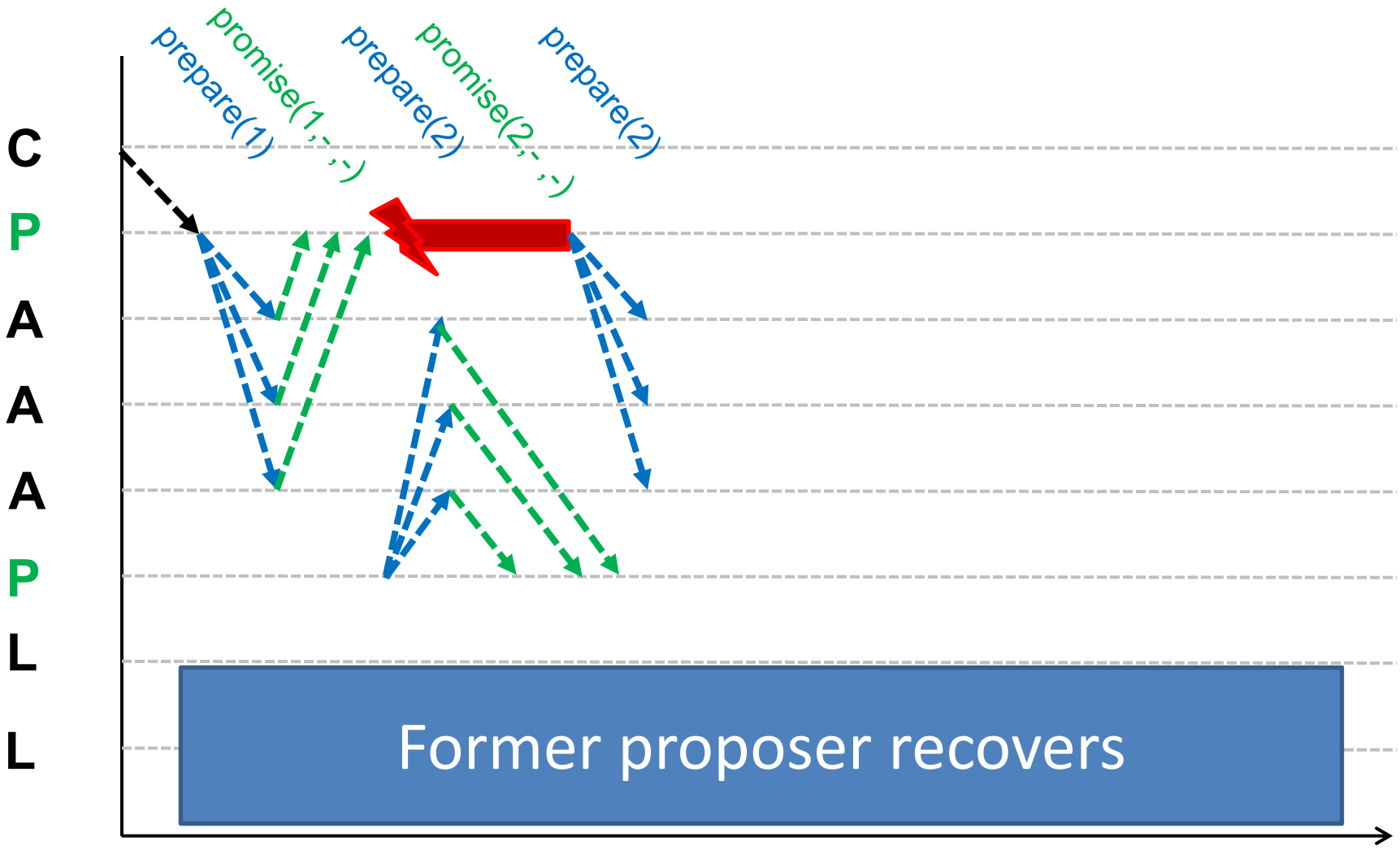
Dueling proposers



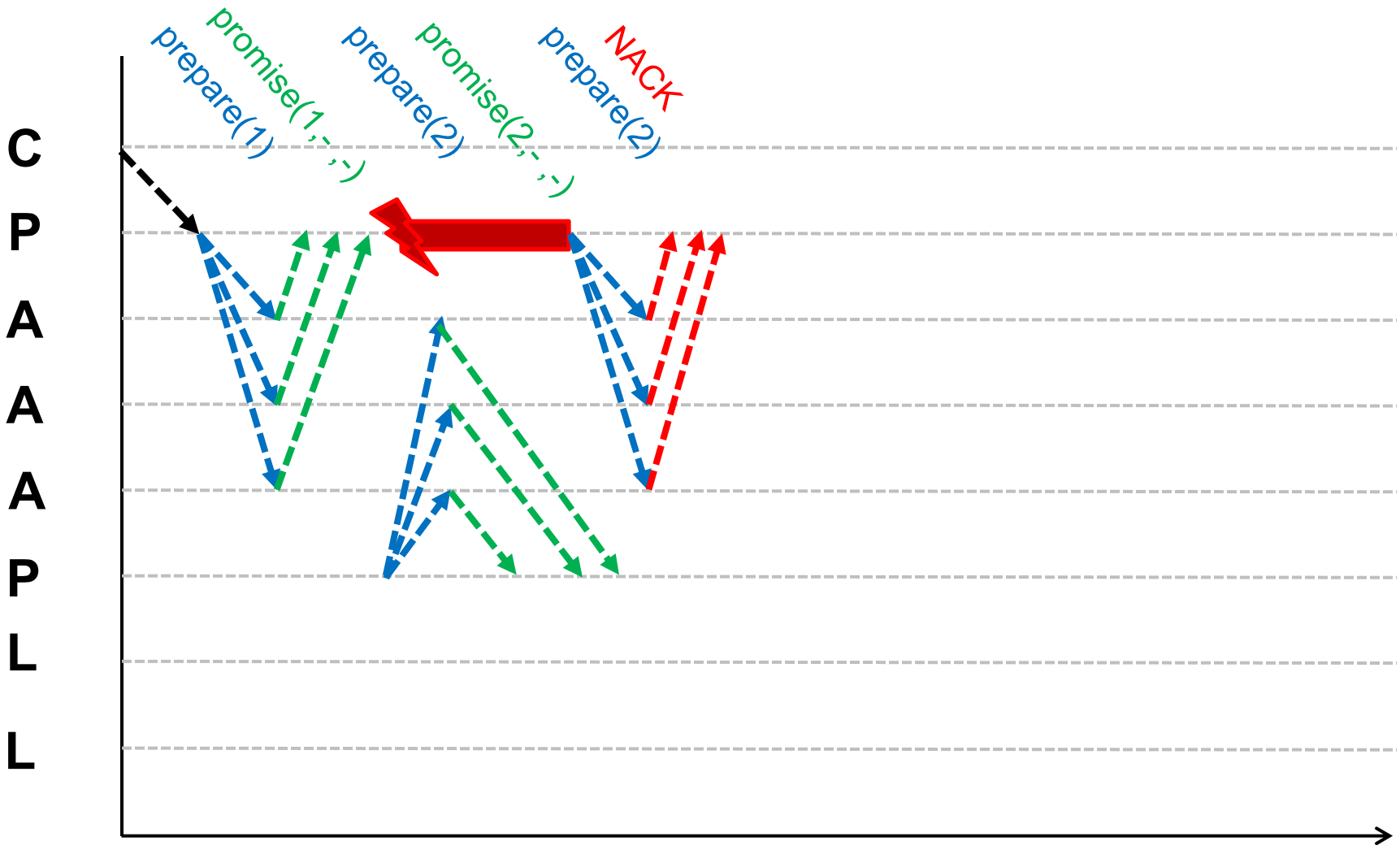
Dueling proposers



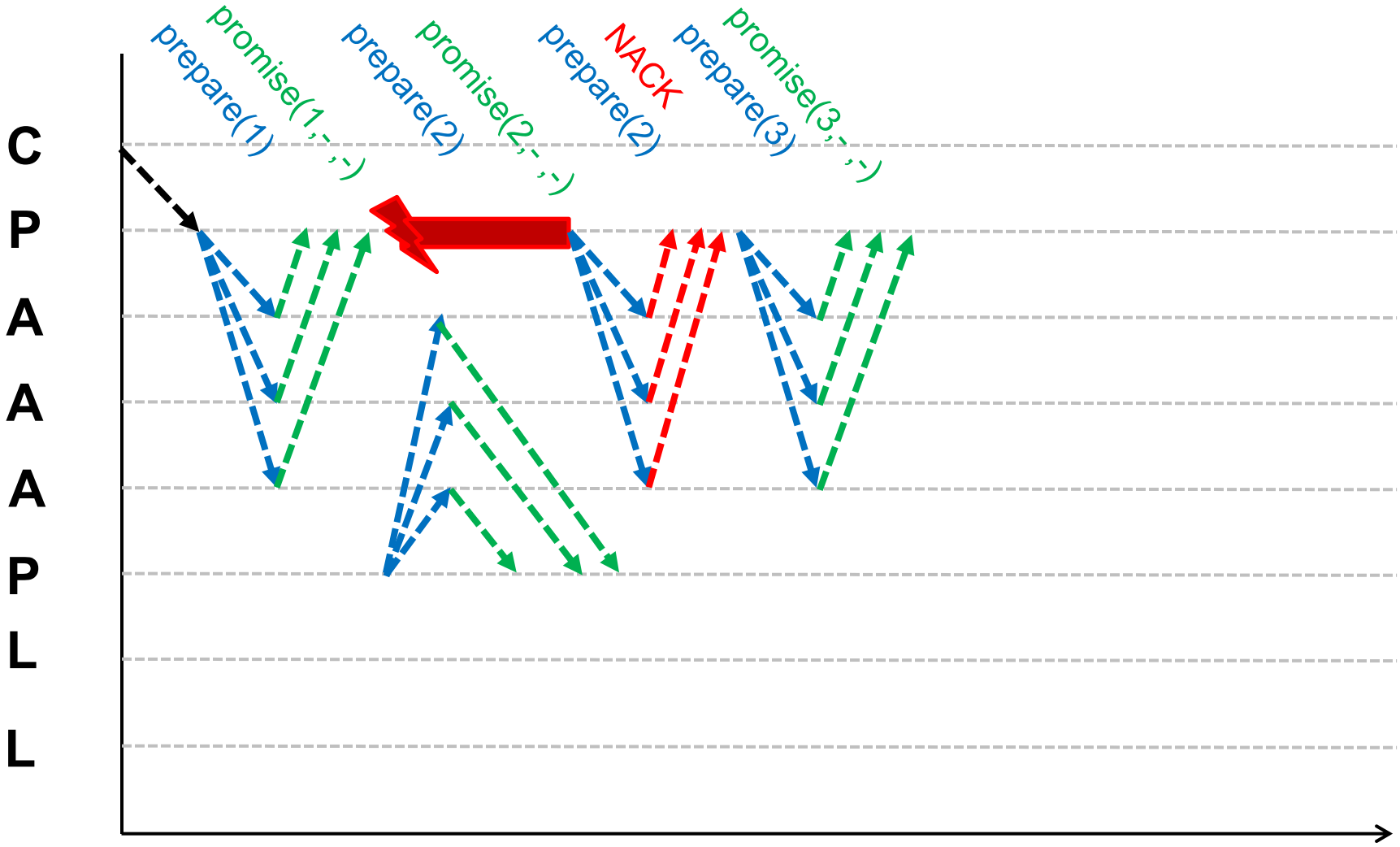
Dueling proposers



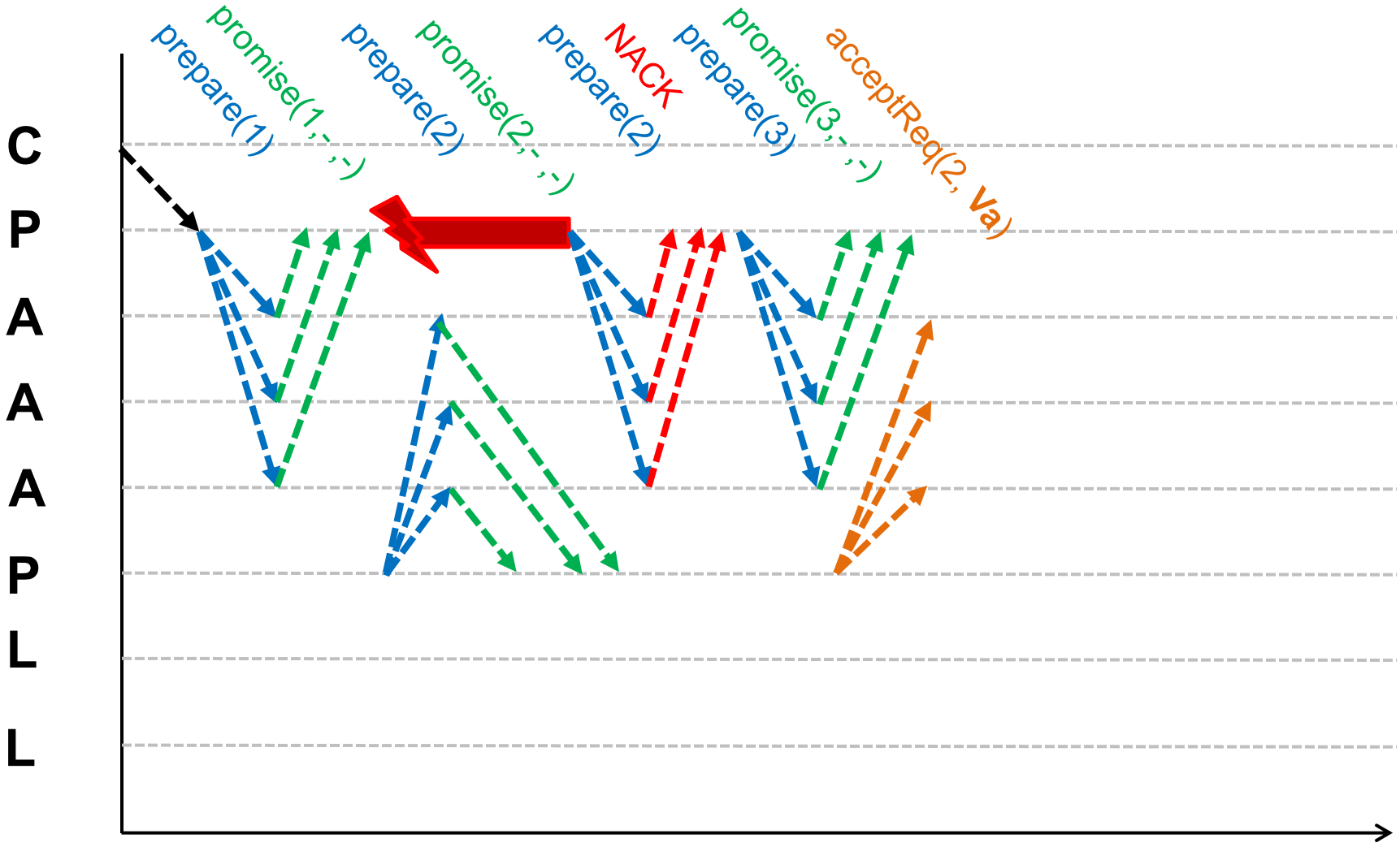
Dueling proposers



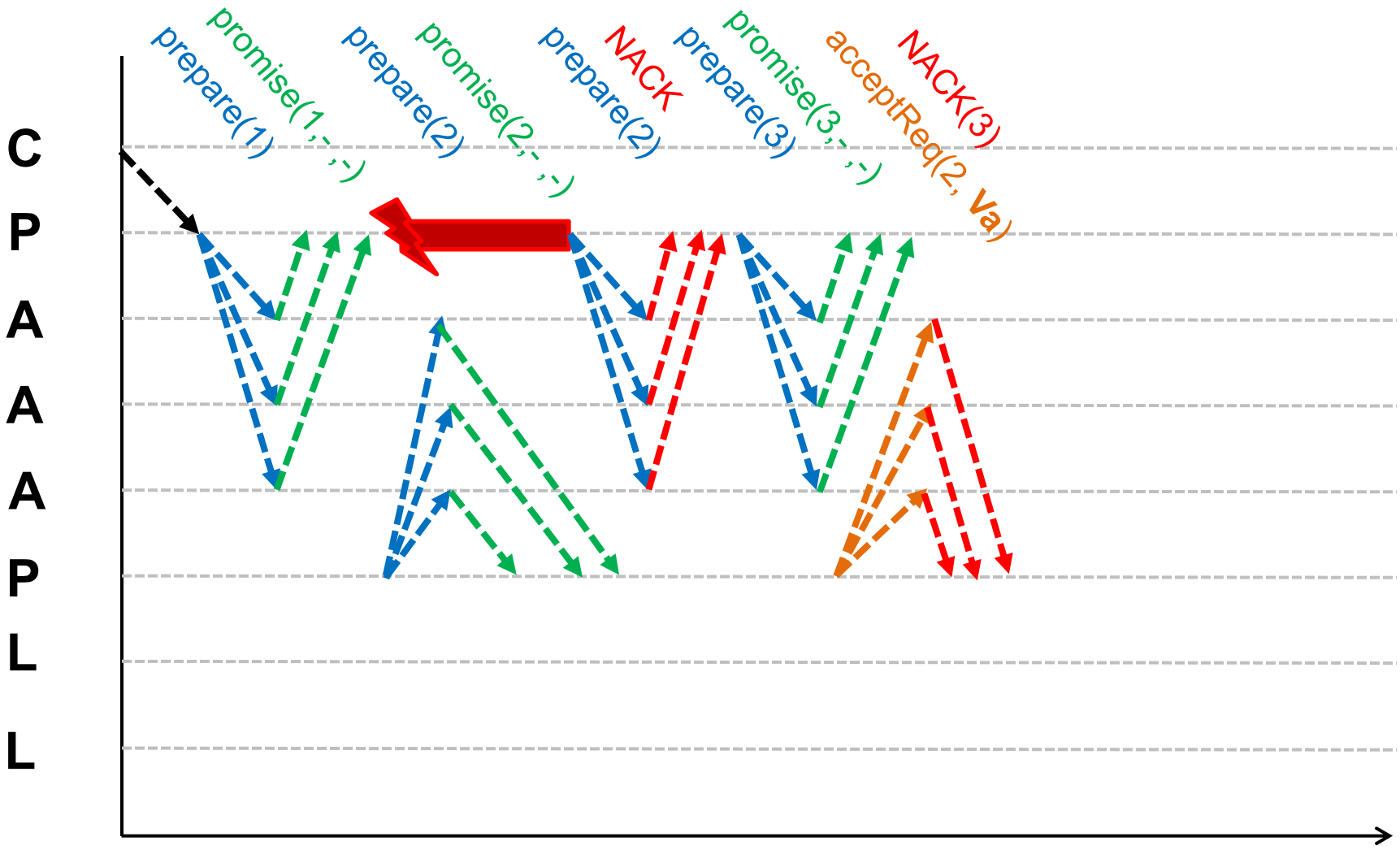
Dueling proposers



