Lecture# 9-10 –Replication and Fault Tolerance

S. M. Ahsan Kazmi

Outline

- CONSISTENCY AND REPLICATION
  - DATA-CENTRIC CONSISTENCY MODELS
  - CLIENT-CENTRIC CONSISTENCY MODELS
- CONSISTENCY PROTOCOLS
  - Primary-Based Protocols
  - Replicated-Write Protocols
- Fault Tolerance
  - Consensus/agreement approaches in Faulty Systems
    - Paxos
    - Raft
  - Byzantine Failures
  - Commit protocols
  - Recovery

Data Replication

- Data replication: common technique in distributed systems

- Why?
  - Enhances reliability.
    - If one replica is unavailable or crashes, use another
    - Protect against corrupted data
  - Improves performance.

- Replicas allows remote sites to continue working in the event of local failures.

- Replicas allow data to reside close to where it is used.
  - This directly supports the distributed systems goal of enhancing *scalability*.

Replication (cont.)

- If there are many replicas of the same thing, how do we keep all of them up-to-date or consistent ?
- Consistency can be achieved in a number of ways.
- Challenge:
  - It is not easy to keep all those replicas *consistent*.

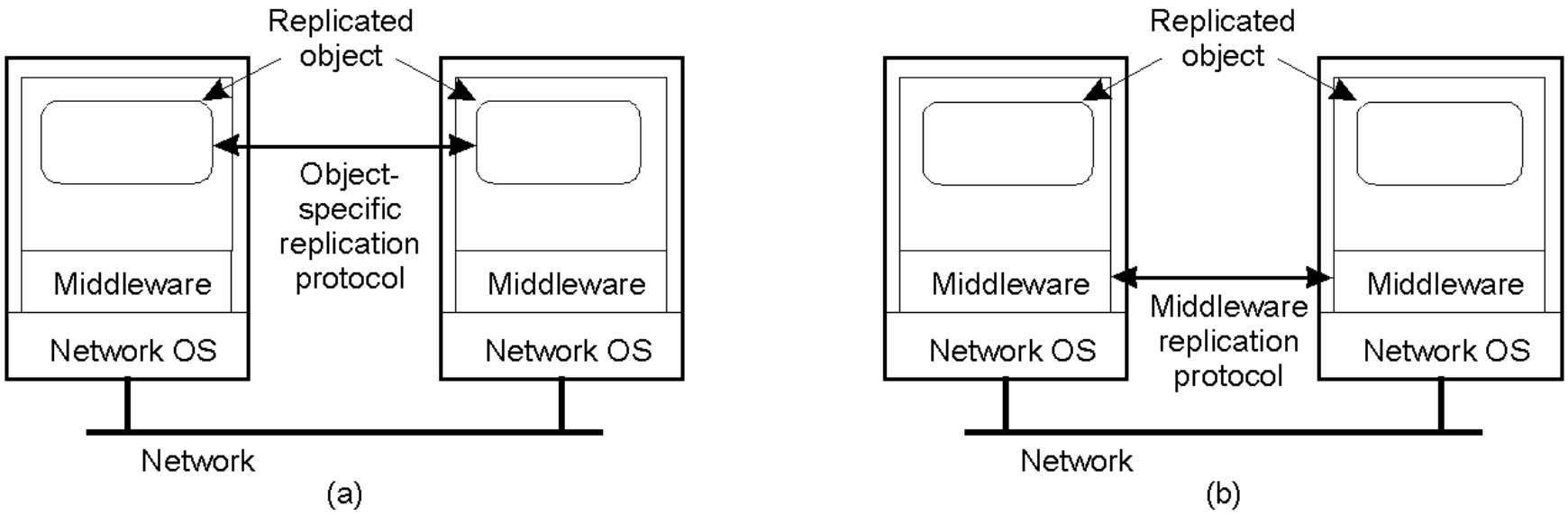
CAP Theorem

- Conjecture by Eric Brewer at PODC 2000 conference
  - It is impossible for a web service to provide all three guarantees:

- Consistency (nodes see the same data at the same time)
- Availability (node failures do not the rest of the system)
- Partition-tolerance (system can tolerate message loss)

- A distributed system can satisfy any two, but not all three, at the same time
- Conjecture was established as a theorem in 2002 (by Lynch and Gilbert)

Object Replication



a) Application is responsible for replication
   - A remote object capable of handling concurrent invocations on its own.

b) System (middleware) handles replication
   - A remote object for which an object adapter is required to handle concurrent invocations (relies on middleware).

Replication and Scalability

- Replication is a widely-used scalability technique:
  - Example: web clients and web proxies.

- When systems scale, the first problem to surface are those associated with performance – as the systems get bigger (e.g., more users), they get often slower.

- Replicating the data and moving it closer to where it is needed helps to solve this scalability problem.

- Dilemma: adding replicas improves scalability, but incurs the (oftentimes considerable) overhead of keeping the replicas up-to-date

Replication and Scaling (cont.)

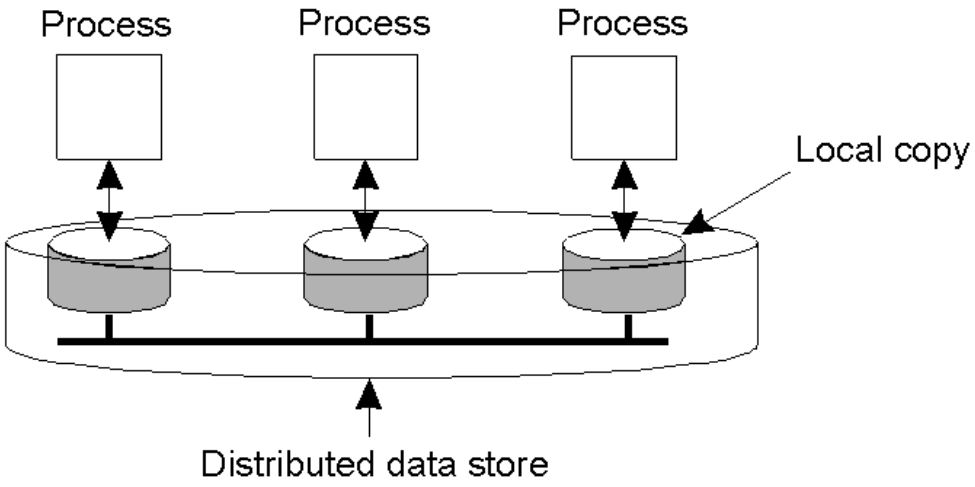
- Replication and caching is used for system scalability

- Multiple copies:
  - Improves performance by reducing access latency
  - But higher network overheads of maintaining consistency
  - Example: object is replicated *N* times
    - Read frequency *R,* write frequency *W*
    - If *R<<W*, high consistency overhead and wasted messages
    - Consistency maintenance is itself an issue
      - What semantics to provide?
      - Tight consistency requires globally synchronized clocks.

- Solution: loosen consistency requirements
  - Variety of consistency semantics possible

Data-Centric Consistency Models

- A data-store can be read from or write to any process in a distributed system.
- A local copy of the data-store (replica) can support “fast reads”.
- However, a write to a local replica needs to be propagated to *all* remote replicas.



- Various consistency models help to understand the various mechanisms used to achieve and enable this.

What is a Consistency Model?

- A "consistency model" is a CONTRACT between a DS data-store and its processes.

- If the processes agree to the rules, the data-store will perform properly and efficiently.

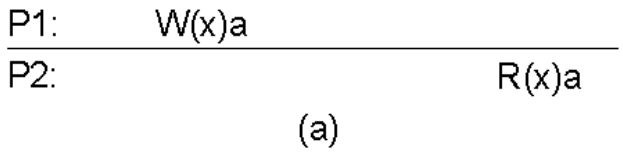
- *Strict Consistency*, which is defined as:

- *Any read on a data item 'x' returns a value corresponding to the result of the most recent write on 'x' (regardless of where the write occurred).*

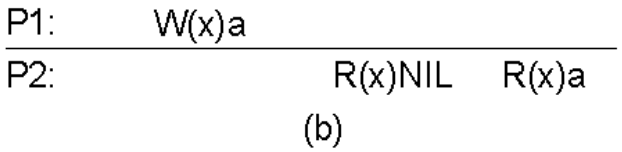
Consistency Model Notation

- $W_i(x)a$ – a write by process 'i' to item 'x' with a value of 'a'. That is, 'x' is set to 'a'.

- $R_i(x)b$ – a read by process 'i' from item 'x' producing the value 'b'. That is, reading 'x' returns 'b'.



(a)



(b)

- Behavior of two processes, operating on the same data item:

a) A strictly consistent data-store.

b) A data-store that is not strictly consistent.

Strict Consistency

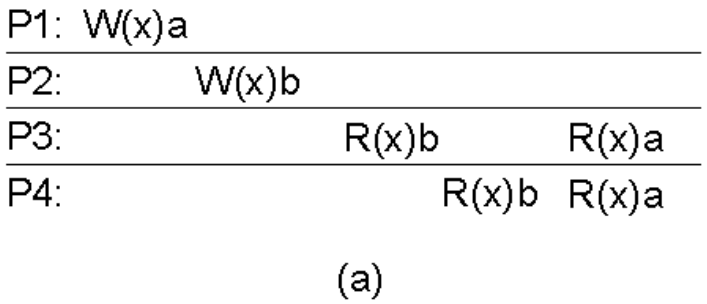
- With *Strict Consistency*, all writes are *instantaneously visible* to all processes and *absolute global time order* is maintained throughout the DS.
- This is the strict consistency model
  - not at all easy in the real world but almost *impossible* within a large DS.
- So, other, less strict (or "weaker") models have been developed

Sequential Consistency

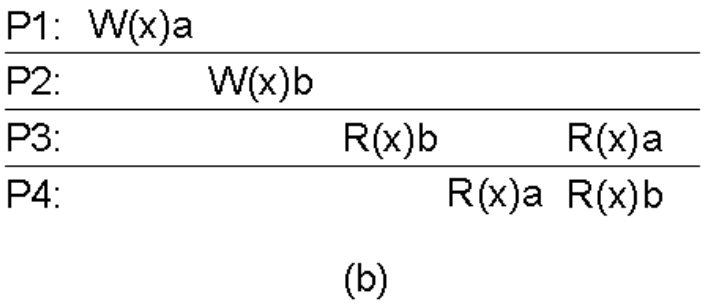
- A weaker consistency model, which represents a relaxation of the rules.
- It is also must easier (possible) to implement.
- Definition of “Sequential Consistency”:
- *The result of any execution is the same as if the (read and write) operations by all processes on the data-store were executed in the same sequential order and the operations of each individual process appear in this sequence in the order specified by its program.*

Sequential Consistency

All processes see the same interleaving set of operations, regardless of what that interleaving is.



(a)



(b)

a) A sequentially consistent data-store – the "first" write occurred *after* the "second" on all replicas.

b) A data-store that is not sequentially consistent – it appears the writes have occurred in a non-sequential order, and this is NOT allowed.

FIFO Consistency

- Defined as follows:

  Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

- The attractive characteristic of FIFO is that it is easy to implement.

- There are no guarantees about the order in which different processes see writes: except that two or more writes from a single process must be seen in order.

FIFO Consistency Example

P1: W(x)a						
P2:	R(x)a	W(x)b	W(x)c			
P3:				R(x)b	R(x)a	R(x)c
P4:				R(x)a	R(x)b	R(x)c

- A valid sequence of FIFO consistency events.
- Note that none of the consistency models studied so far would allow this sequence of events.

Client-Centric Consistency Models

- The previously studied consistency models concern themselves with maintaining a consistent (globally accessible) data-store in the presence of concurrent read/write operations.
- Another class of distributed data-store is that which is characterized by *the lack of simultaneous updates*.
- The emphasis is more on maintaining a consistent view of things *for the individual client process* that is currently operating on the data-store.

Client-Centric Consistency

- Assume read operations by a single process *P* at two *different* local copies of the same data store. Four consistency semantics:

- *Monotonic reads:* Once read, subsequent reads on that data items return the same or more recent values

- *Monotonic writes:* A write must be propagated to all replicas before a successive write by the *same process, r*esembles FIFO consistency

- *Read your writes*: read(x) always returns write(x) by that process

- *Writes follow reads*: write(x) following read(x) will take place on same or more recent version of x

Client-Centric Consistency

- How fast should updates (writes) be made available to read-only processes?
  - Most database systems: *mainly read*.
  - DNS: *write-write conflicts* do not occur.
  - WWW: as with DNS, except that heavy use of client-side caching is present: *even the return of stale pages is acceptable to most users*.

- These systems all exhibit a high degree of acceptable inconsistency
  - *replicas* gradually become consistent over time.

Eventual Consistency

- Assume a replicated database with few updaters and many readers
- Eventual consistency: in absence of updates, all replicas converge towards identical copies
- The only requirement is that all replicas will *eventually* be the same.
- Cheaper to implement
- This works well if every client always updates the same replica.
- Things are a little difficult if the clients are *mobile*.

Epidemic Protocols

- This is an interesting class of protocol that can be used to implement *Eventual Consistency*
- The main concern is the propagation of updates to all the replicas in *as few a number of messages as possible*.
- With this "update propagation model", the idea is to "infect" as many replicas as quickly as possible.

The Anti-Entropy Protocol

- Server P picks Q at random and exchanges updates, using one of three approaches:
  1. P only pushes to Q.
  2. P only pulls from Q.
  3. P and Q push and pull from each other.

- A pure push-based approach does not spread updates quickly.
  - If we just push, we may end up just talking to machines that have already seen the updates.

- Sooner or later, all the servers in the system will be infected (updated).

The Gossiping Protocol

- This variant is referred to as “gossiping” or “rumor spreading”, as works as follows:

1. P has just been updated for item ‘x’.
2. It immediately pushes the update of ‘x’ to Q.
3. If Q already knows about ‘x’, P becomes disinterested in spreading any more updates (rumors) with prob 1/k.
4. Otherwise P gossips to another server, as does Q.

- This approach is good, but can be shown not to guarantee the propagation of all updates to all servers.

The Best of Both Worlds

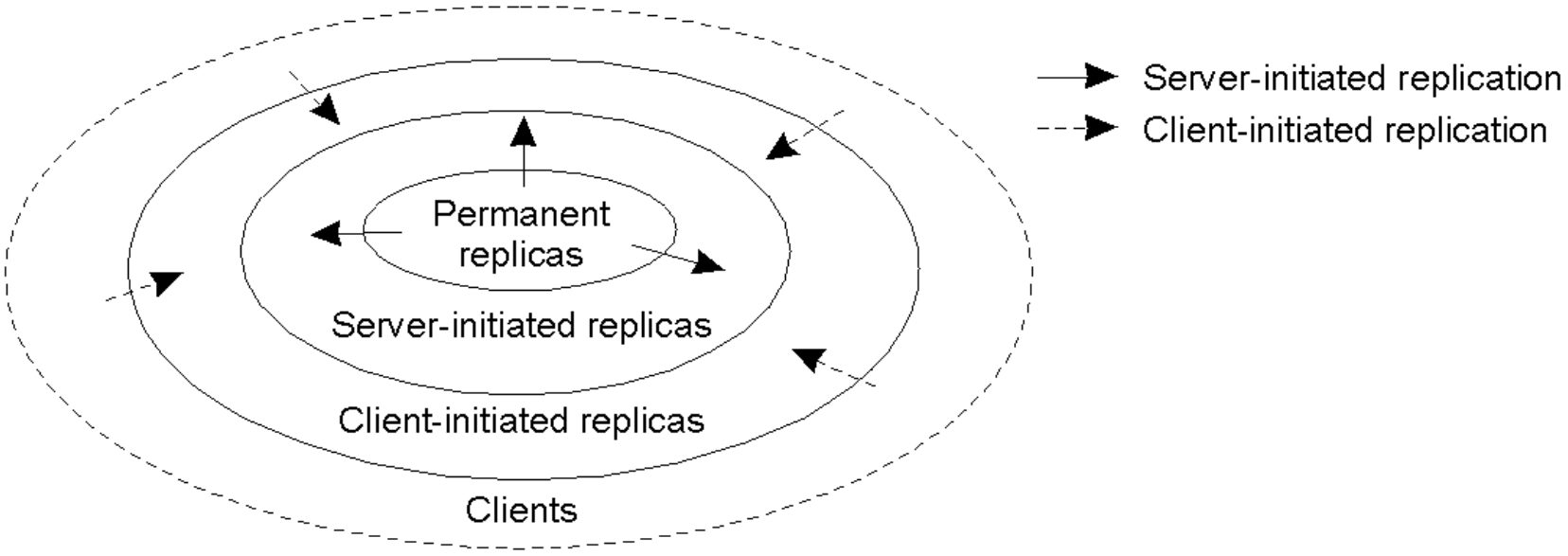
- A mix of anti-entropy and gossiping is regarded as the best approach to rapidly infecting systems with updates.

- However, what about ***removing*** data?

- Updates are easy, deletion is much, much harder!
- Under certain circumstances, after a deletion, an "old" reference to the deleted item may appear at some replica and cause the deleted item to be *reactivated*!
- One solution is to issue "Death Certificates" for data items – these are a special type of update.

Distribution Protocols

- *Regardless of which consistency model is chosen, we need to decide **where**, **when** and **by whom** copies of the data-store are to be placed.*
- *Replica management*



Replica Placement Types

- There are three types of replica:

1. ***Permanent replicas***: tend to be small, organized as COWs (Clusters of Workstations) or mirrored systems.

2. ***Server-initiated replicas***: used to enhance performance at the initiation of the owner of the data-store.
   - Typically used by web hosting companies to geographically locate replicas close to where they are needed most. (Often referred to as "push caches").

3. ***Client-initiated replicas***: created as a result of client requests – think of browser caches.

Consistency Protocols

- Two popular techniques to implement consistency models

- **Primary-based protocols**
  - Assume a primary replica for each data item
  - Primary responsible for coordinating all operations

- **Replicated write protocols**
  - No primary is assumed for a data item
  - Writes can take place at any replica

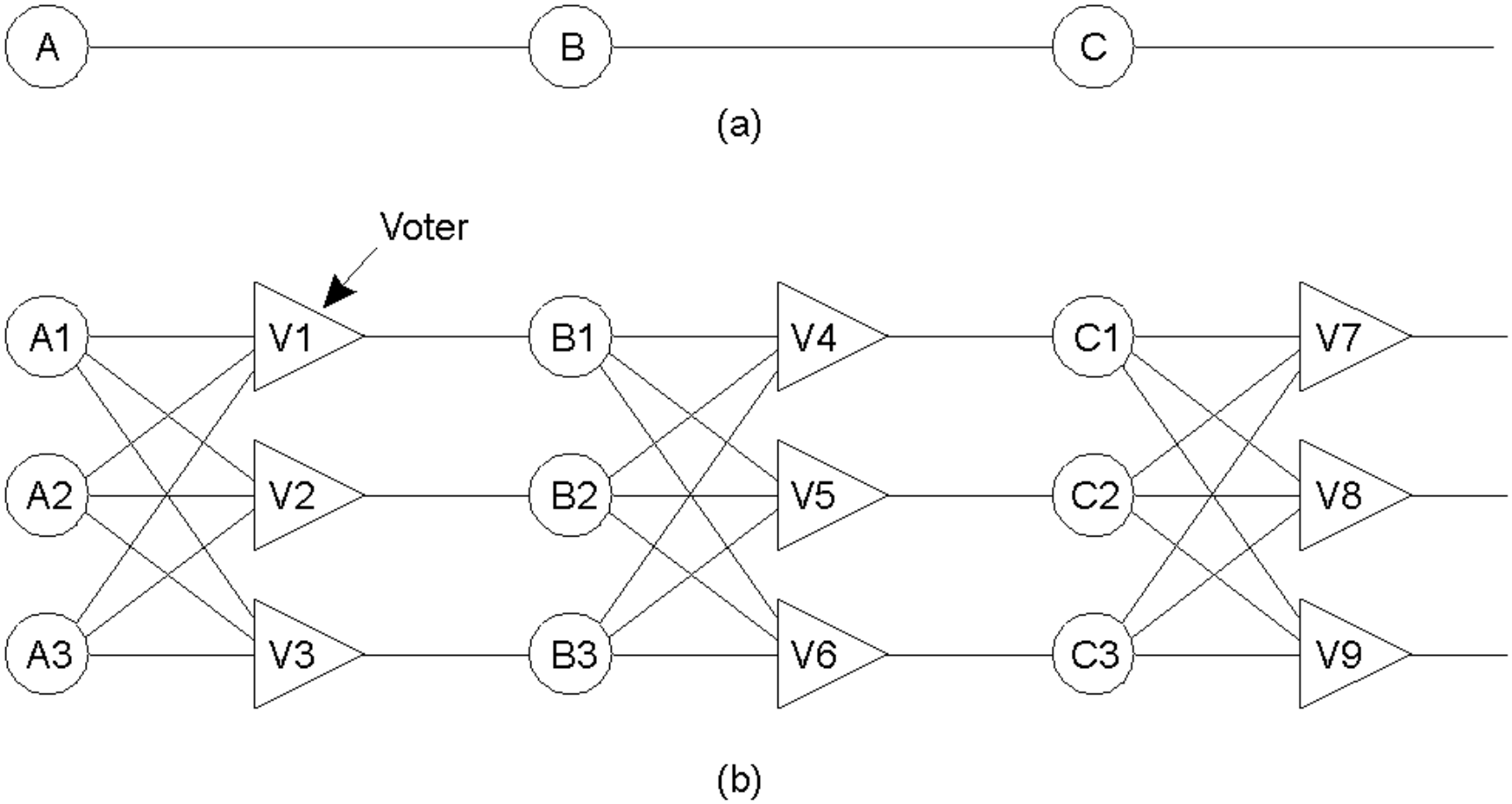
Summary: Replication

- Replication and caching improve performance in distributed systems, however, its expensive to maintain consistency
- Consistency of replicated data is crucial
- Many consistency semantics (models) possible
  - Need to pick appropriate model depending on the application
  - Example: web caching: weak consistency is OK since humans are tolerant to stale information (can reload browser)
  - Implementation overheads and complexity grows if stronger guarantees are desired

Fault Tolerance

- A DS should be fault-tolerant
  - Should be able to continue functioning in the presence of faults
- Fault tolerance is related to **dependability**
- Dependability factors:
  - Availability
  - Reliability
  - Safety
  - Maintainability

Example – Redundancy in Circuits



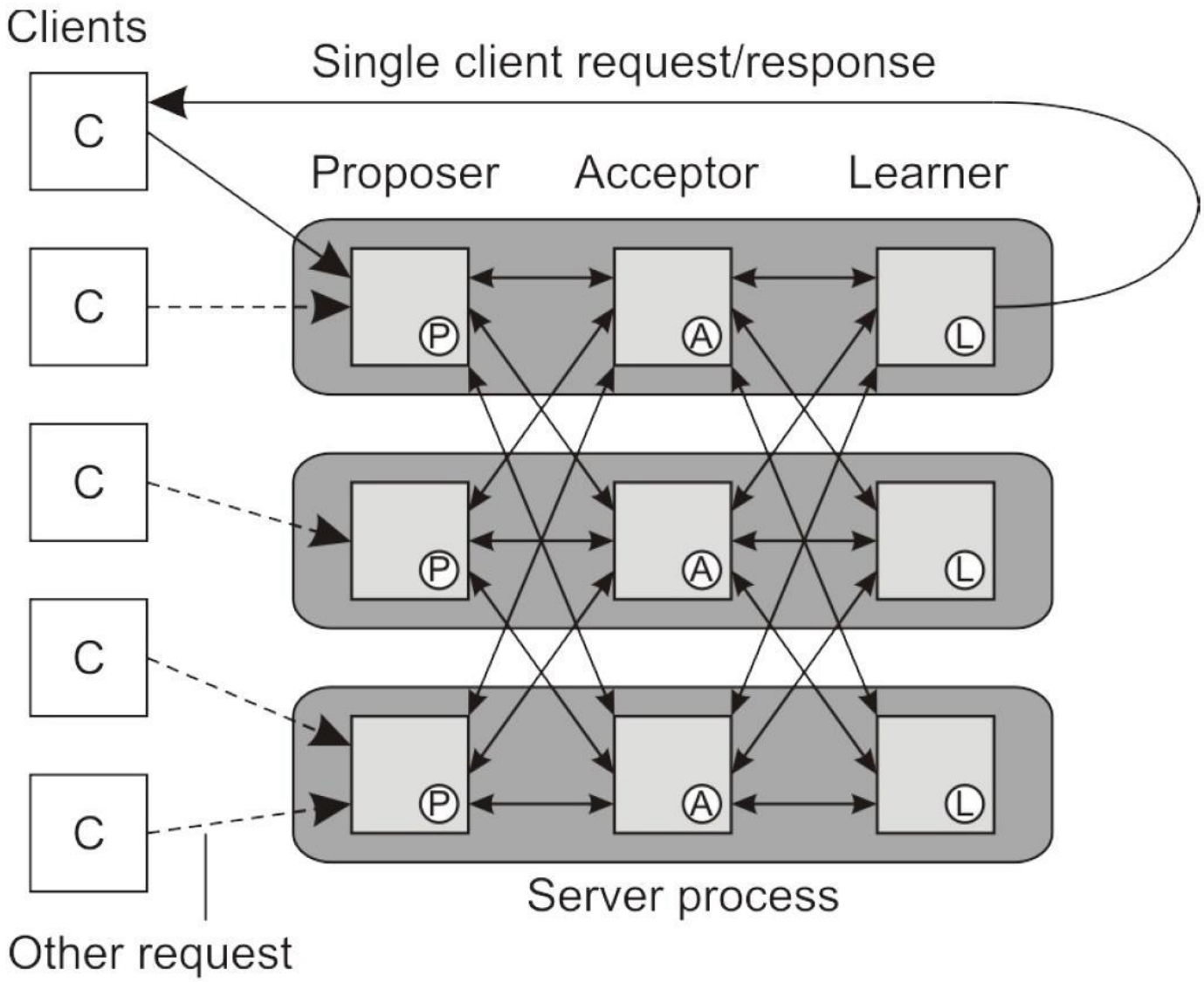
Paxos

- Assumptions:
  - An asynchronous system
  - Communication may be unreliable (meaning that messages may be lost, duplicated, or reordered)
  - Corrupted messages are detectable (and can thus be discarded)
  - All operations are deterministic
  - Process may exhibit halting failures, but not arbitrary failures