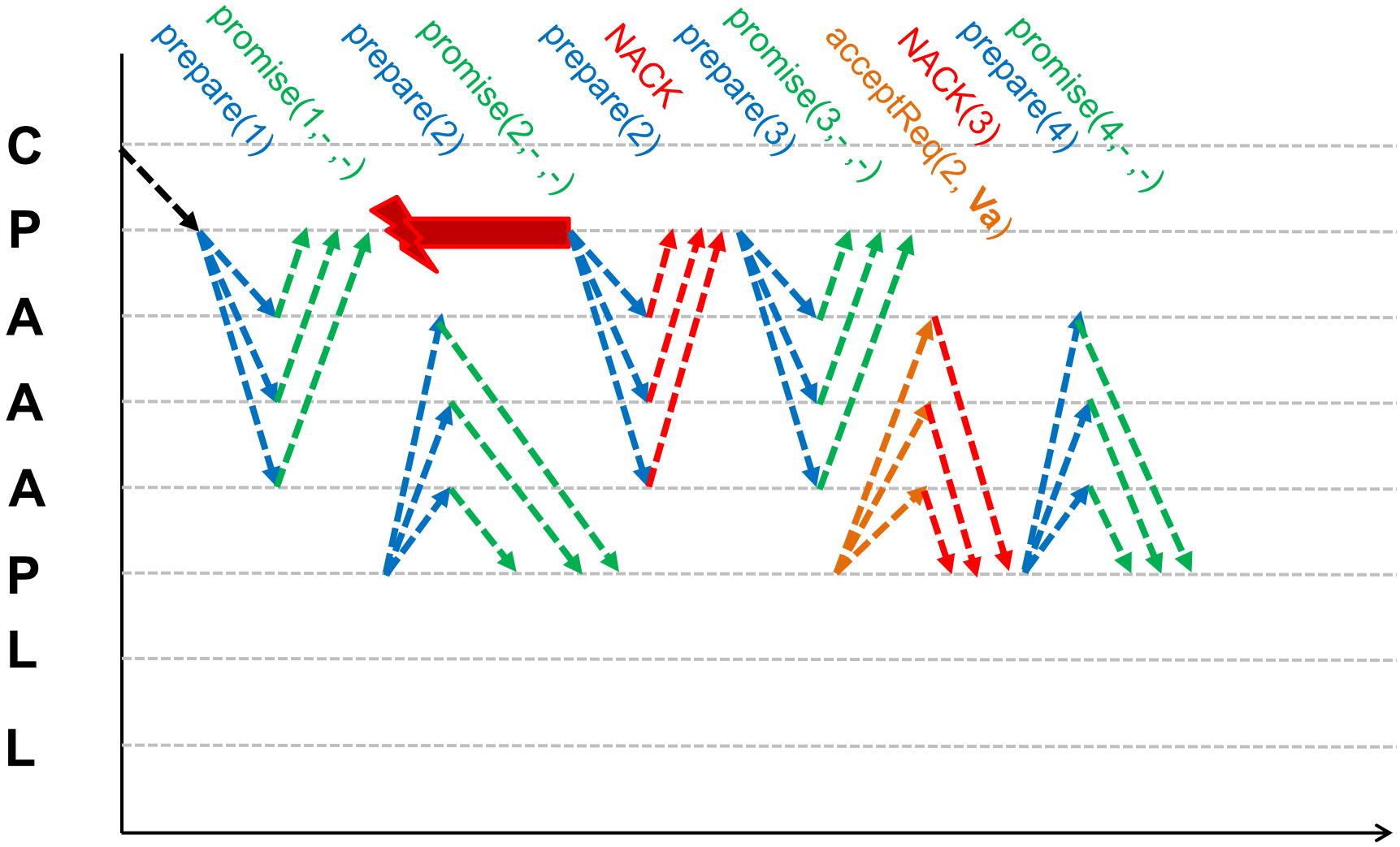
Dueling proposers



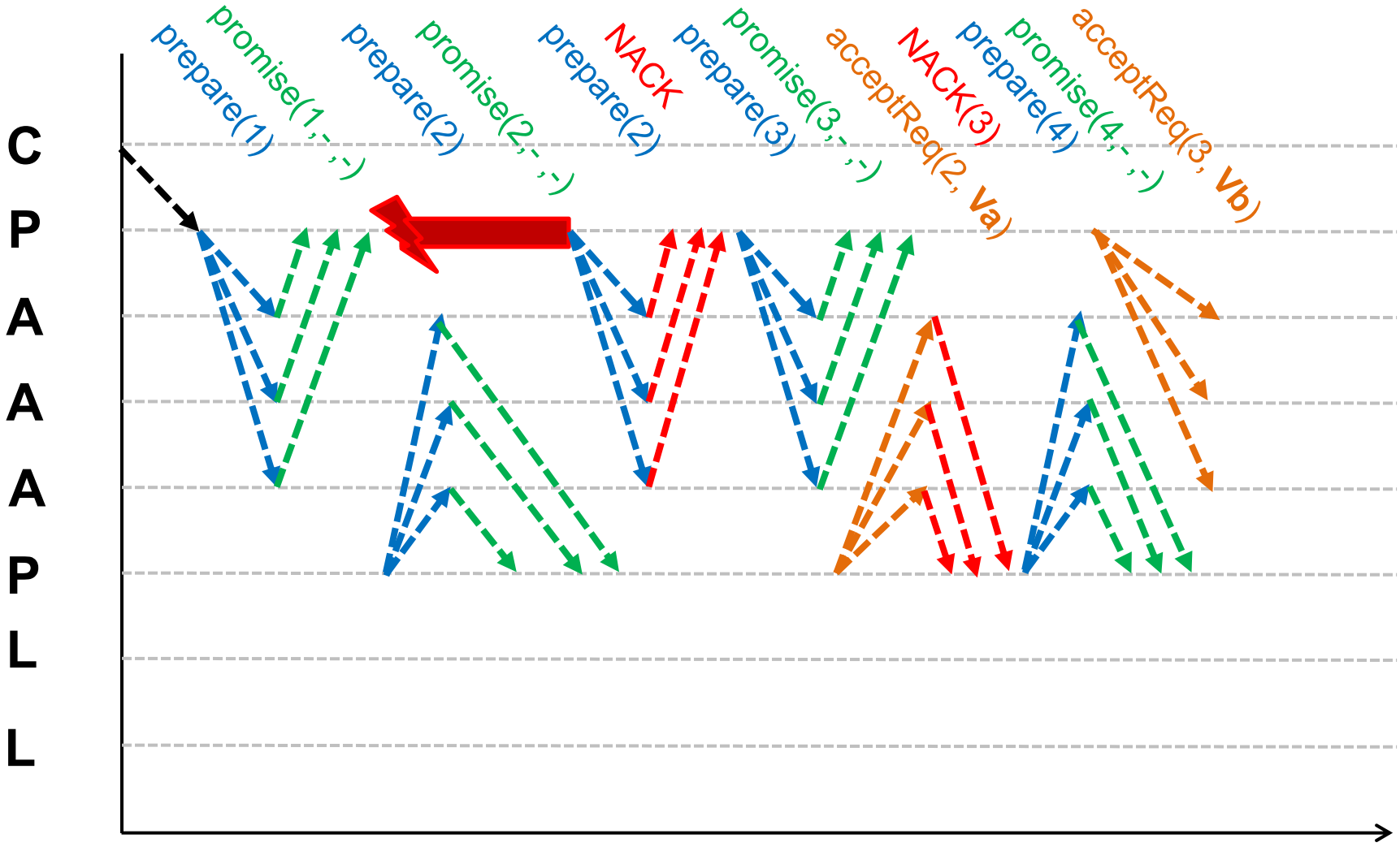
Dueling proposers



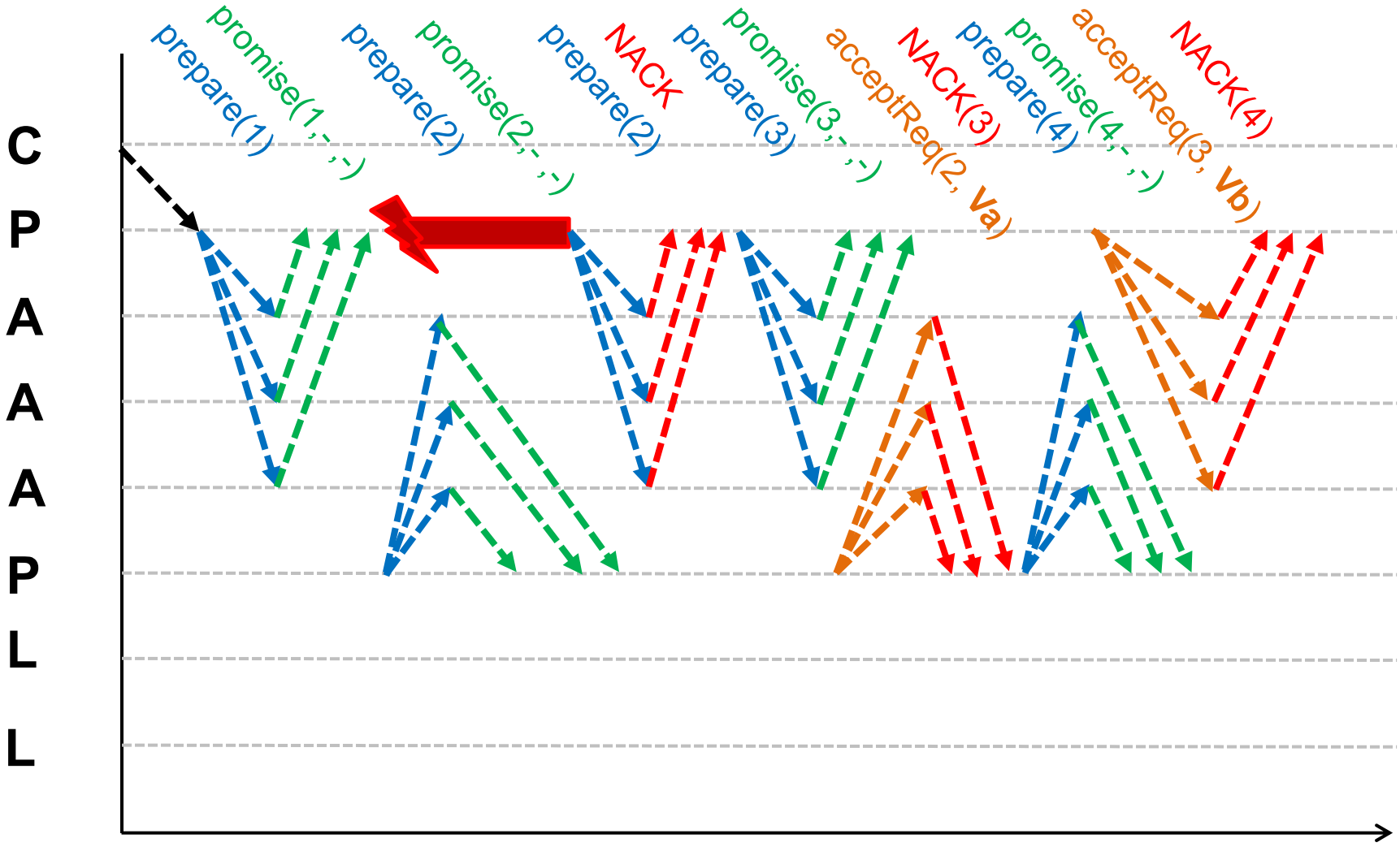
Dueling proposers



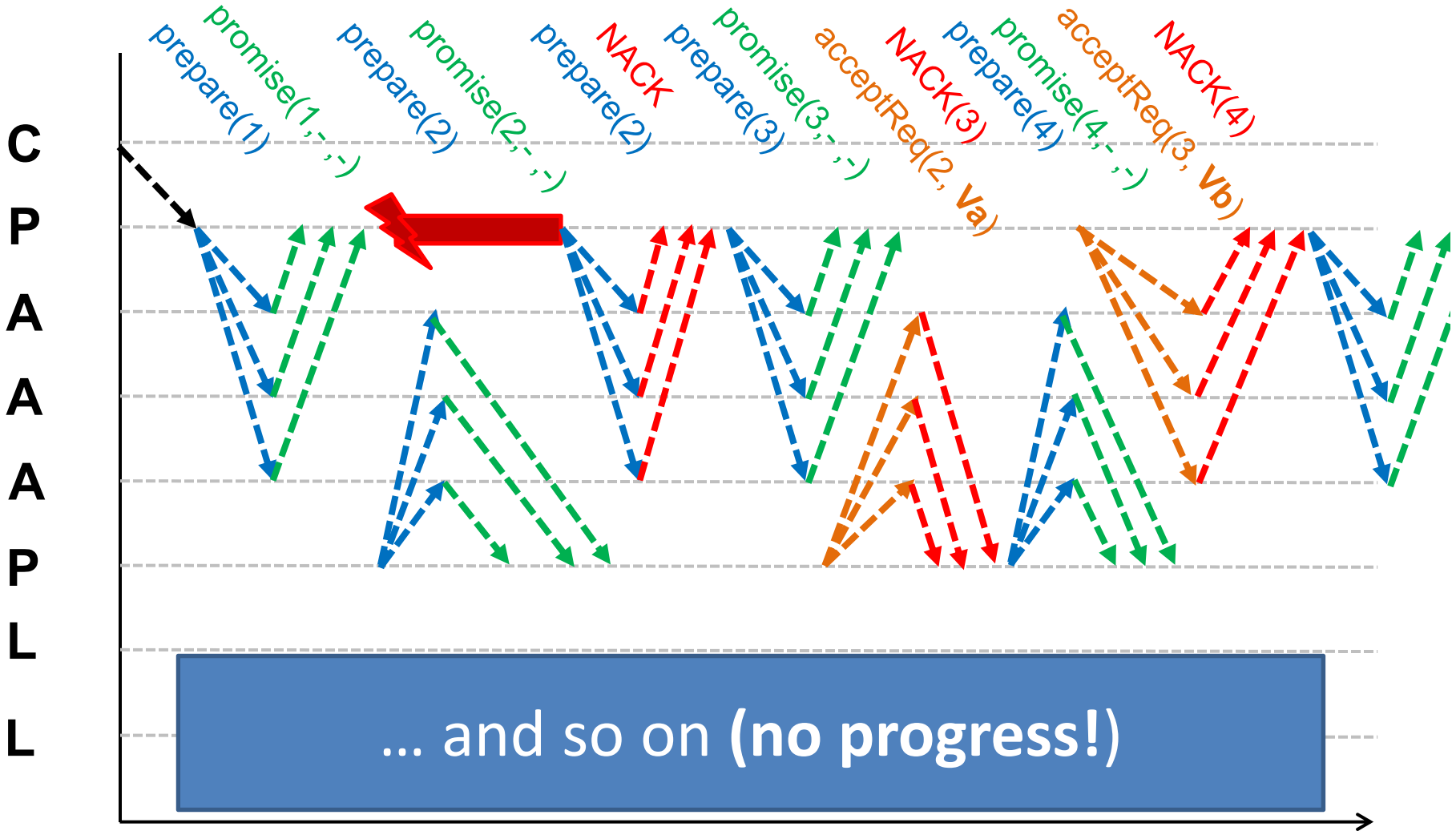
Dueling proposers



Dueling proposers

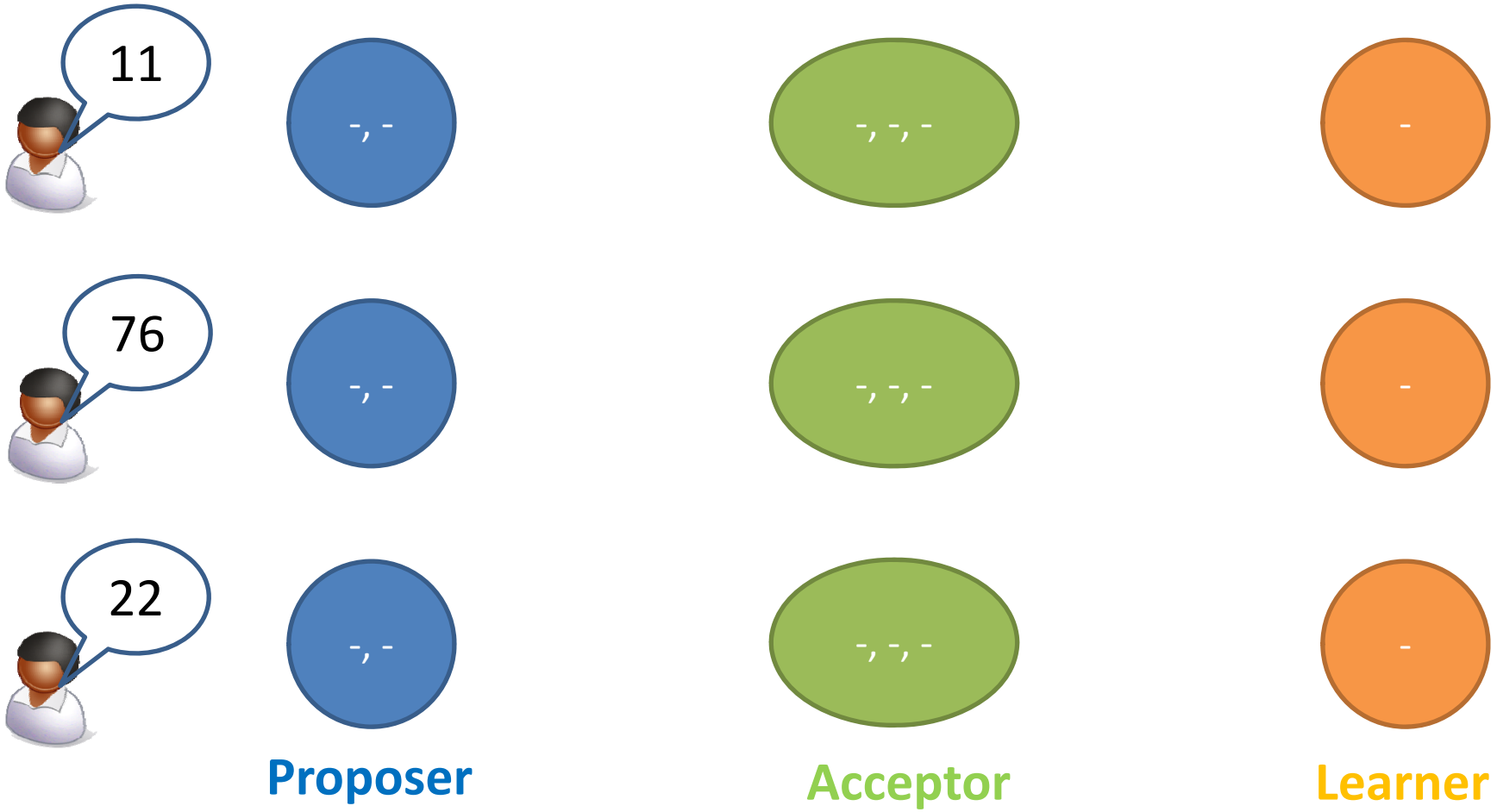