Essential Paxos

- A collection of (replicated) threads, collectively fulfilling the following roles:
  - **Client**: a thread that requests an operation
  - **Learner**: a thread that performs an operation
  - **Acceptor**: a thread that operates in a quorum to vote for the execution of an operation
  - **Proposer**: a thread that takes a client's request and attempts to have the requested operation accepted for execution

Paxos

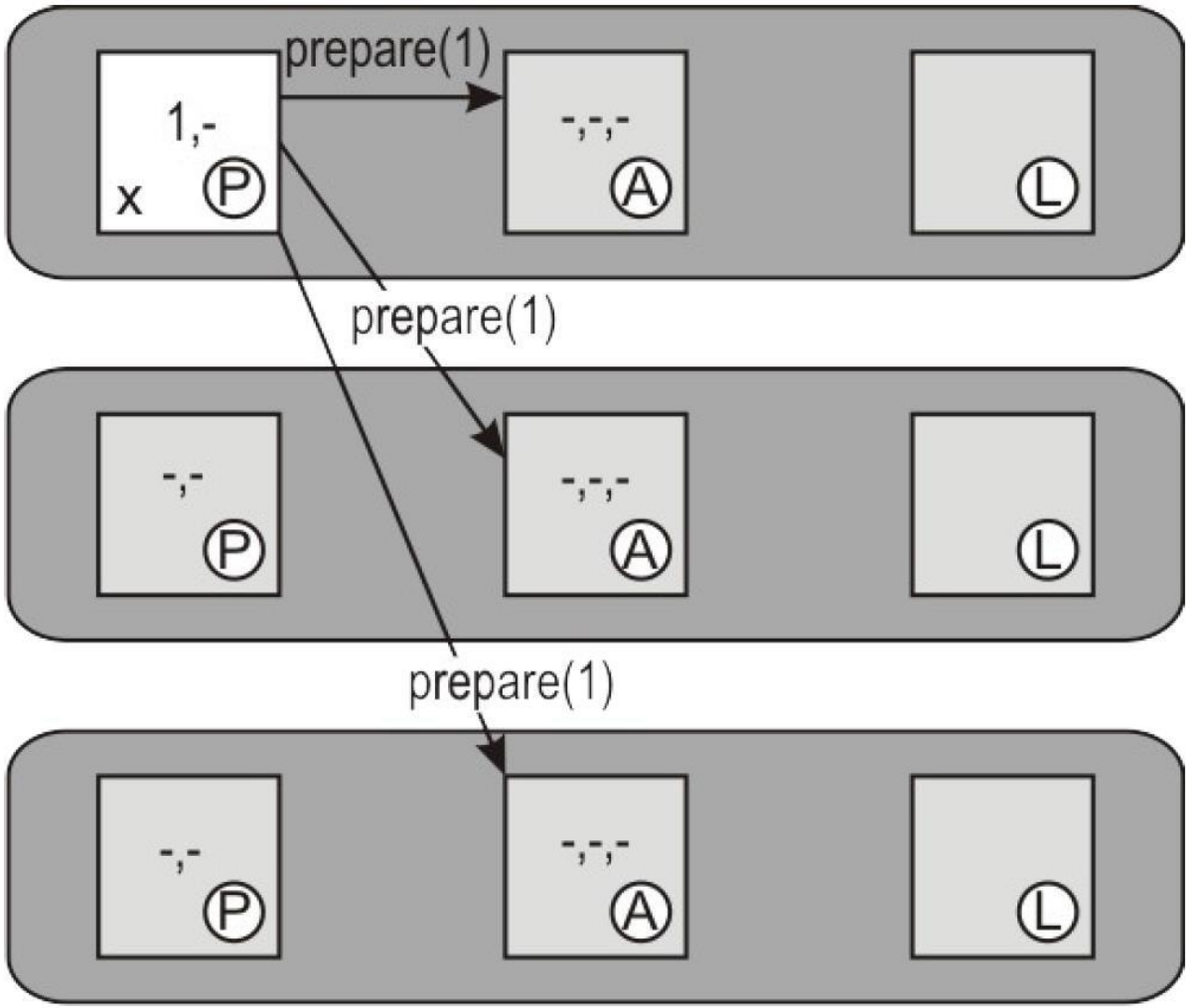

Clients
Single client request/response
C
C
C
C
C
Proposer
Acceptor
Learner
P
A
L
P
A
L
P
A
L
Server process
Other request

Paxos: Phase 1a (prepare)

- A proposer P:
  - has a unique ID, say i
  - communicates only with a quorum of acceptors
  - For requested operation cmd:
    - Selects a counter m higher than any of its previous counters, leading to a proposal number r = (m,i).
    - Note: (m,i) < (n,j) iff m < n
    - Sends prepare(r) to a majority of acceptors

- Goal:
  - Proposer tries to get its proposal number anchored: any previous proposal failed, or also proposed cmd.
  - Note: previous is defined by wrt proposal number

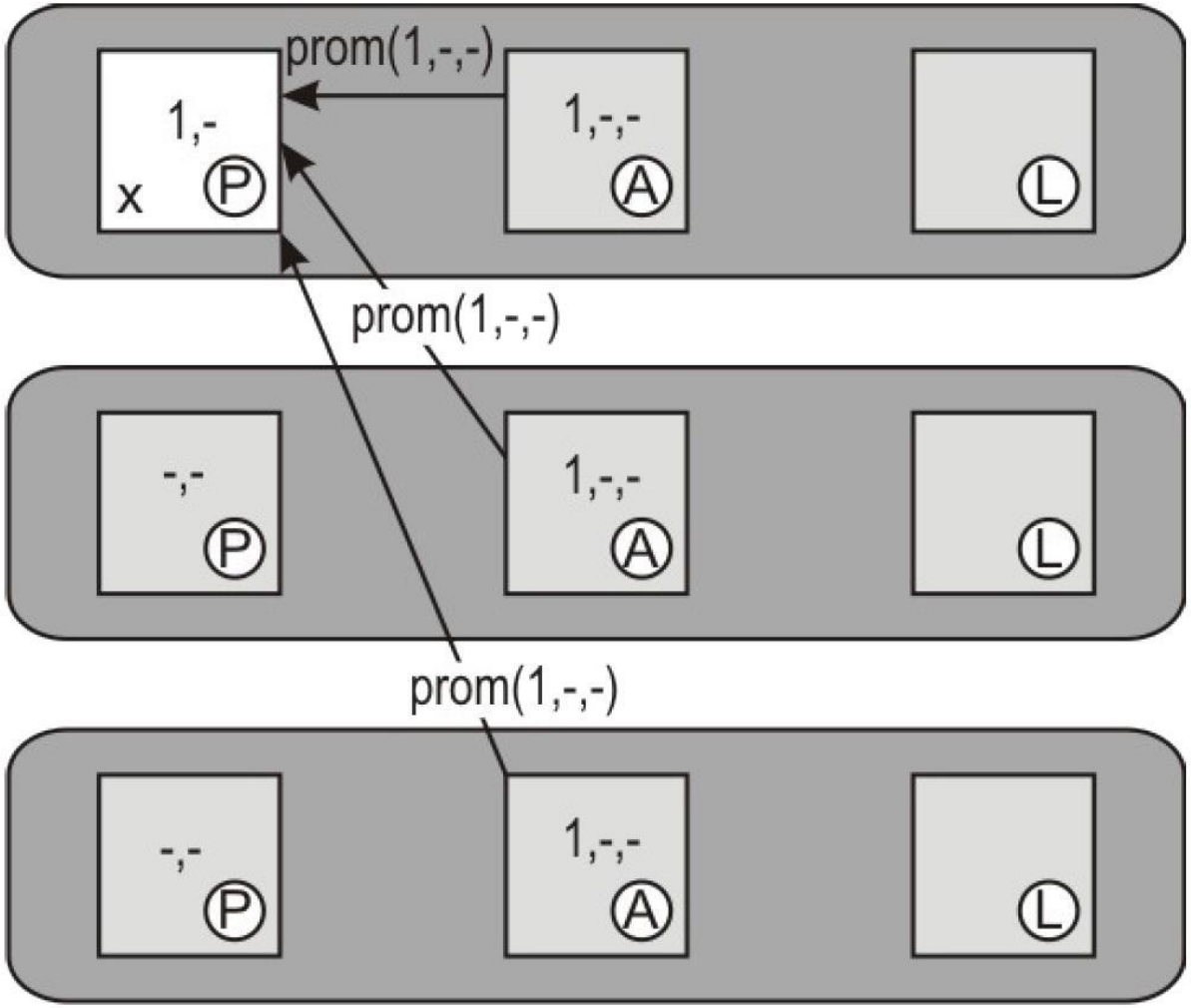
Paxos: Phase 1a (prepare)



Paxos: Phase 1b (promise)

- What the acceptor does:
  - If r is the highest from any proposer:
    - Return promise(r) to p, telling the proposer that the acceptor will ignore any future proposals with a lower proposal number.
  - If r is the highest, but a previous proposal (r',cmd') had already been accepted:
    - Additionally return (r',cmd') to p. This will allow the proposer to decide on the final operation that needs to be accepted.
  - Otherwise: do nothing –there is a proposal with a higher proposal number in the works

Paxos: Phase 1b (promise)

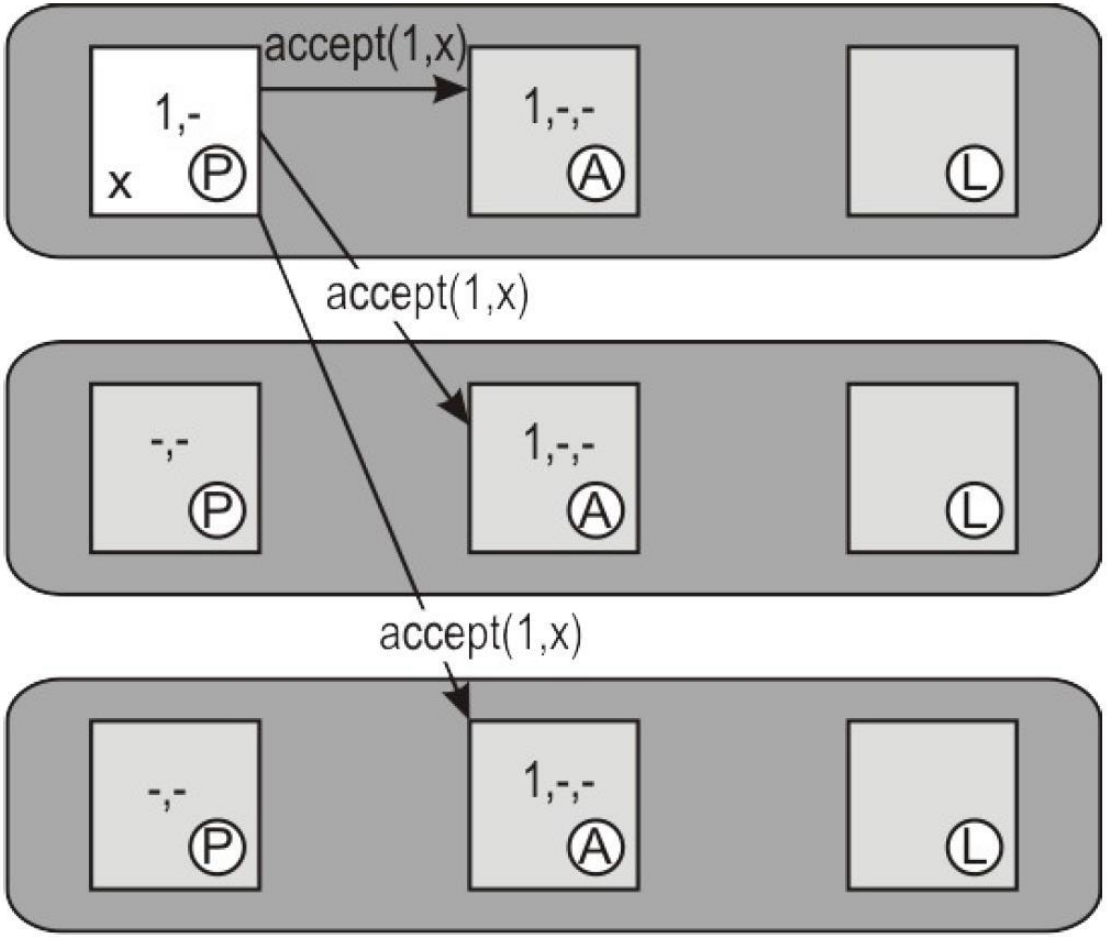


Paxos: Phase 2a (accept)

- It's the proposer's turn again:
  - If it does not receive any accepted operation, it sends accept(r,cmd) to a majority of acceptors

- If it receives one or more accepted operations, it sends accept(r,cmd*), where
  - r is the proposer's selected proposal number
  - cmd* is the operation whose proposal number is the highest among all accepted operations received from acceptors.

Paxos: Phase 2a (accept)



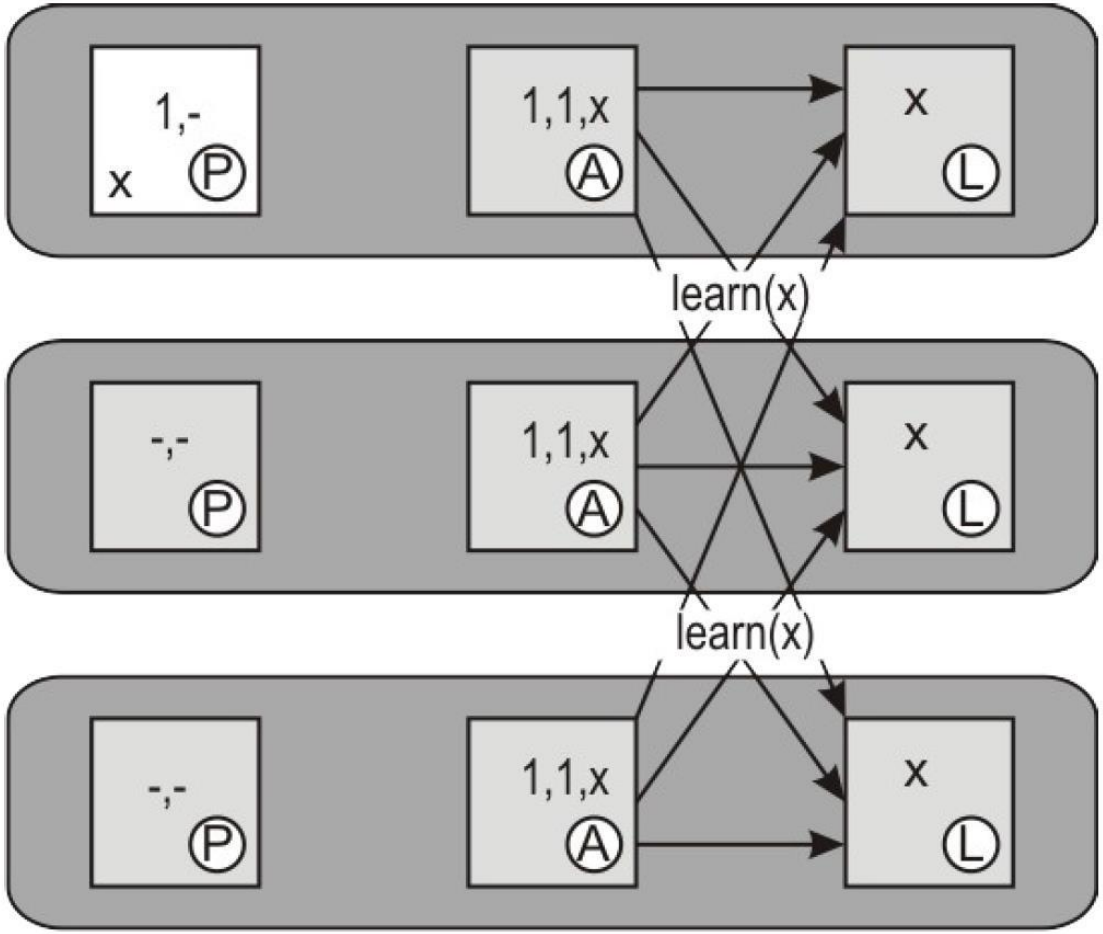
Paxos: Phase 2b (learn)

- An acceptor receives an accept(r,cmd) message:
  - If it did not send a promise(r') with r' > r, it must accept cmd, and says so to the learners: learn(cmd).

- A learner receiving learn(cmd) from most acceptors, will execute the operation cmd.

- Observation:
  - The essence of Paxos is that the proposers drive most of the acceptors to the accepted operation with the highest anchored proposal number

Paxos: Phase 2b (learn)



Safety and liveness properties

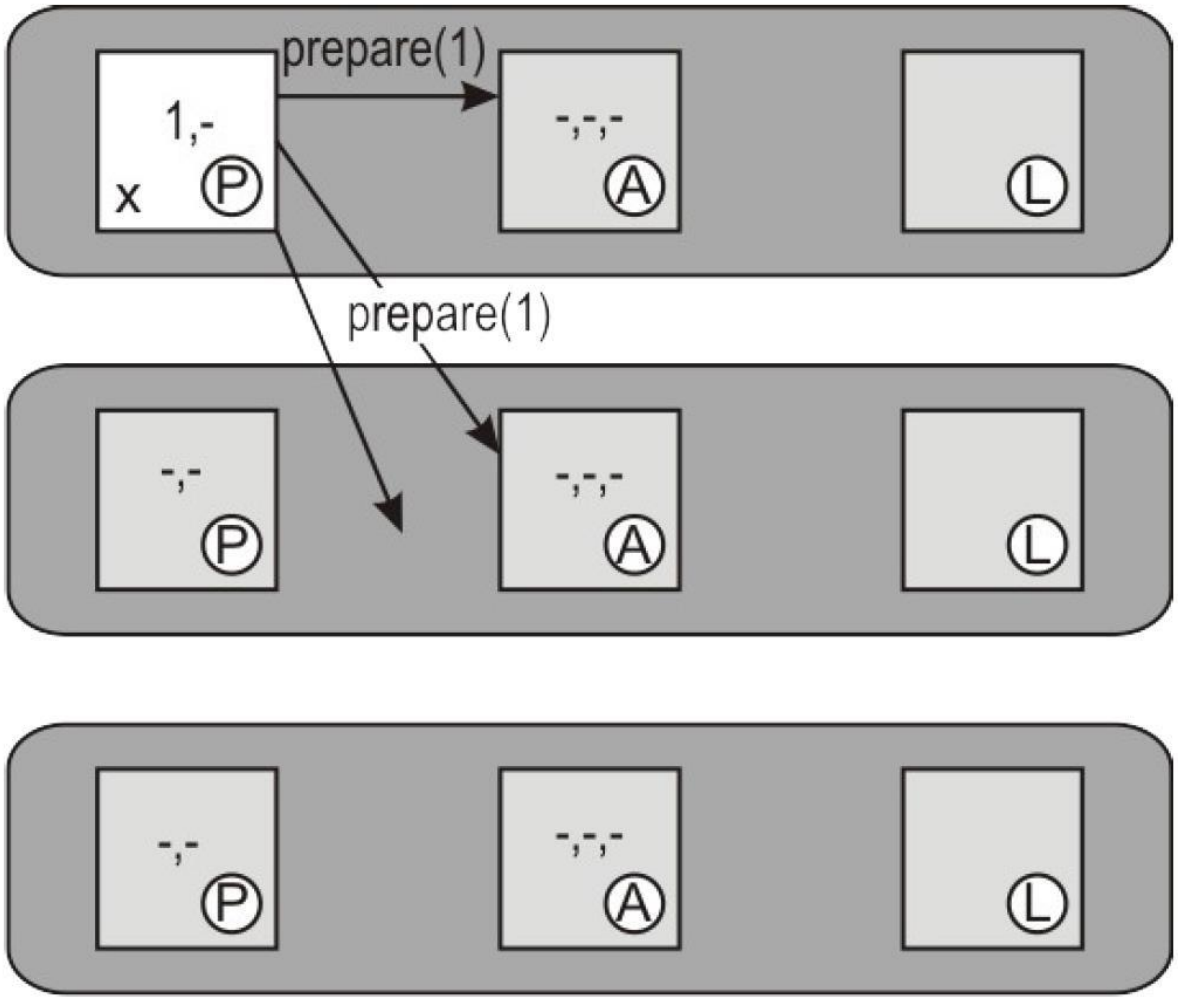
- In order to guarantee safety (also called "consistency"), Paxos defines three properties and ensures they are always held, regardless of the pattern of failures:

- Validity (or non-triviality)
  - Only proposed values can be chosen and learned

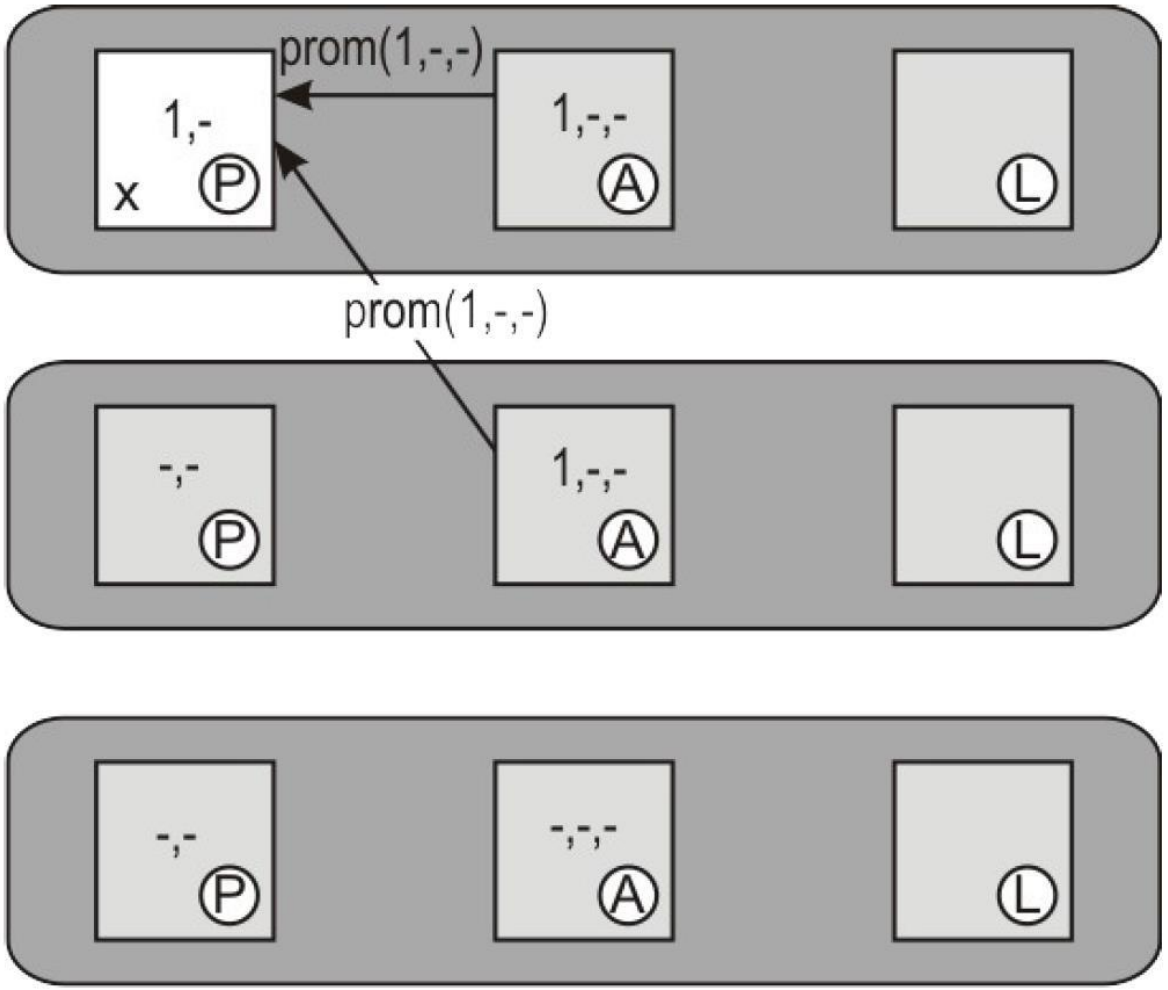
- Agreement (or consistency, or safety)
  - No two distinct learners can learn different values (or there can't be more than one decided value)

- Termination (or liveness)
  - If value C has been proposed, then eventually learner L will learn some value (if sufficient processors remain non-faulty).