Dueling proposers



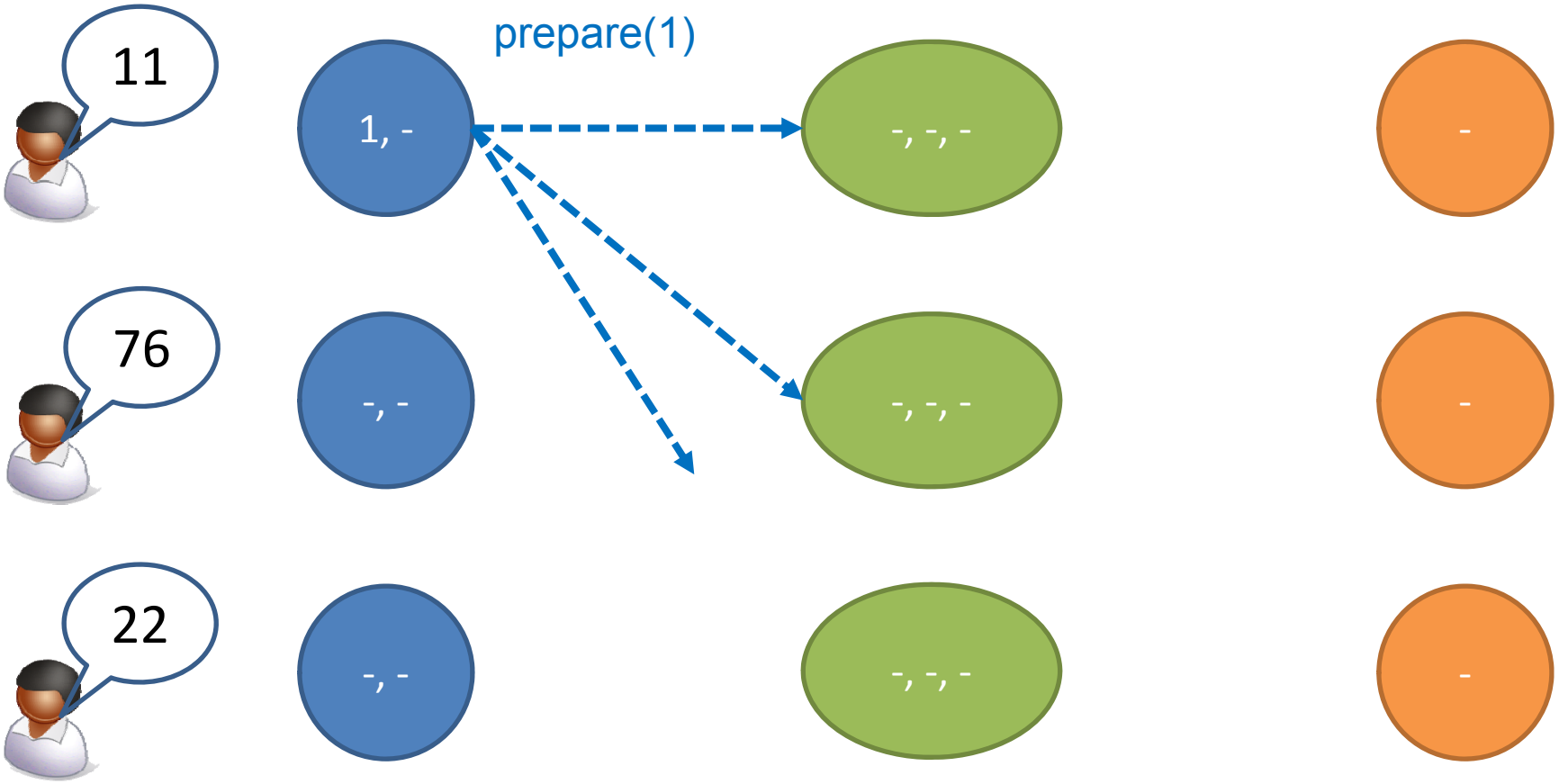
Basic Paxos: Communication failures

Basic Paxos with comm. failures

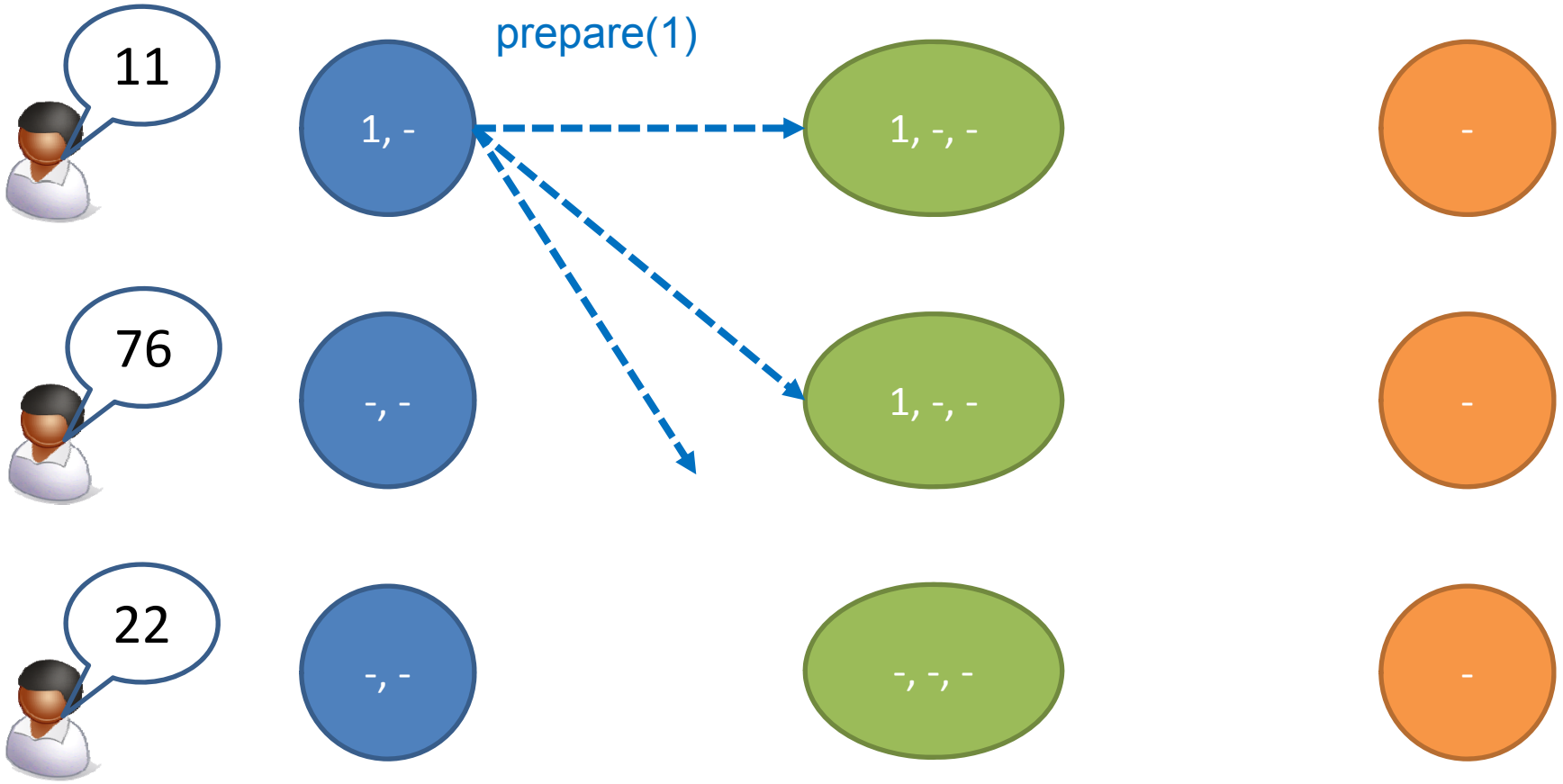


Quorum size is 2

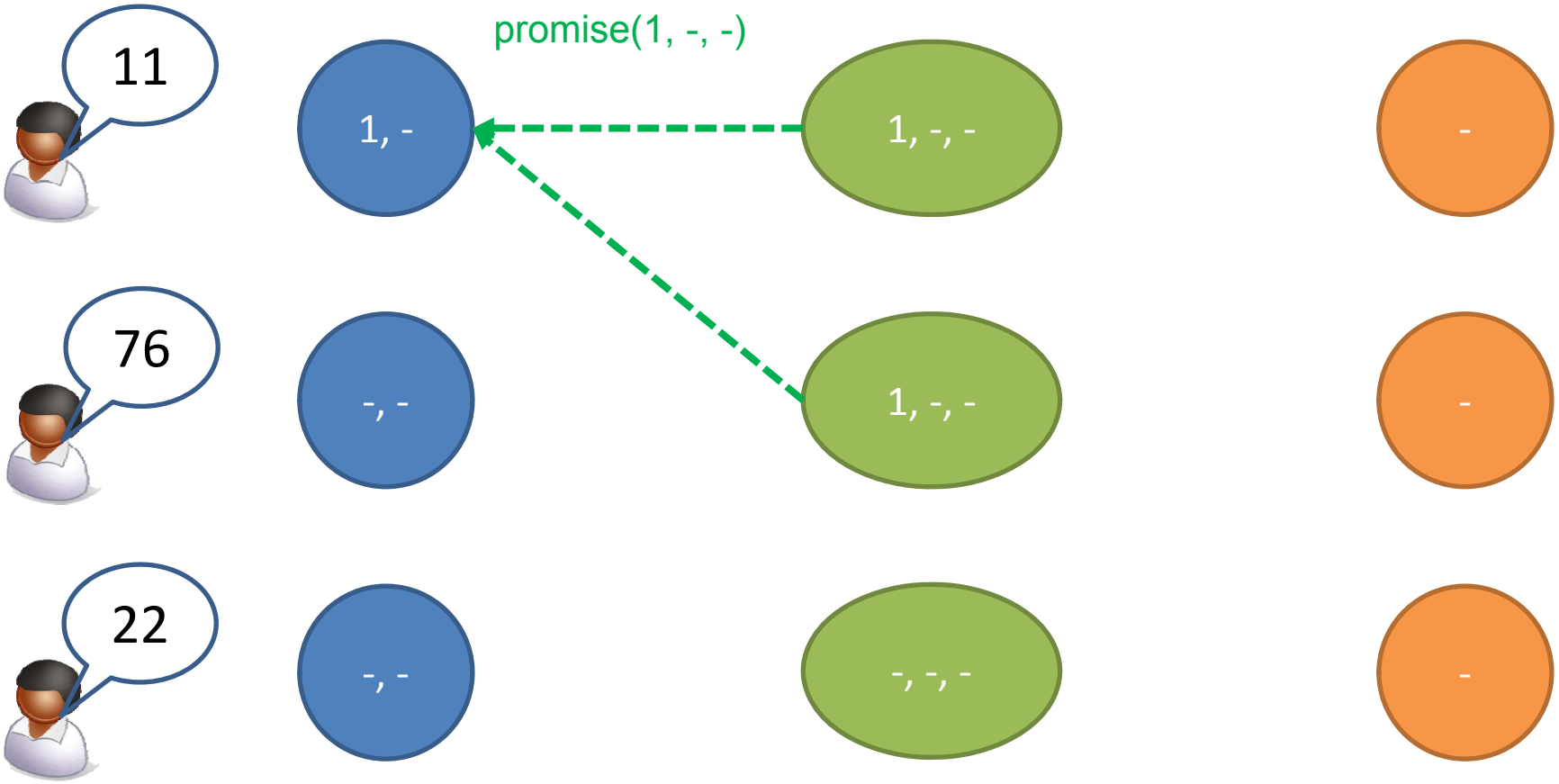
Basic Paxos with comm. failures



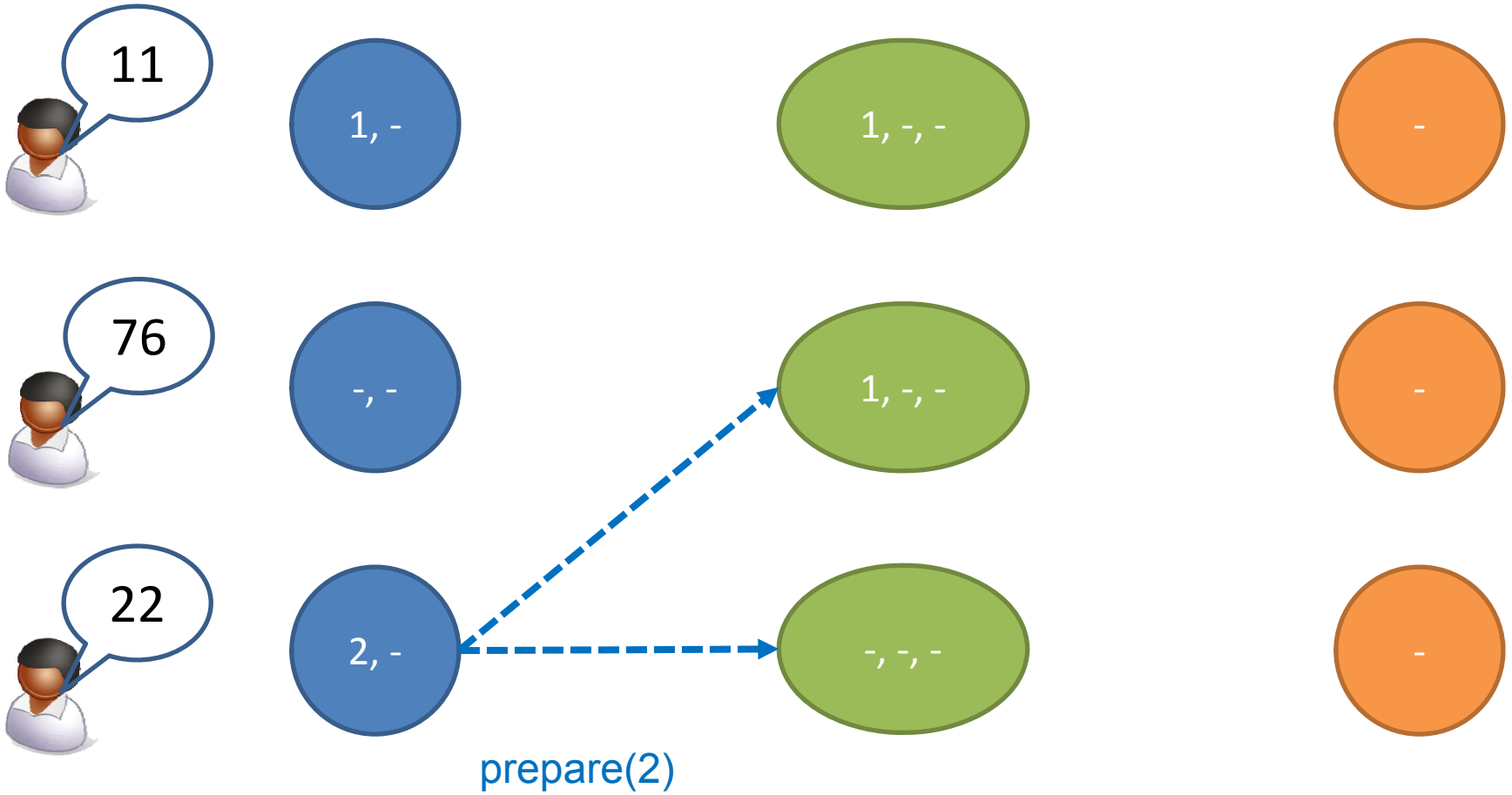