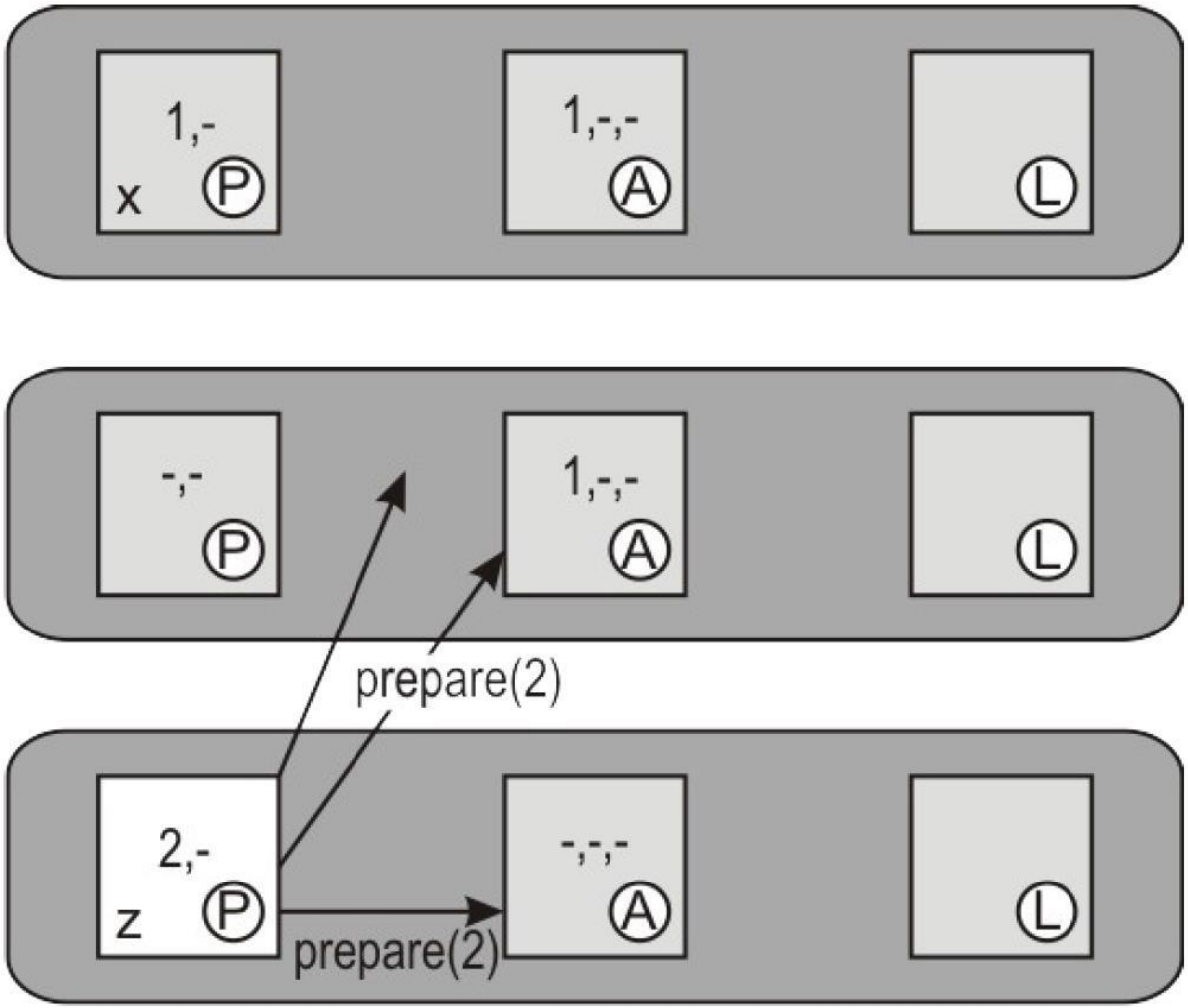
Paxos: Problematic case example



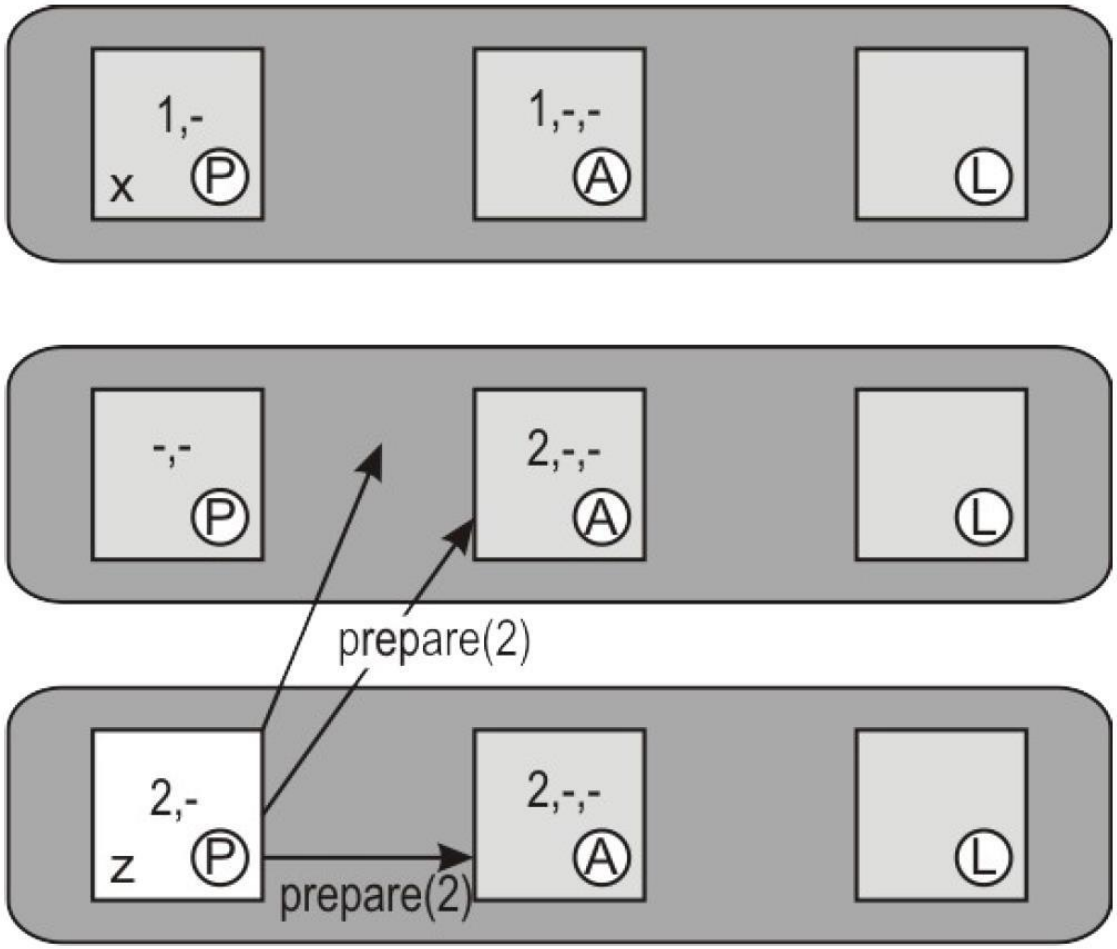
Paxos: Problematic case example



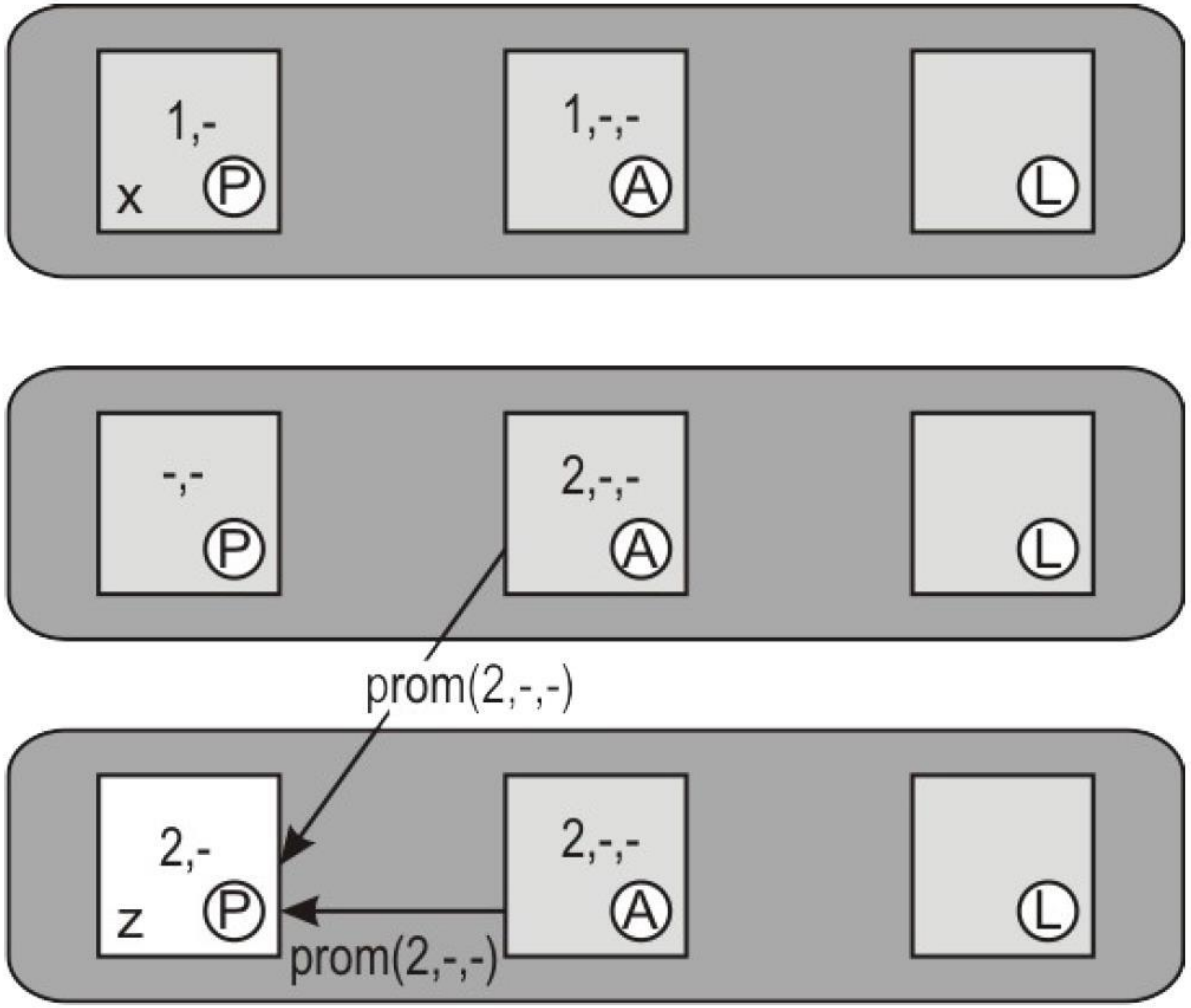
Paxos: Problematic case example



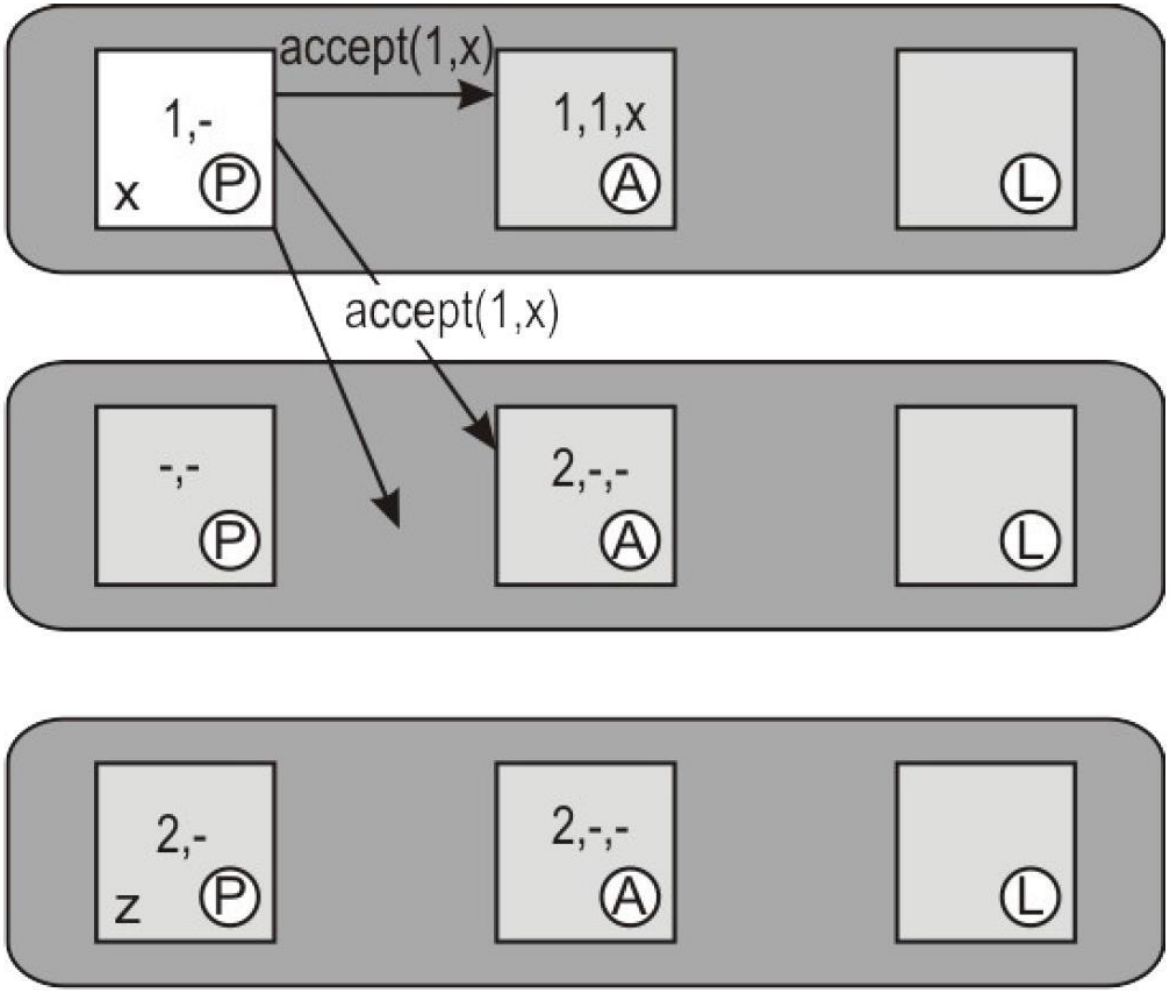
Paxos: Problematic case example



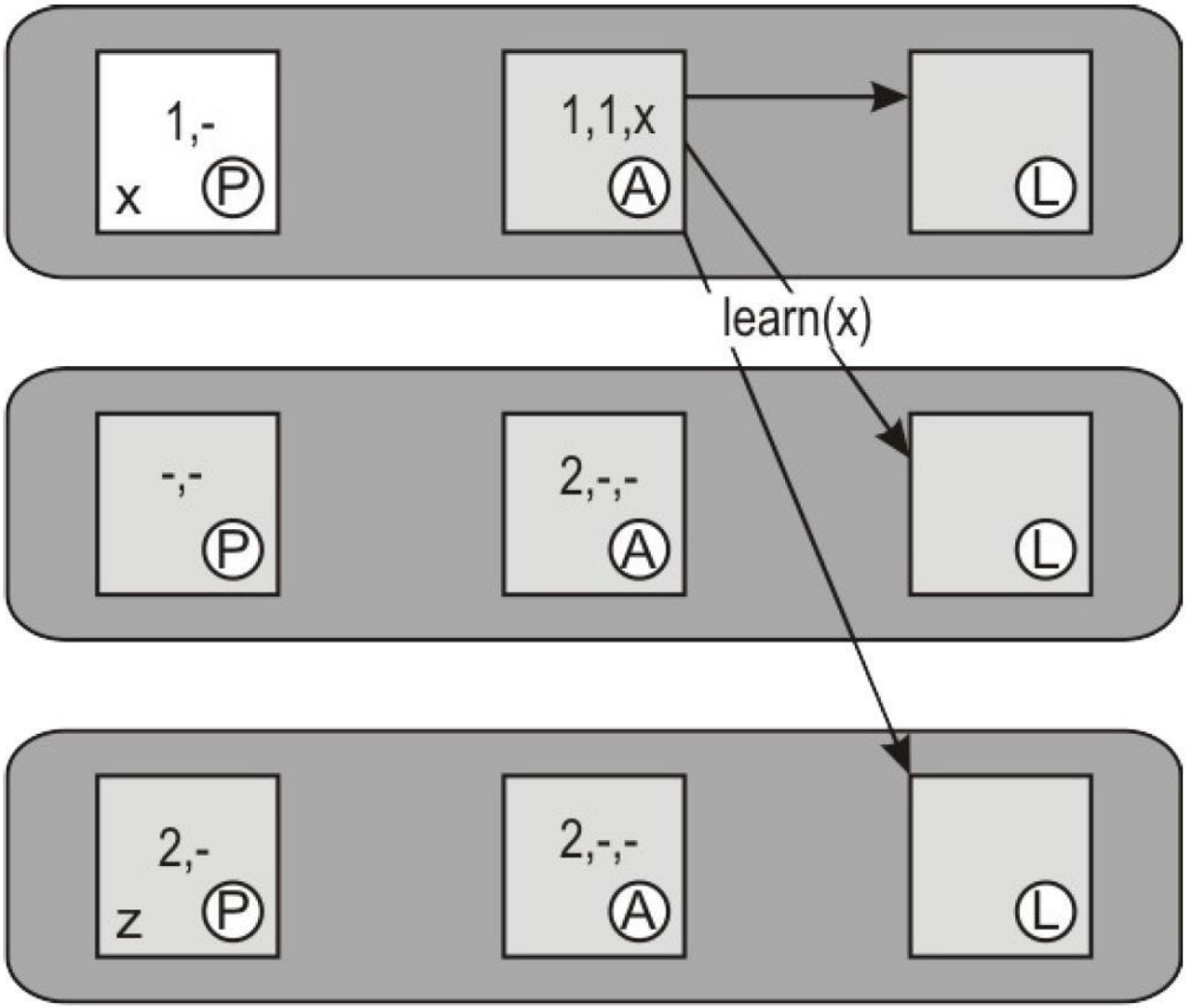
Paxos: Problematic case example



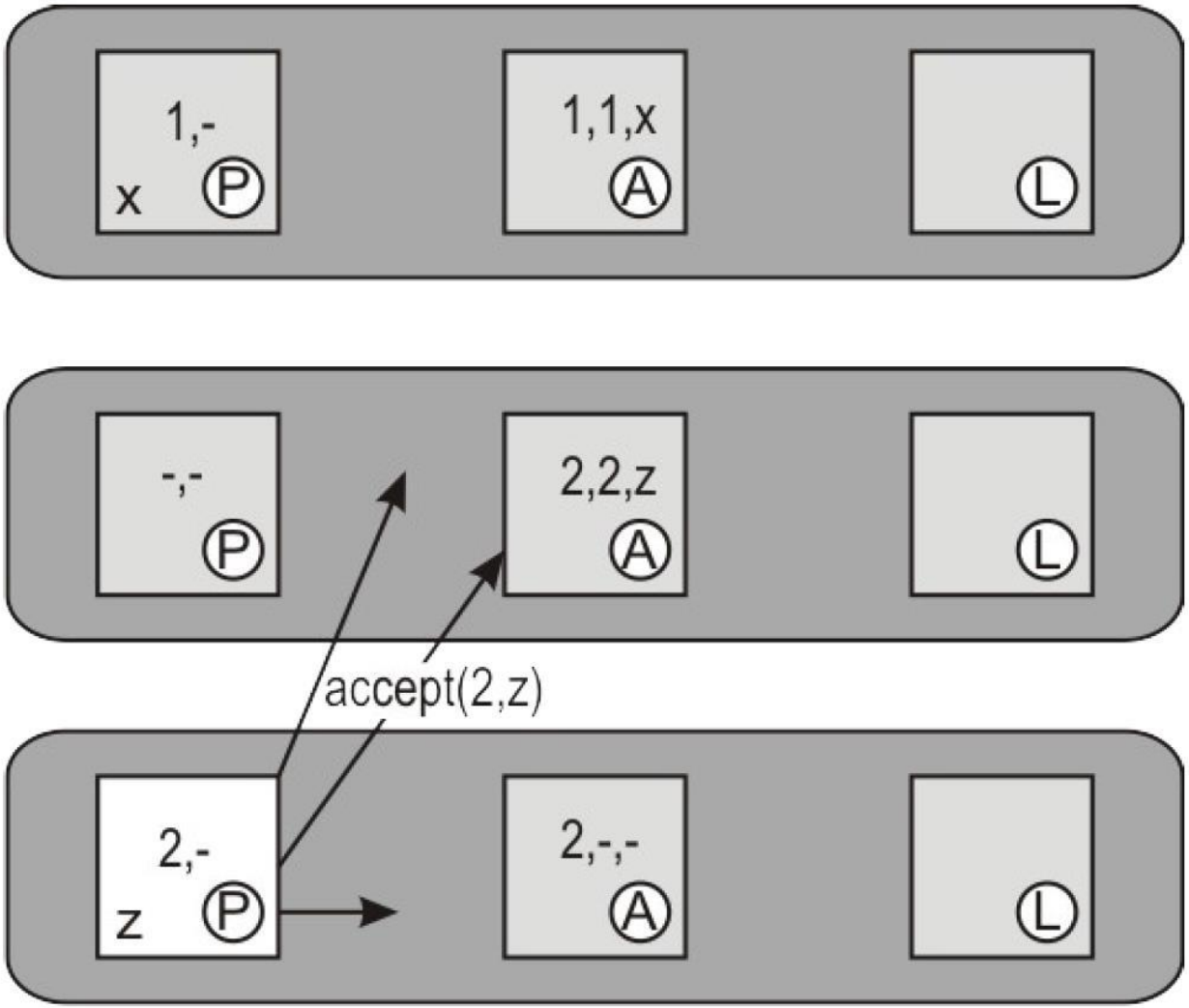
Paxos: Problematic case example



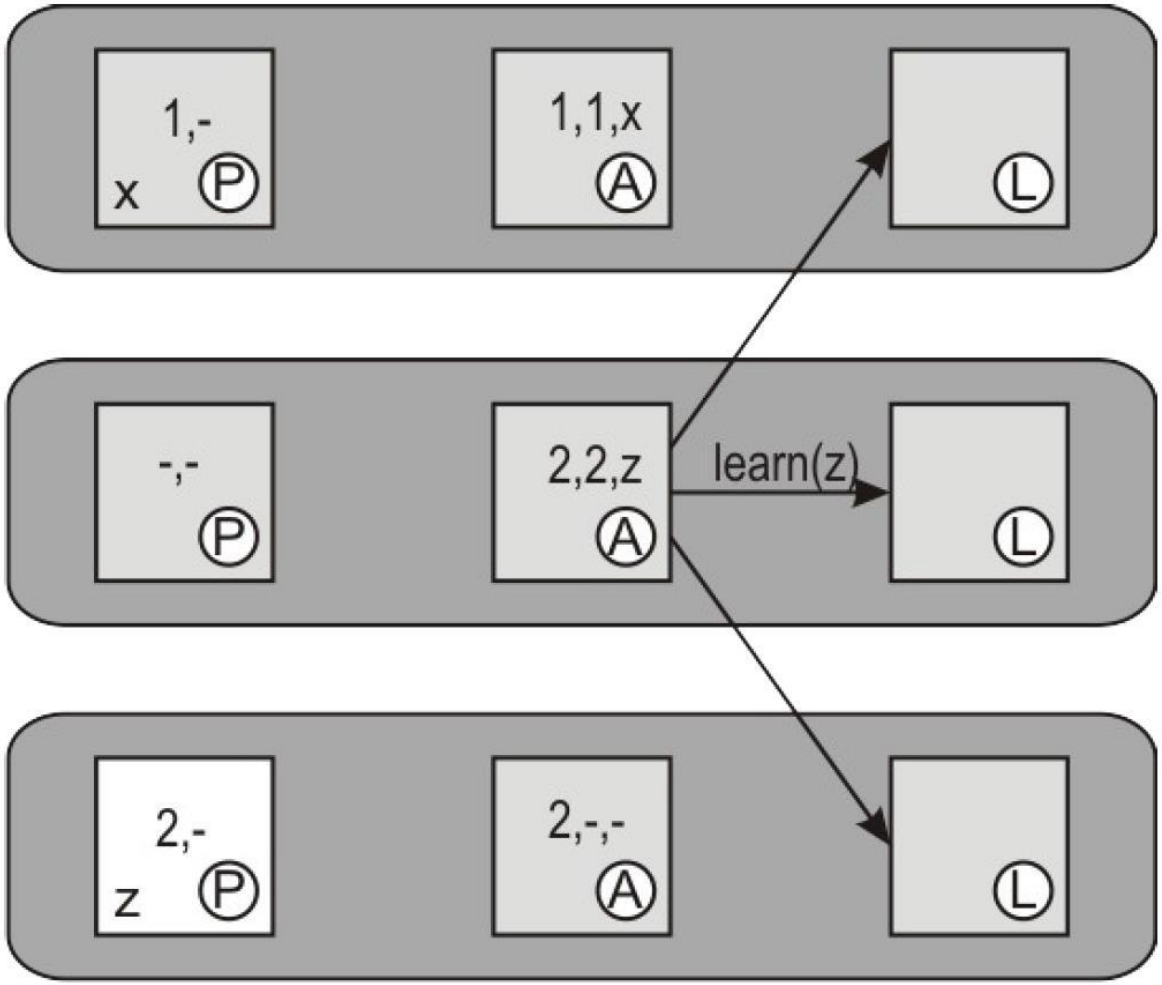