Basic Paxos with comm. failures



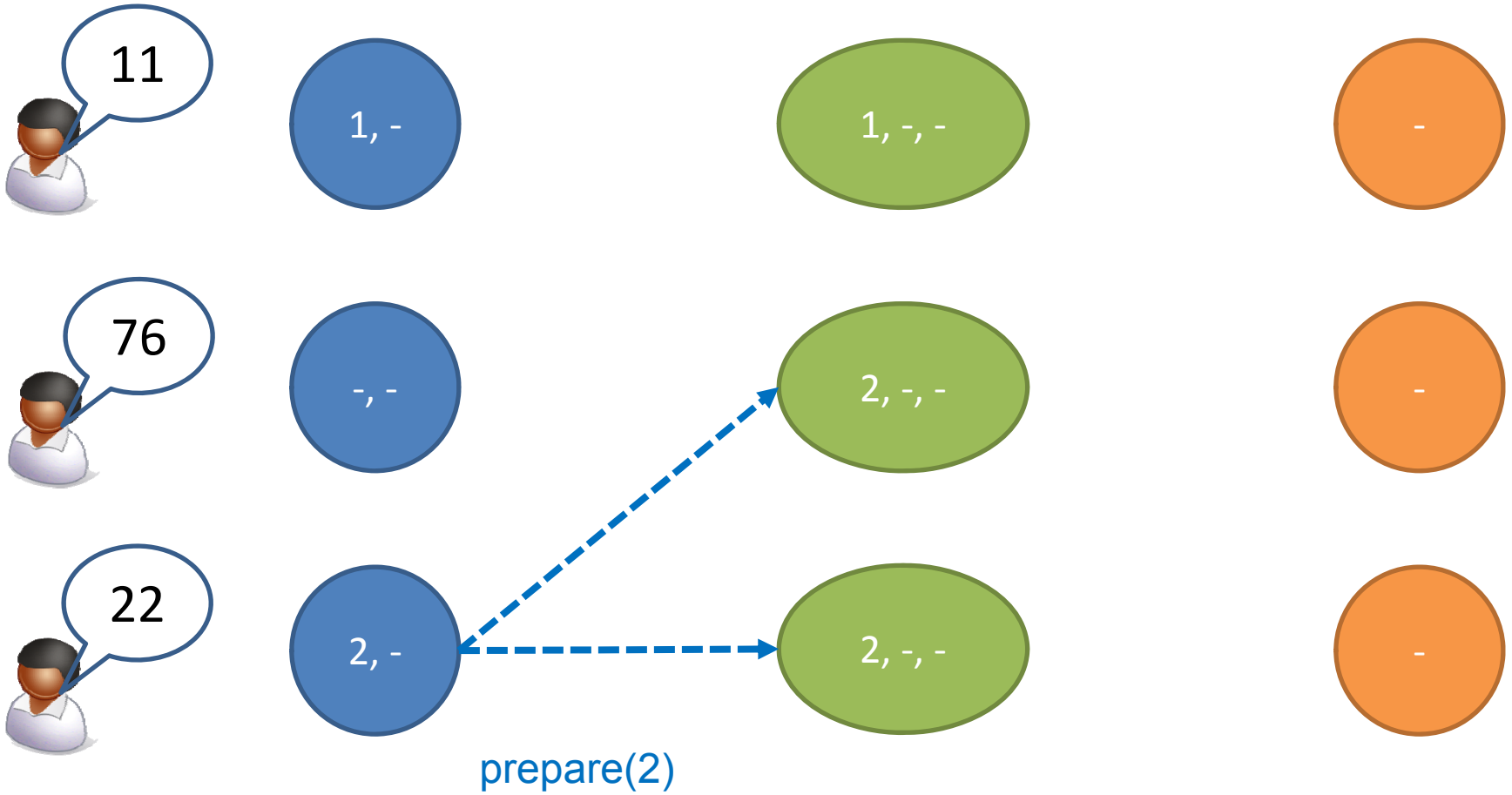
Basic Paxos with comm. failures



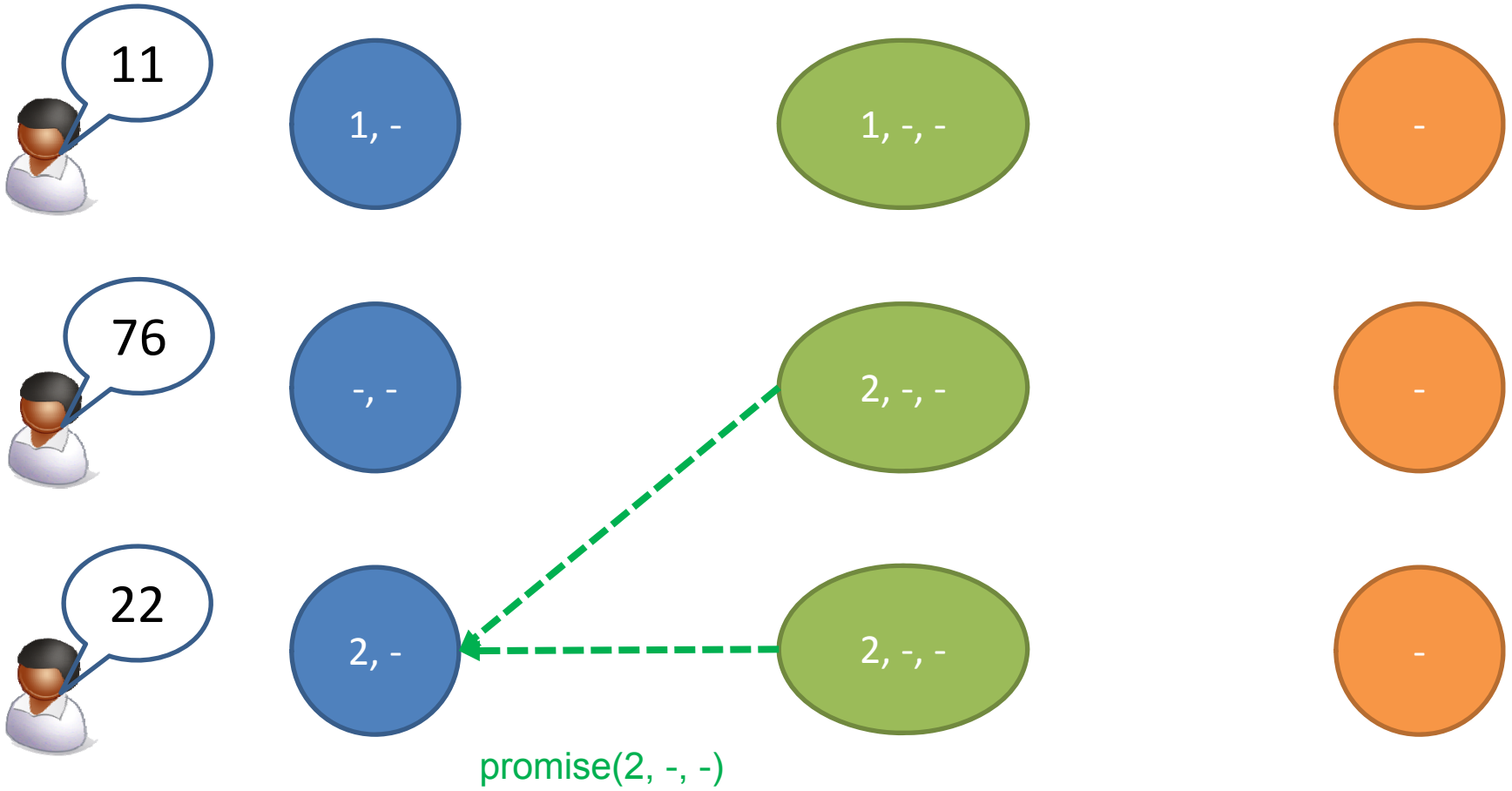
Basic Paxos with comm. failures



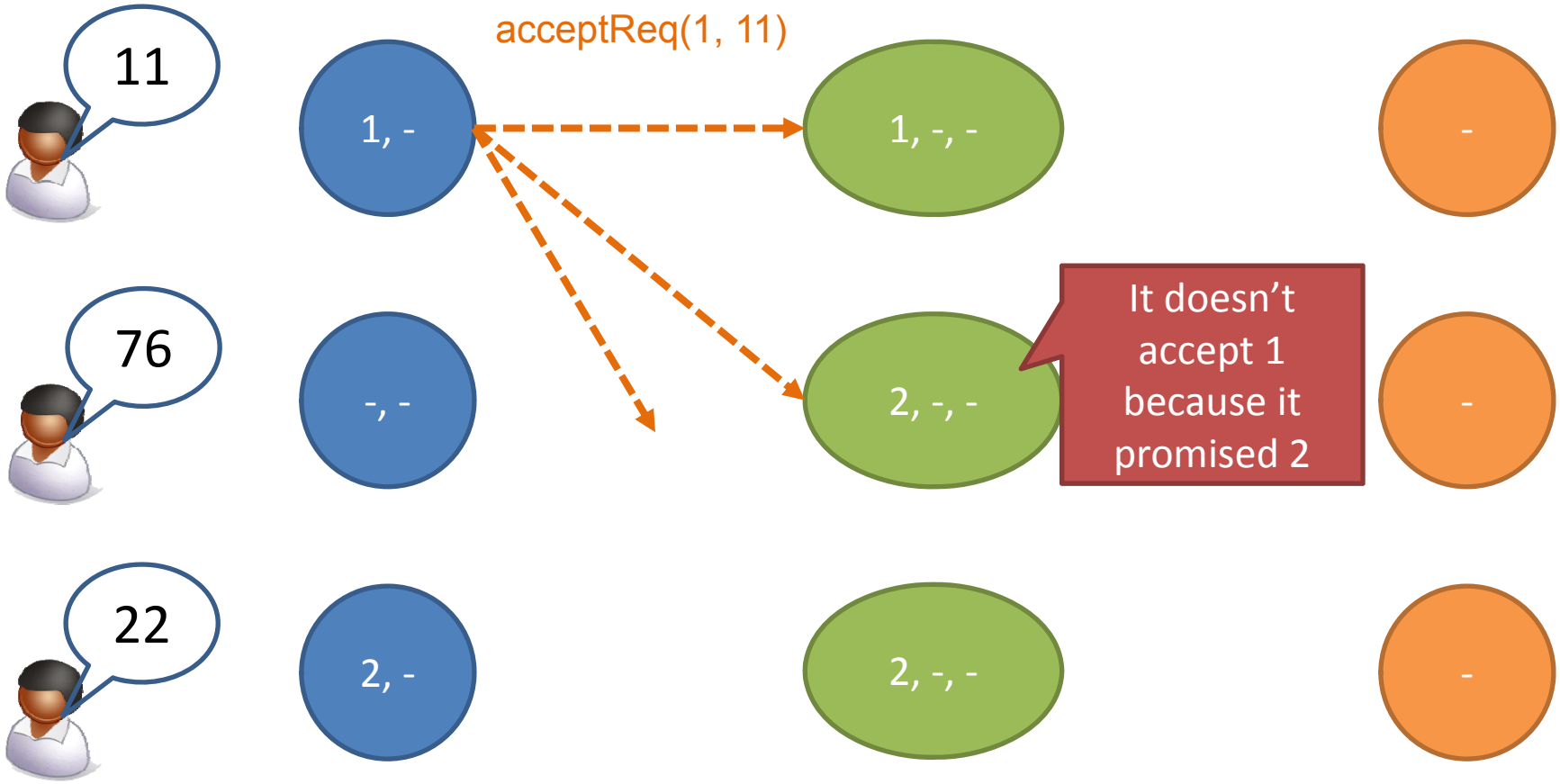
Basic Paxos with comm. failures



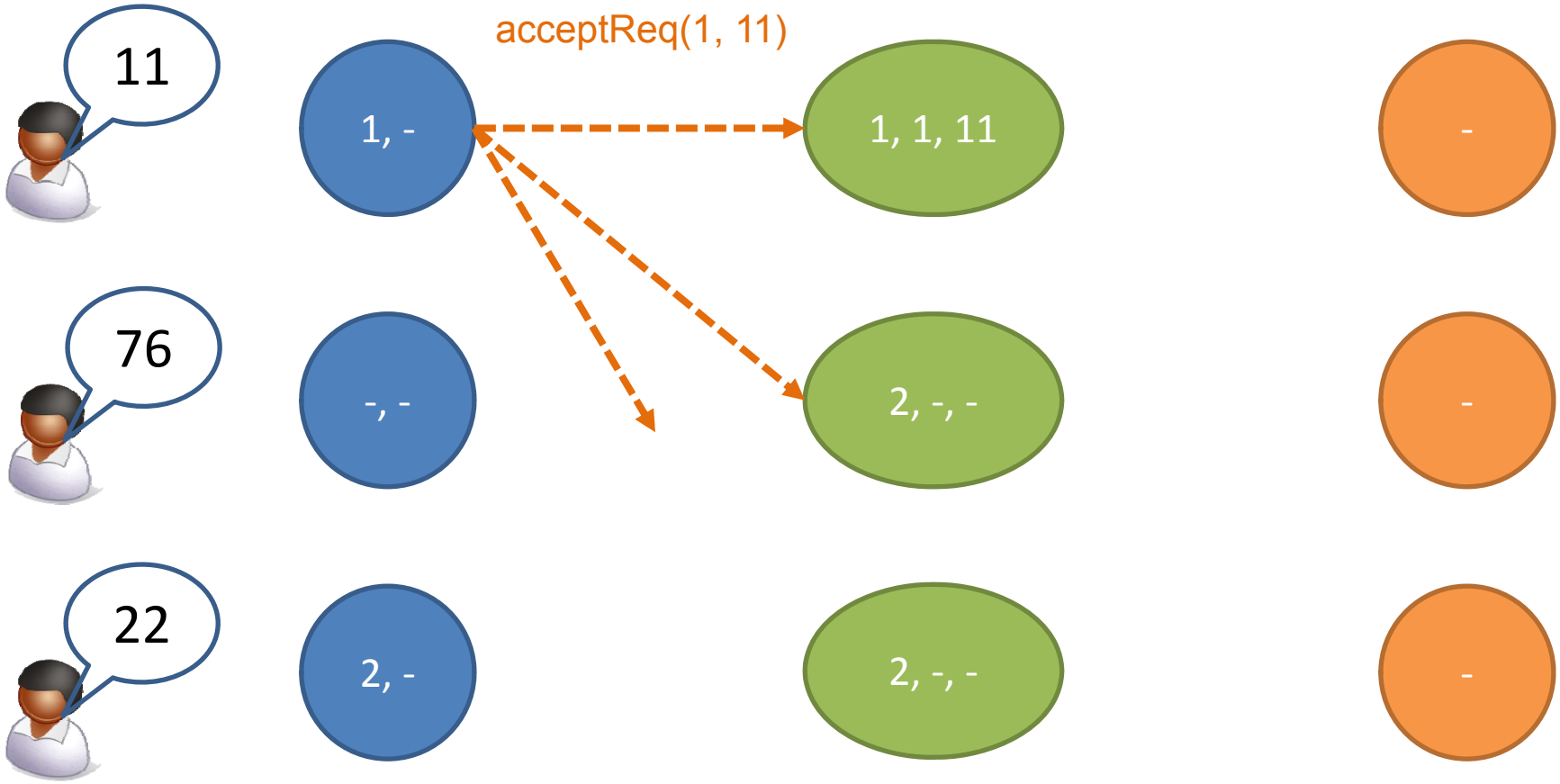
Basic Paxos with comm. failures