Paxos: Problematic case example



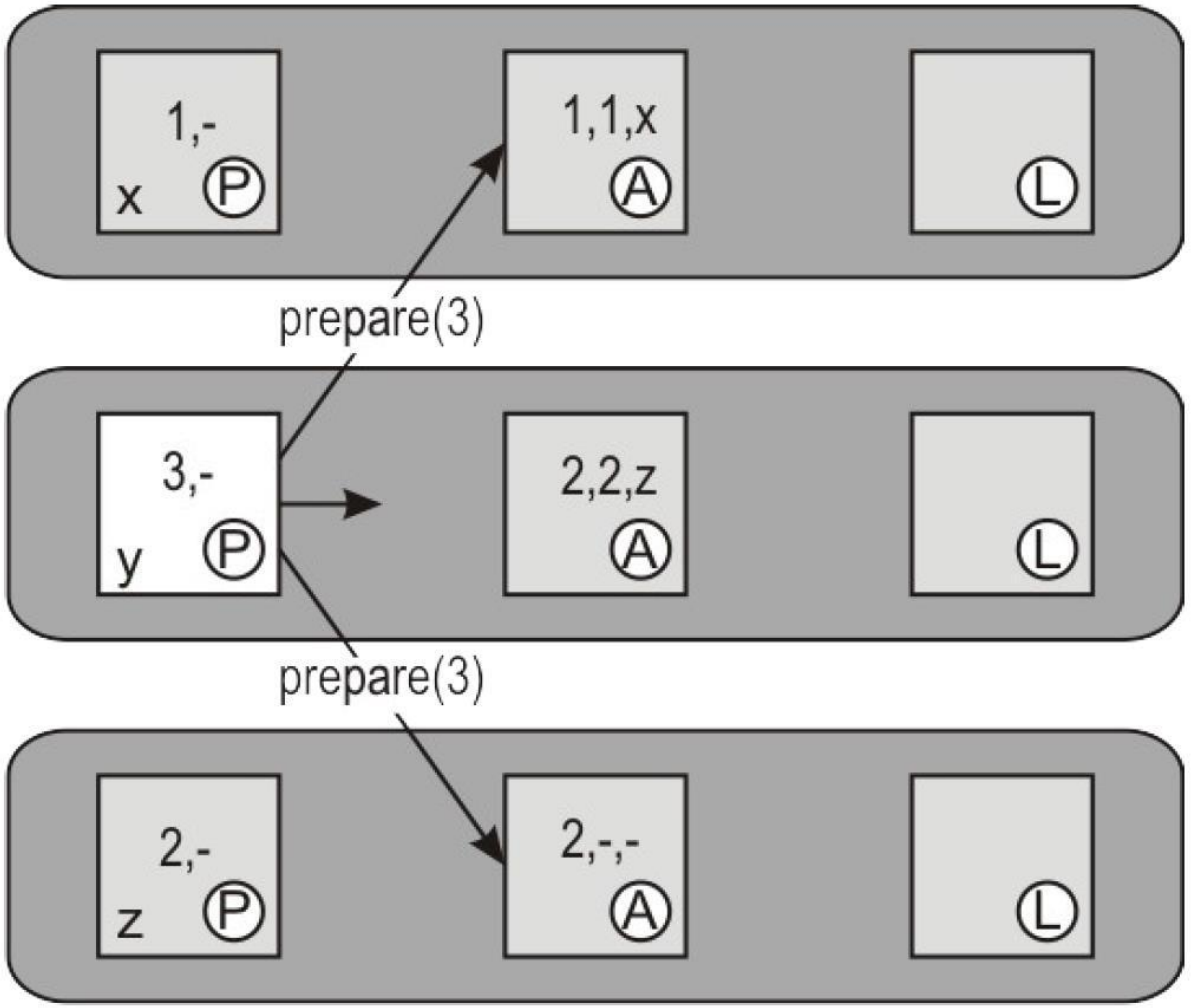
Paxos: Problematic case example



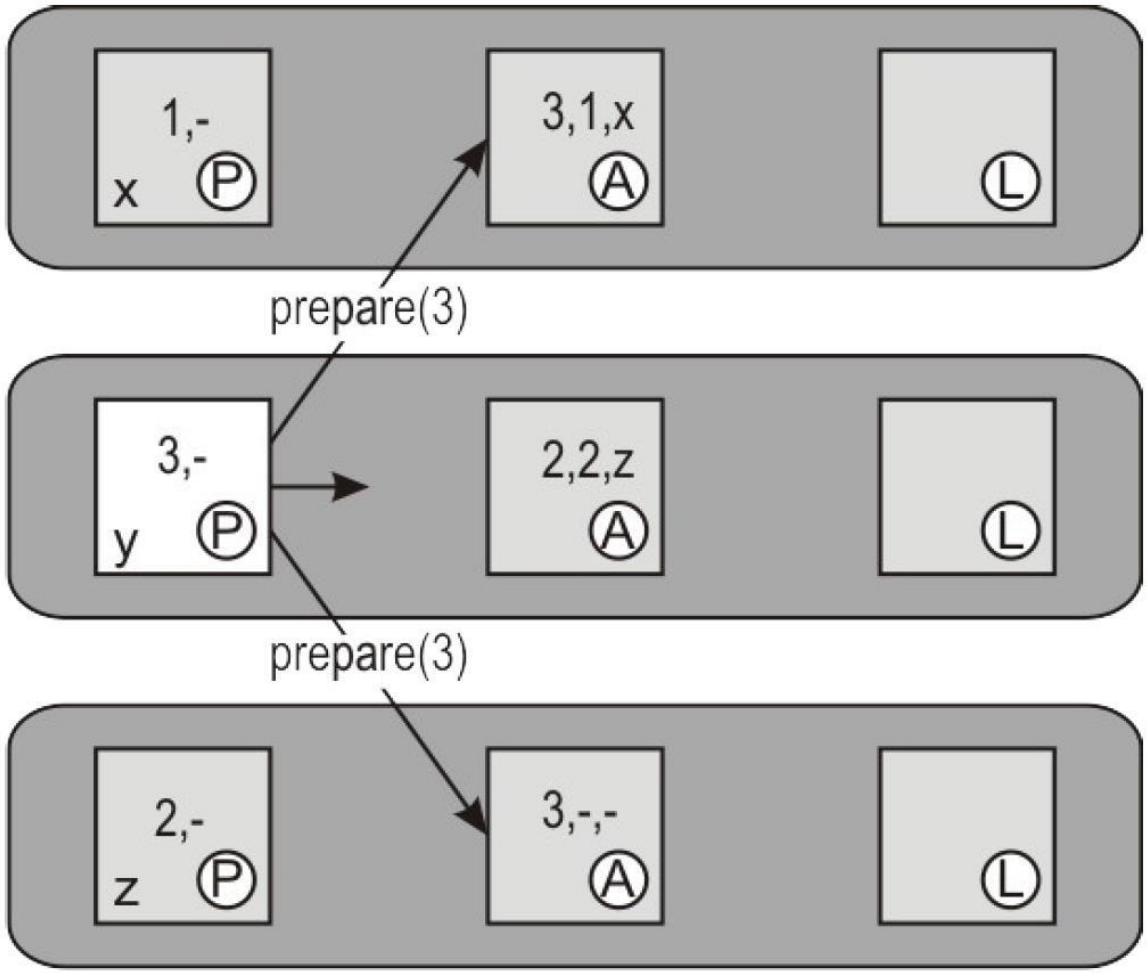
Paxos: Problematic case example



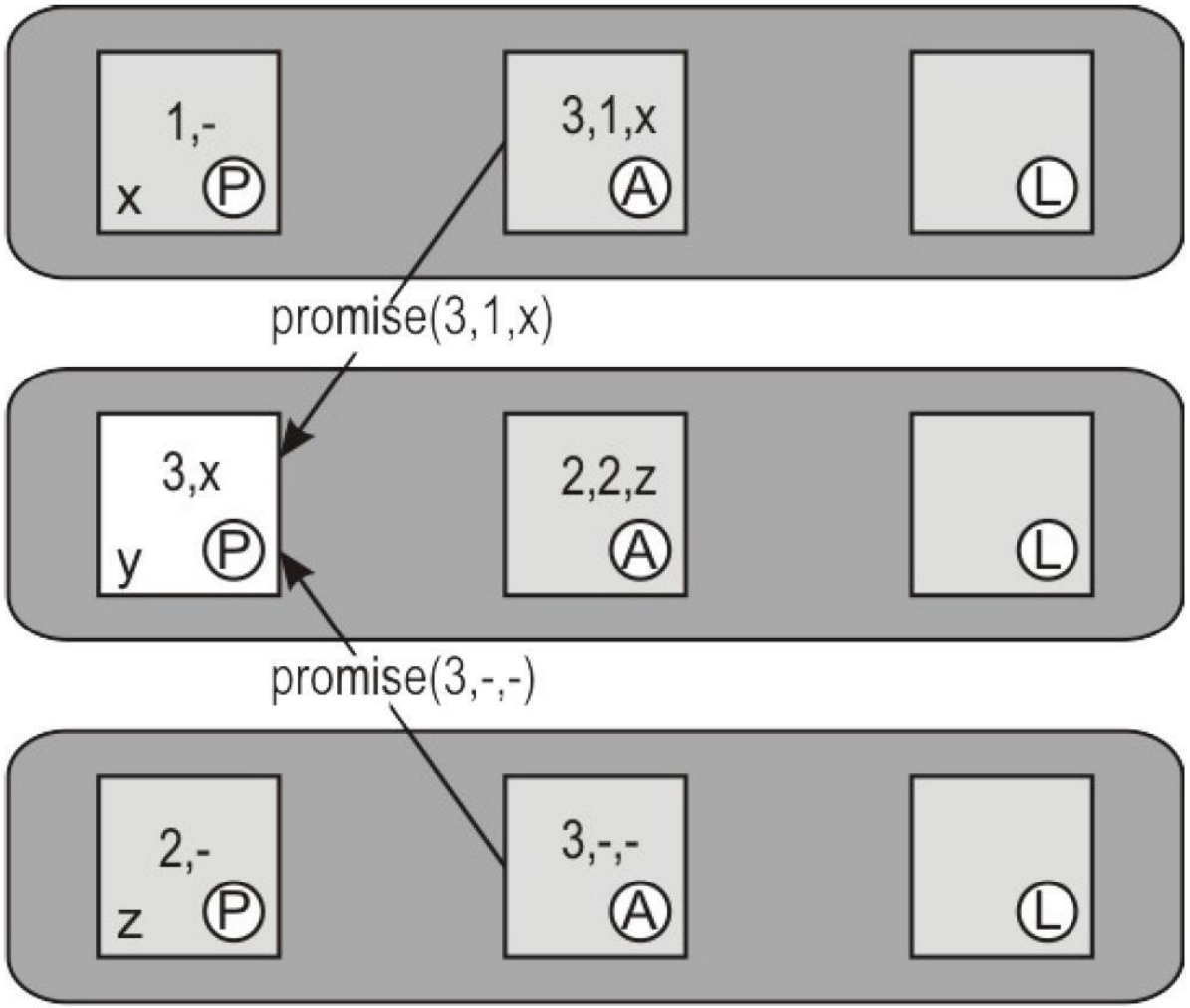