Basic Paxos with comm. failures

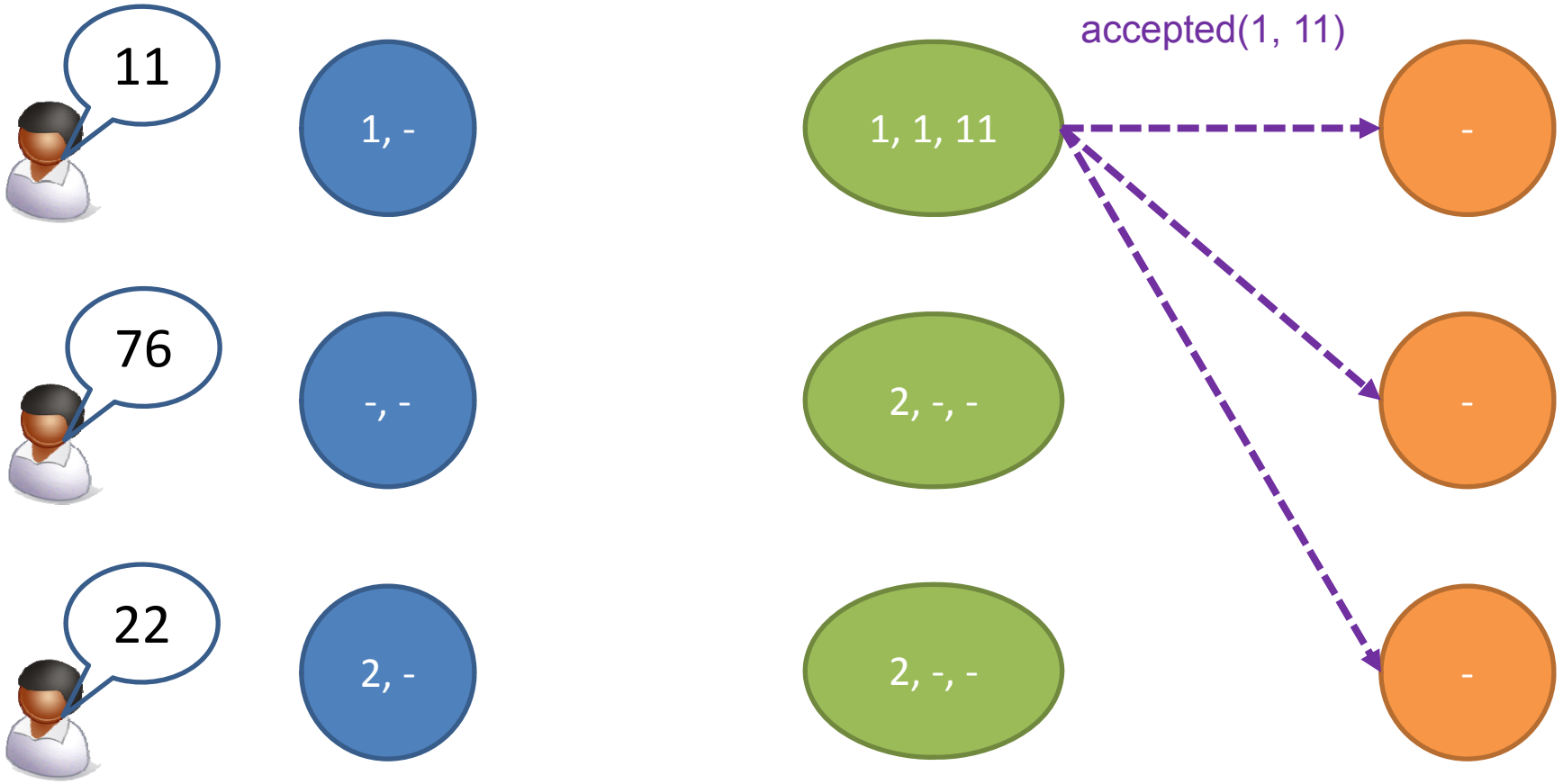


Basic Paxos with comm. failures

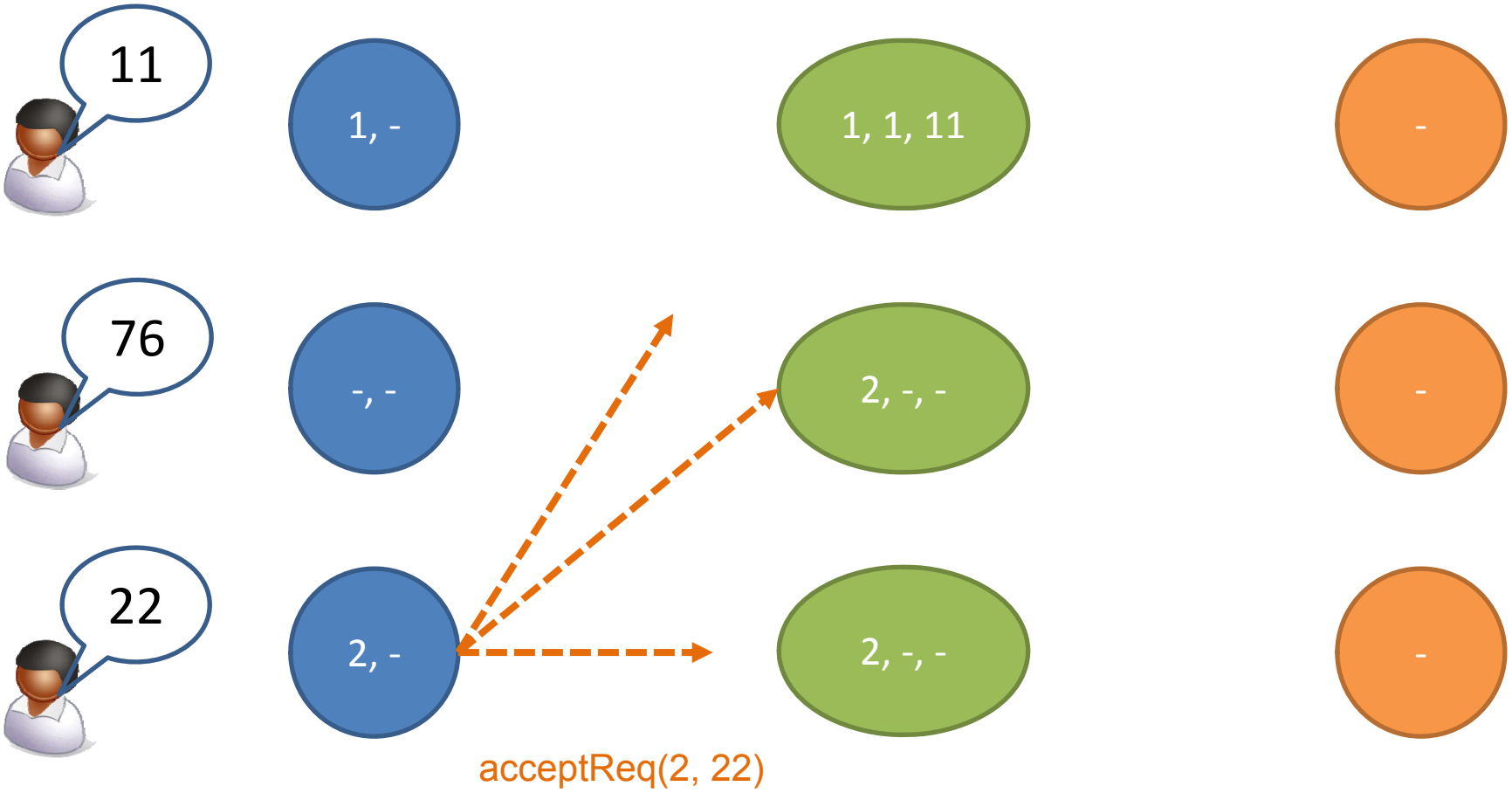


Basic Paxos with comm. failures

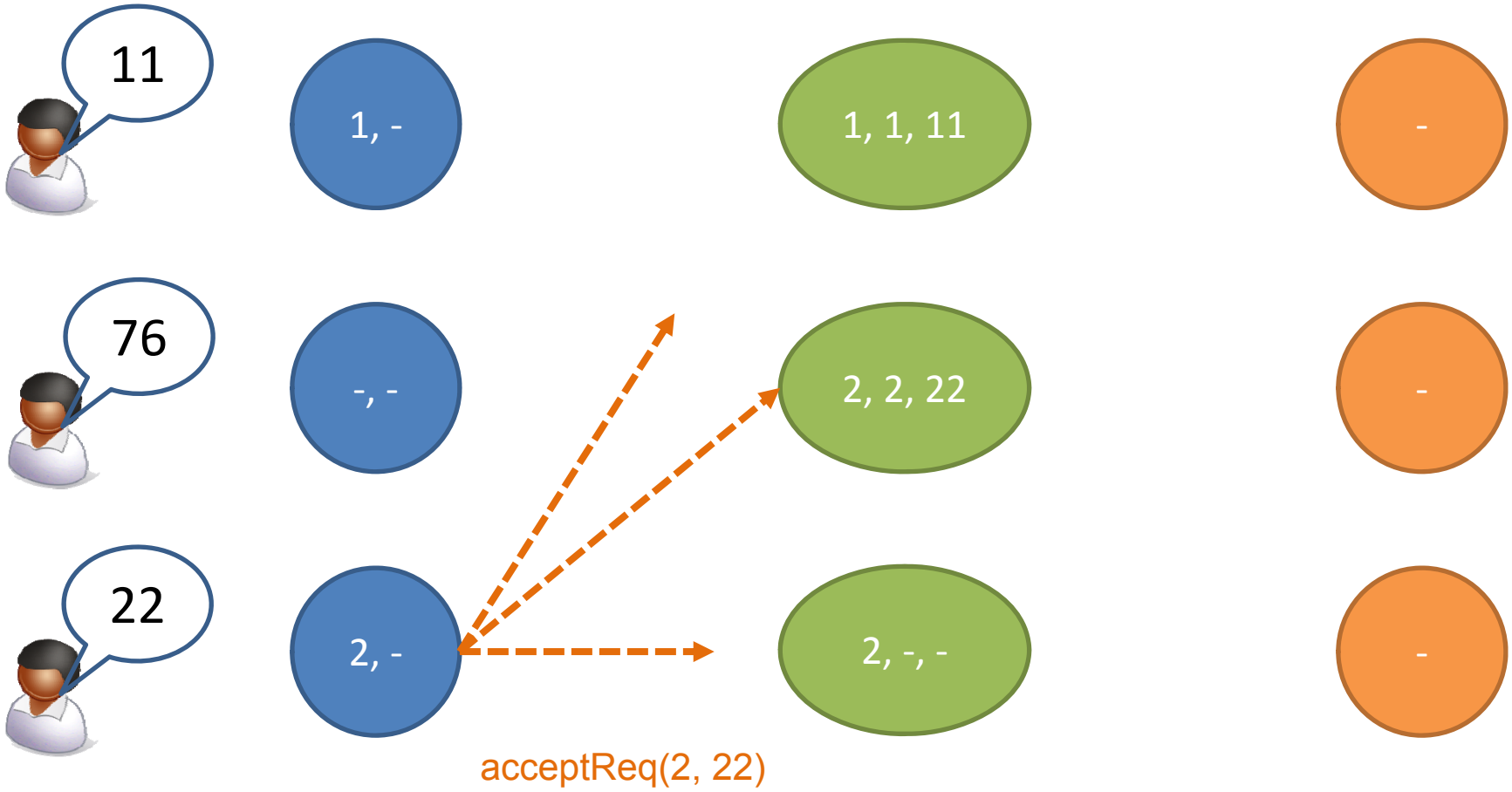
accepted(1, 11) message also send to all **proposers** (not shown here)



Basic Paxos with comm. failures

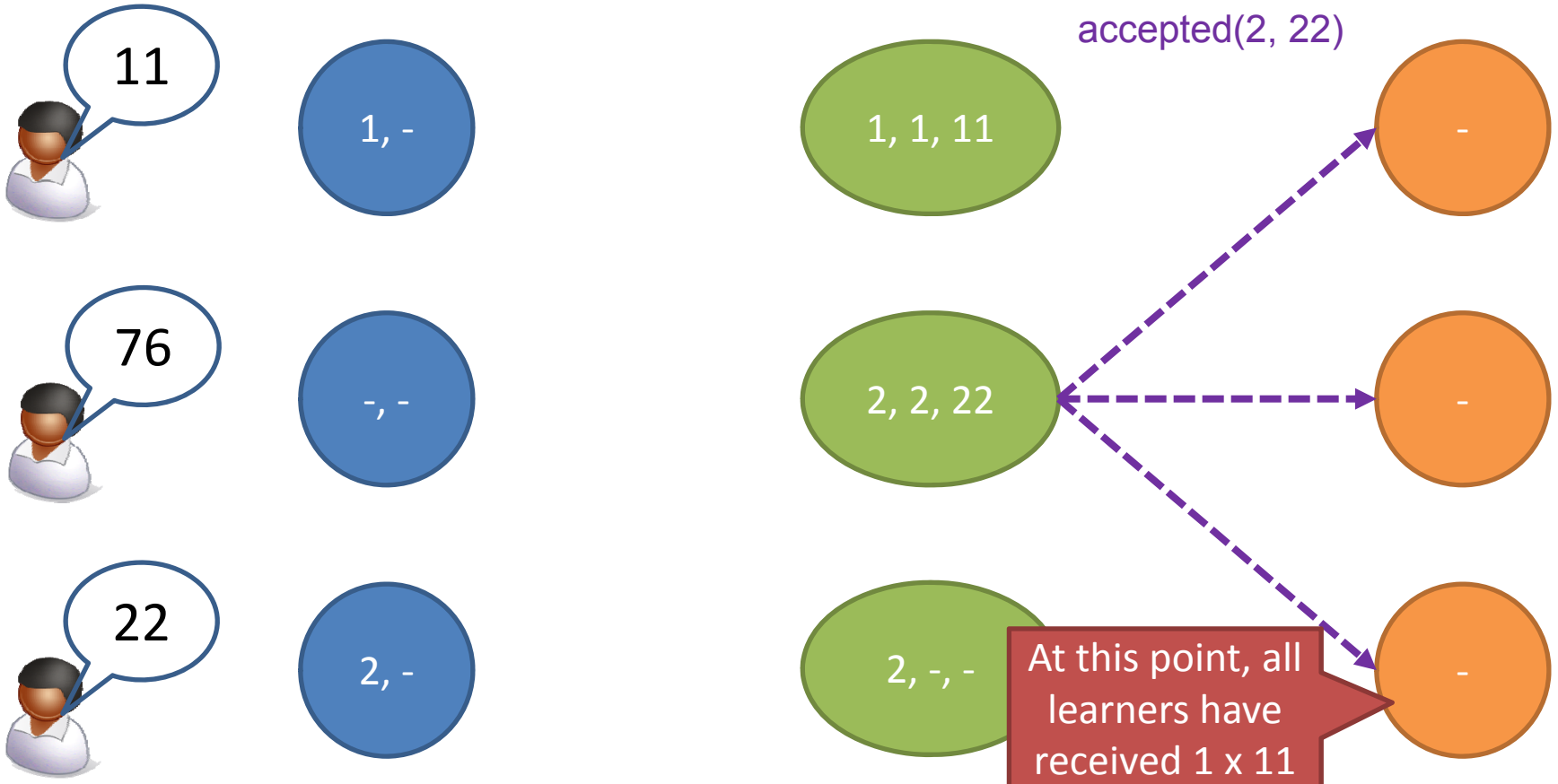


Basic Paxos with comm. failures

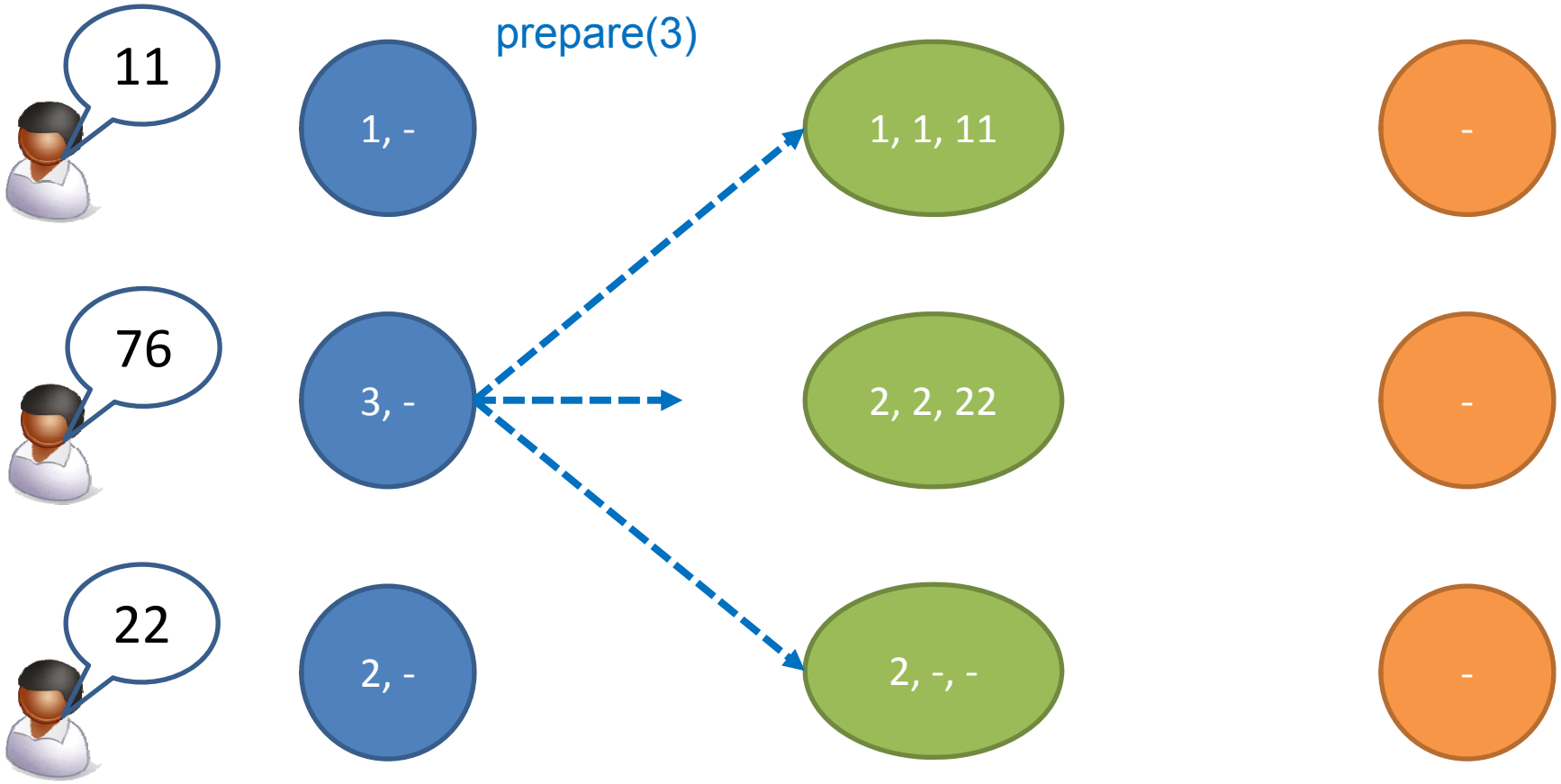


Basic Paxos with comm. failures

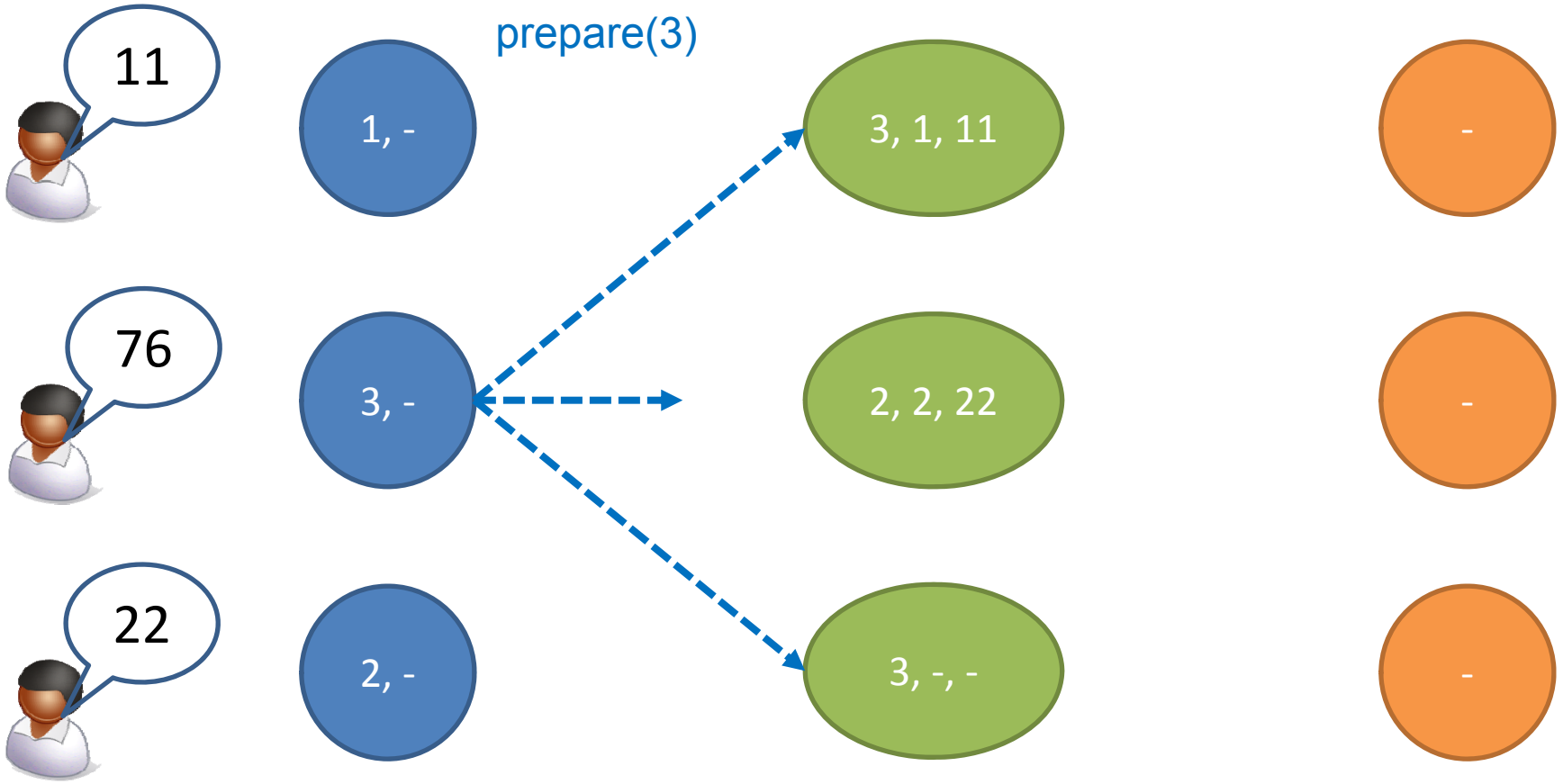
accepted(2, 22) message also send to all **proposers** (not shown here)



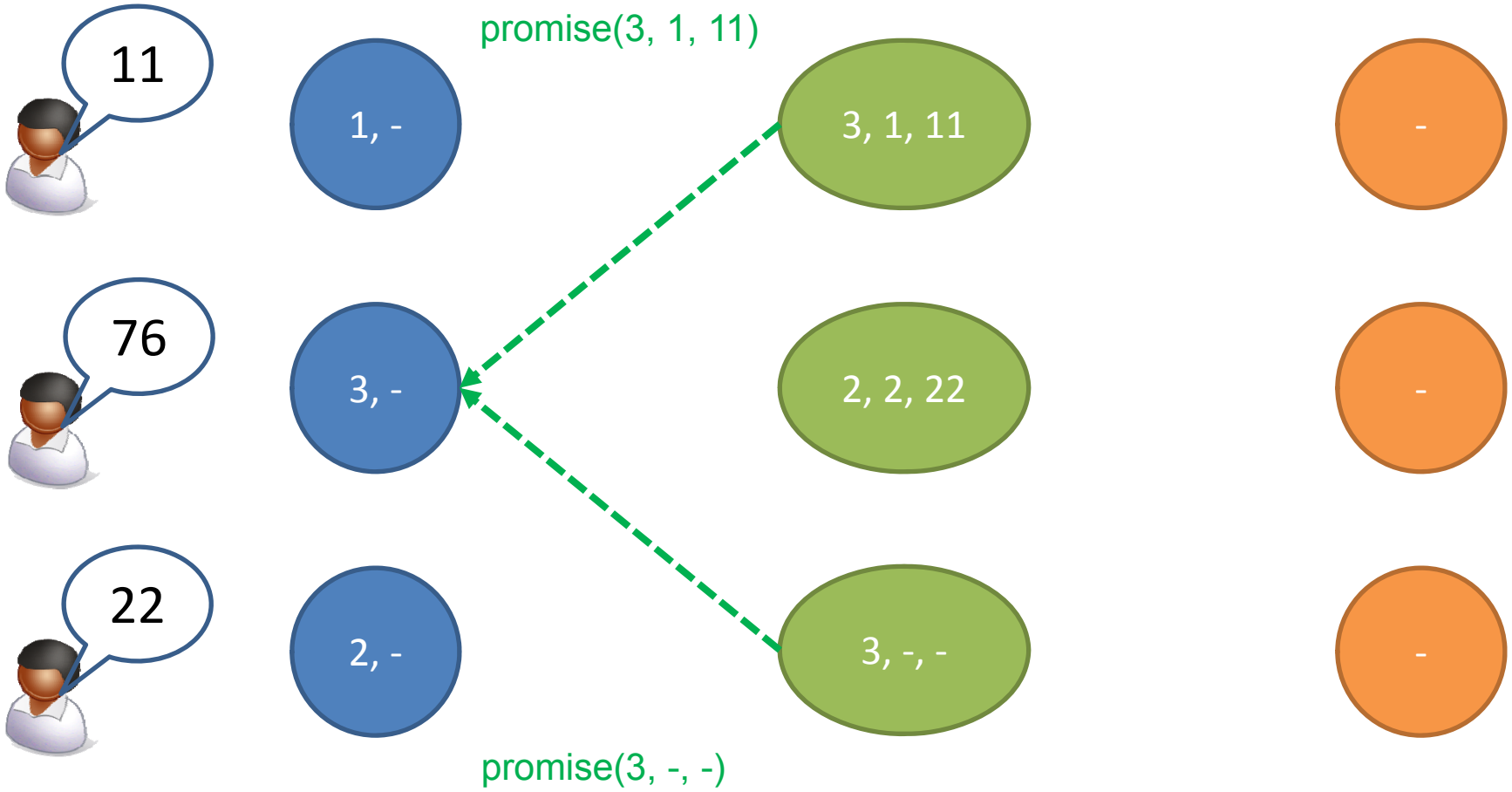
Basic Paxos with comm. failures



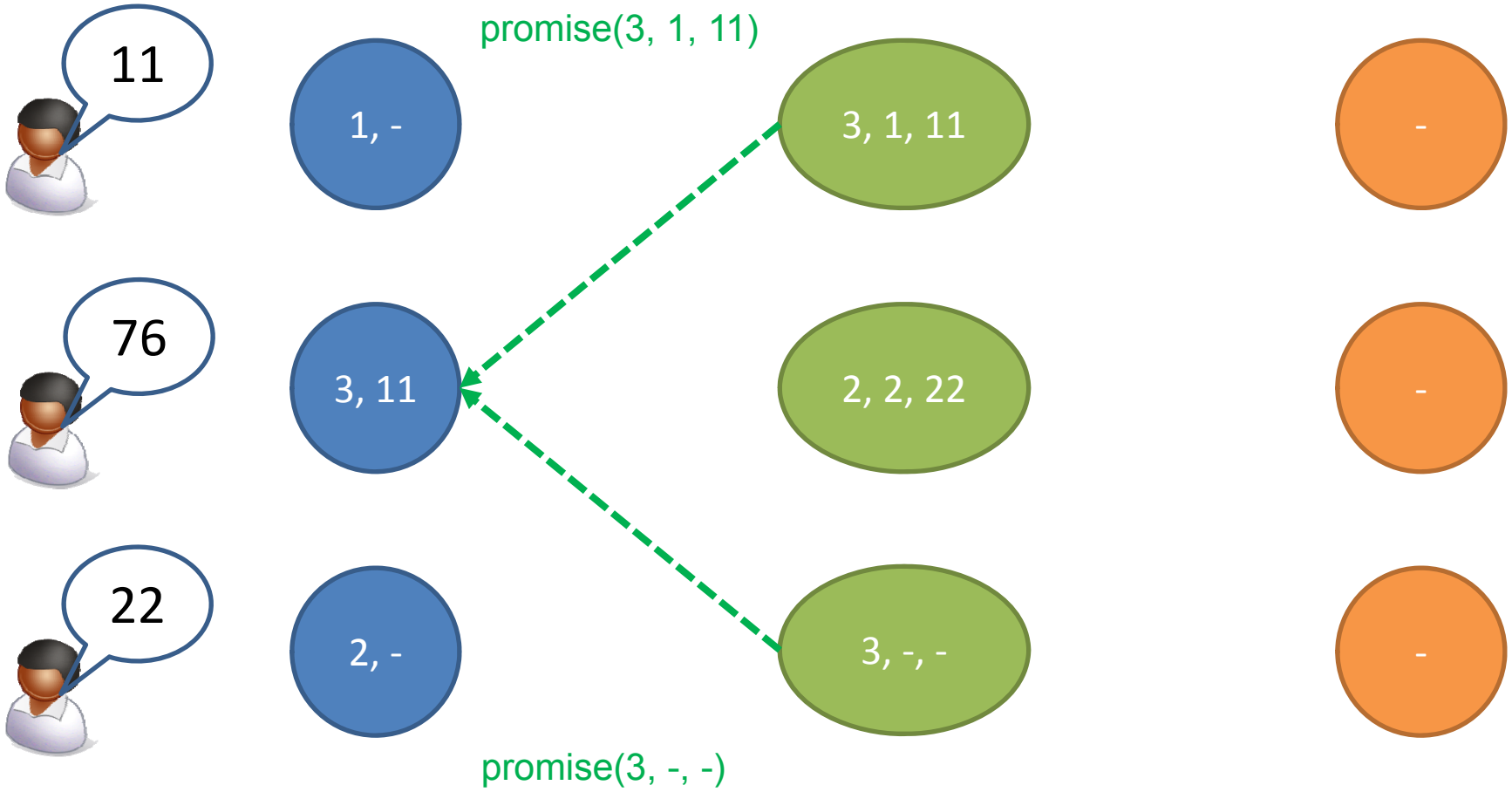
Basic Paxos with comm. failures



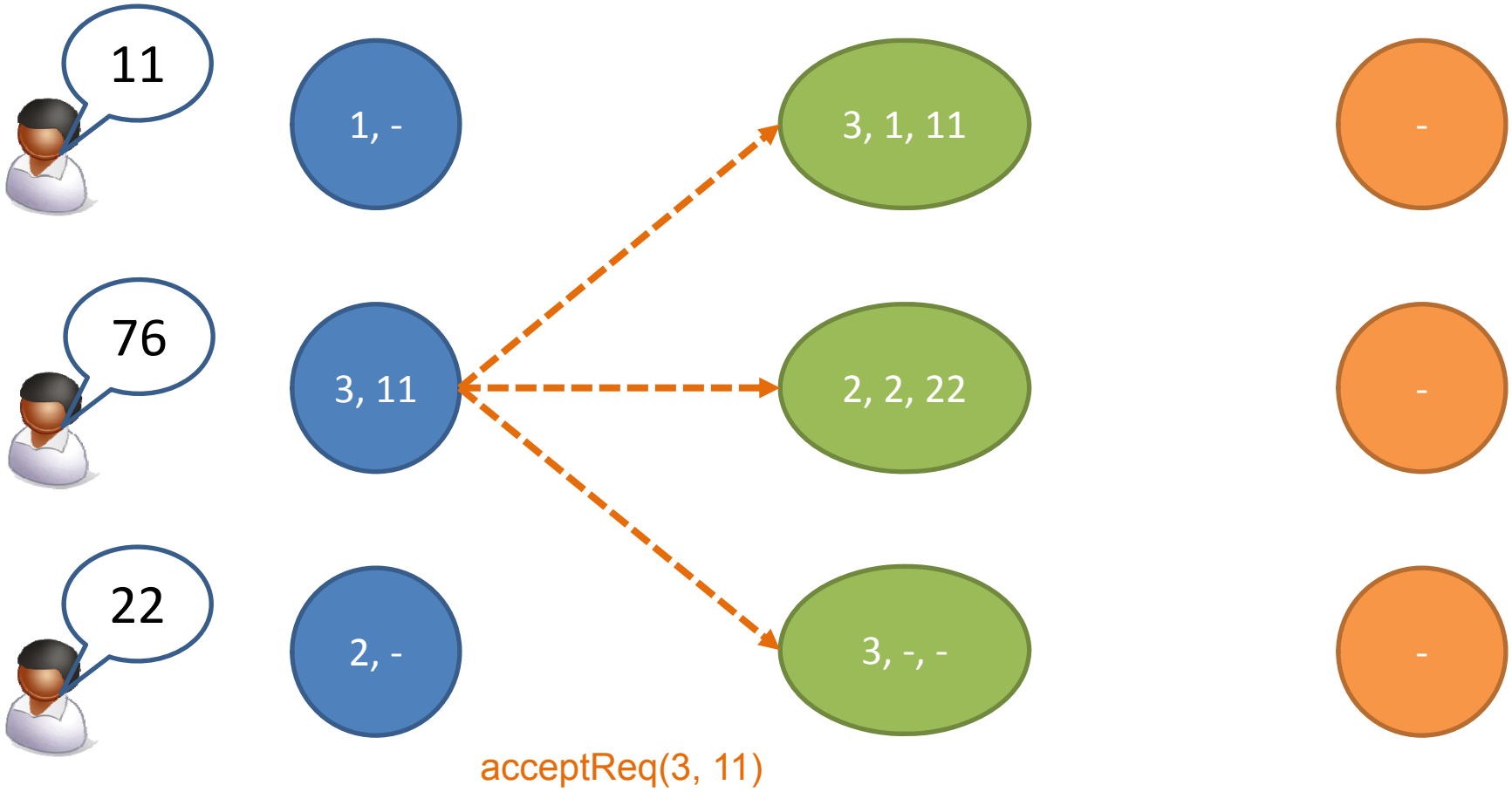
Basic Paxos with comm. failures



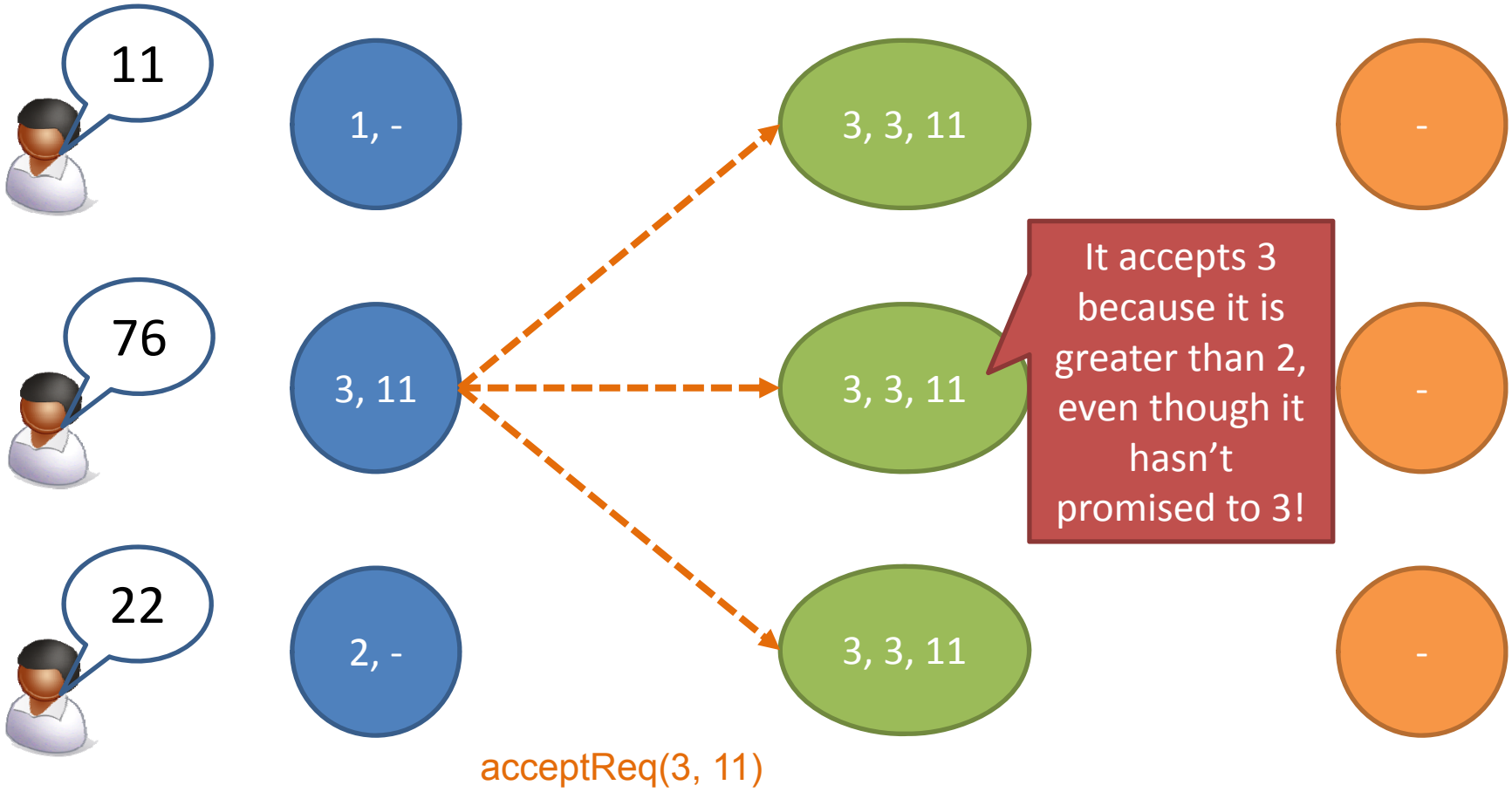
Basic Paxos with comm. failures



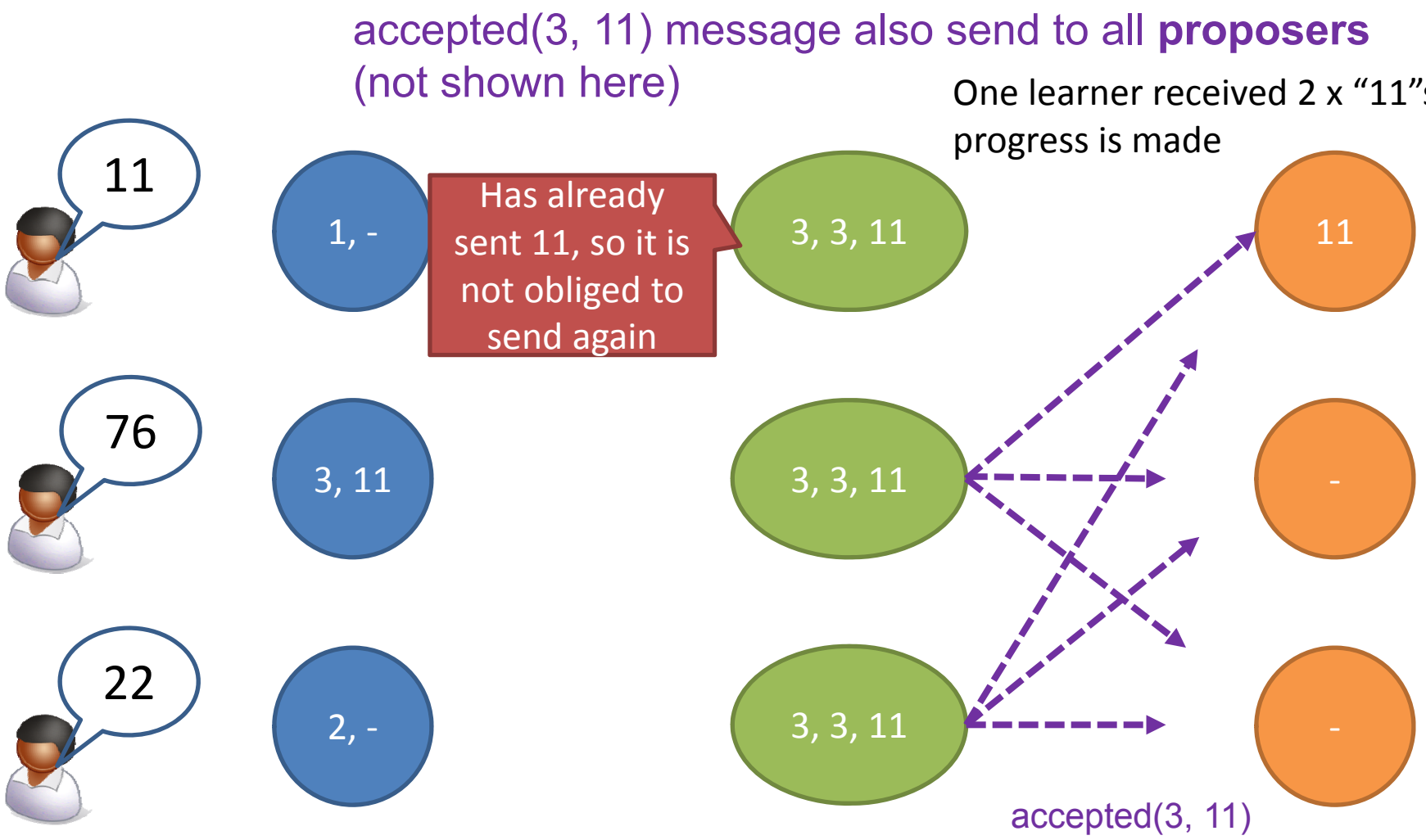
Basic Paxos with comm. failures



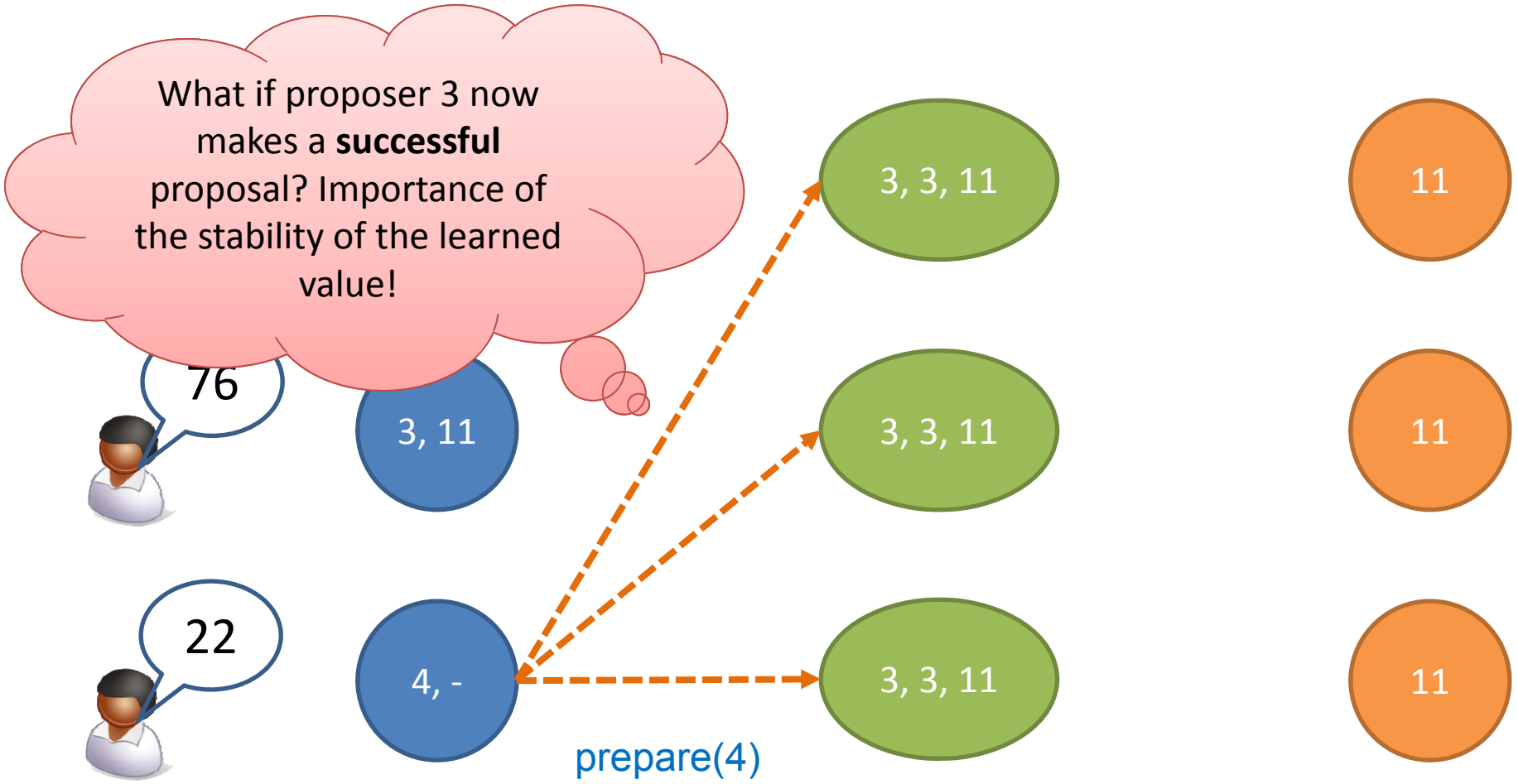
Basic Paxos with comm. failures



Basic Paxos with comm. failures



Basic Paxos with comm. failures



“Paxos family”

- **Basic Paxos**, **Multi**-Paxos, **Cheap** Paxos, **Fast** Paxos, **Byzantine** Paxos etc.
- Protocols with **spectrum of trade-offs** between
  - Number of processes
  - Number of message delays before learning the agreed value
  - Activity level of individual participants
  - Number of messages sent
  - Types of failures tolerated

Discussion

- Configuration management not addressed by Basic Paxos (*n* assumed fix in protocol instance)
- Roles are coalesced in practice, i.e., same node acts a proposer, acceptor and learner
- A client request that does not become the result (consensus value) of an instance of Paxos will have to be retried by the client

Summary

- Consensus in an asynchronous environment with failures and unreliable network assuming $2f+1$ nodes
- Better than previously seen coordination algorithms
- Paxos guarantees safety, but progress may not hold
- Core mechanisms of Paxos are **ordering using proposal numbers** and **quorum accepts** without rollback (e.g., future proposals take older accepted values)
- Paxos is at the heart of Internet-scale systems deployed by major industry players (e.g., Chubby)
- Paxos protocol family has many members; here, Basic Paxos was reviewed (decides only on one value)