Paxos: Problematic case example



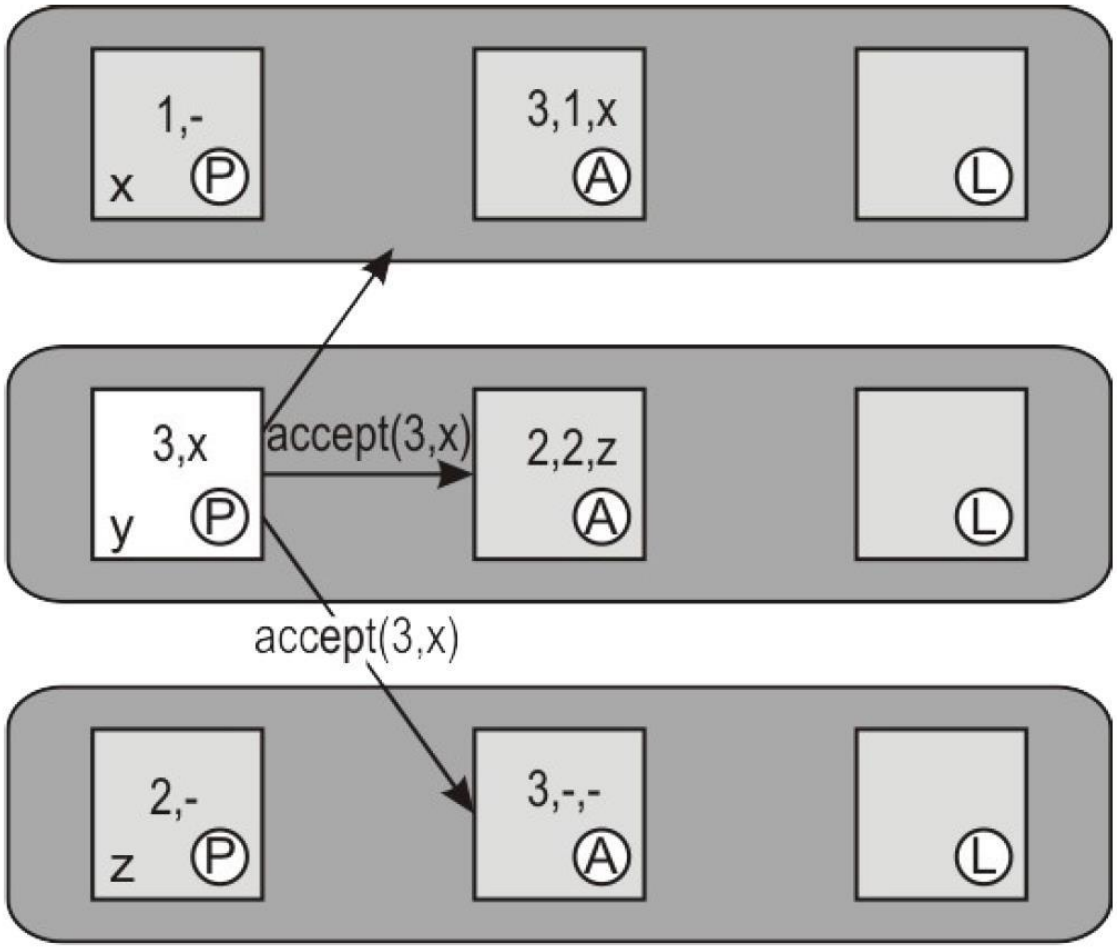
Paxos: Problematic case example



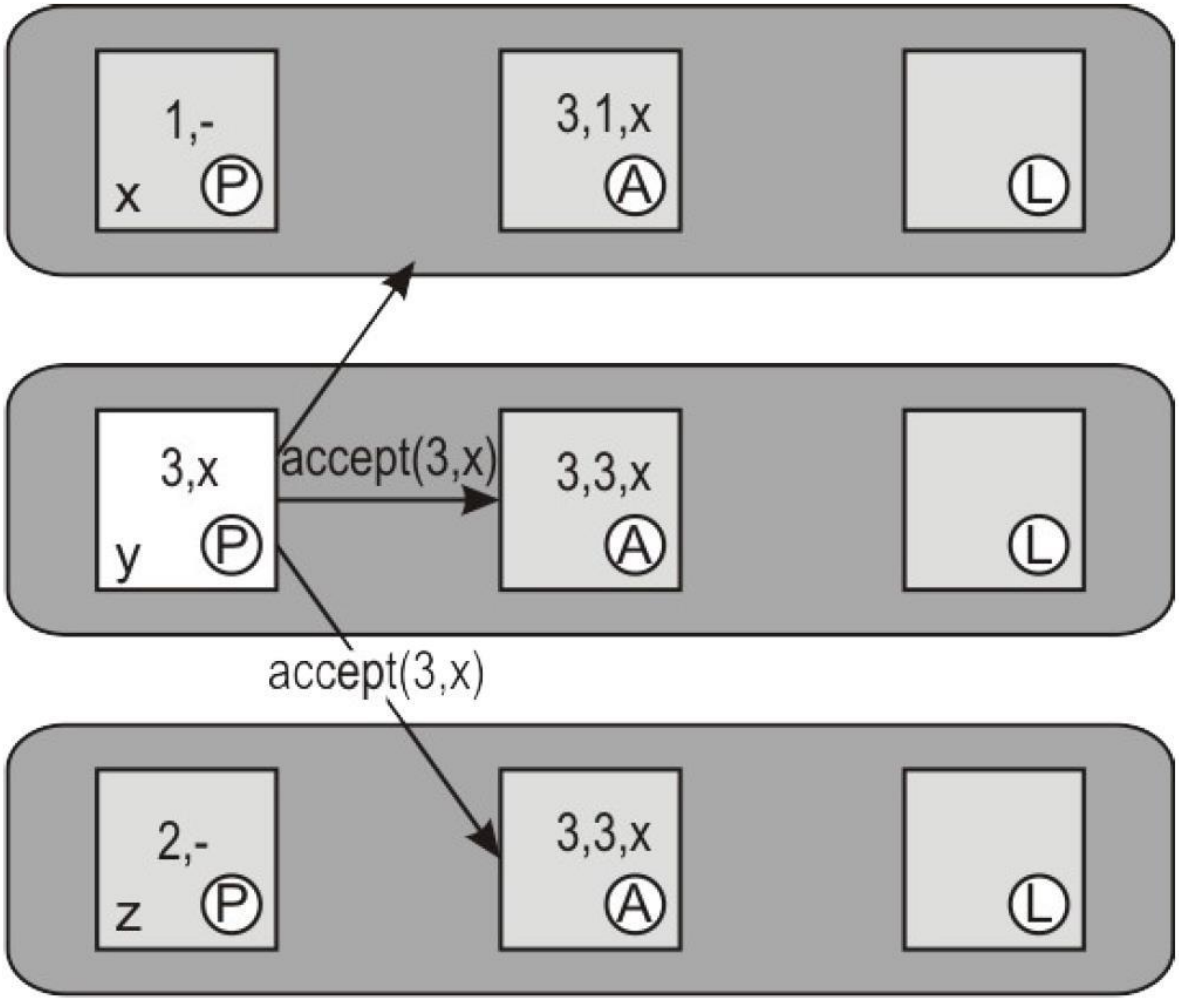
Paxos: Problematic case example



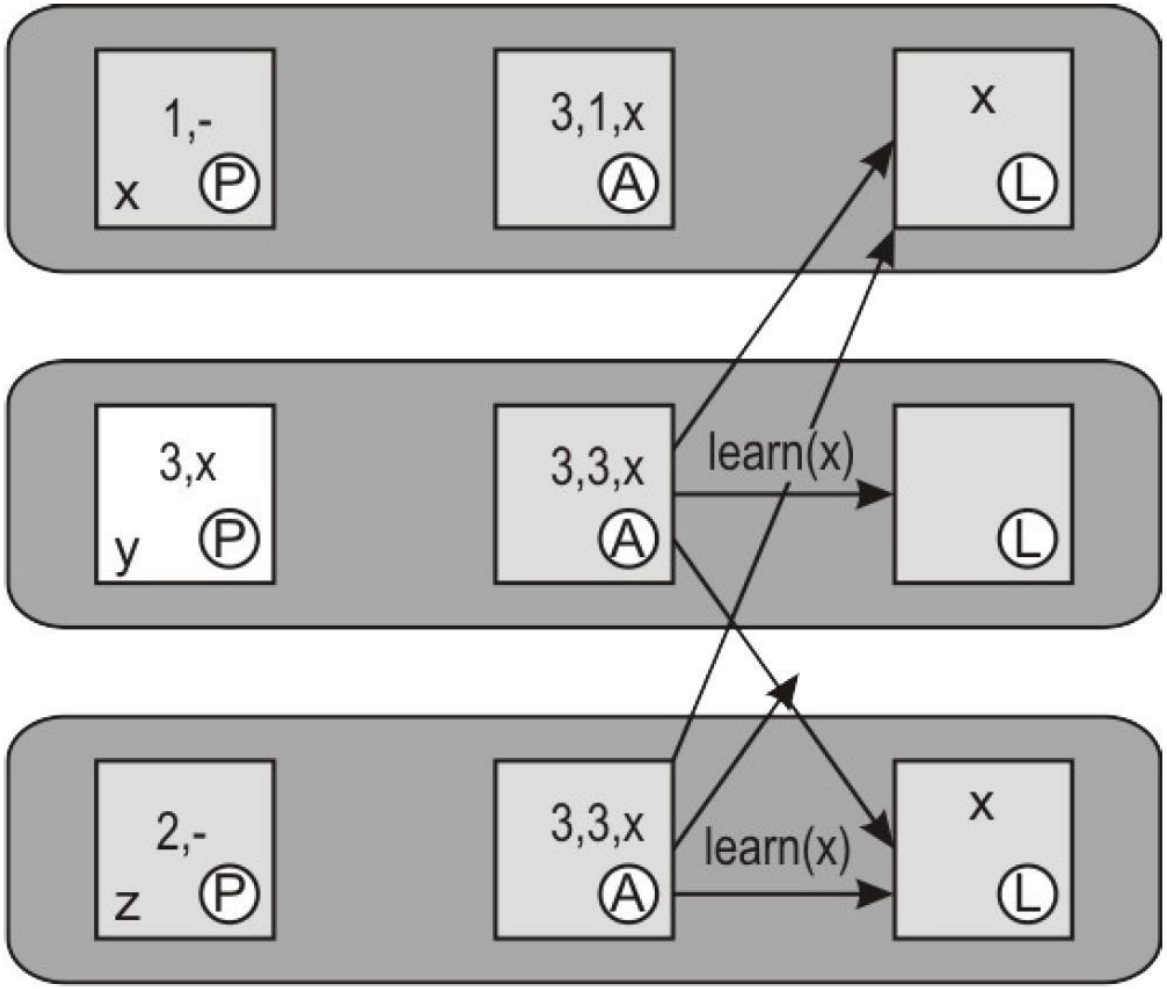
Paxos: Problematic case example



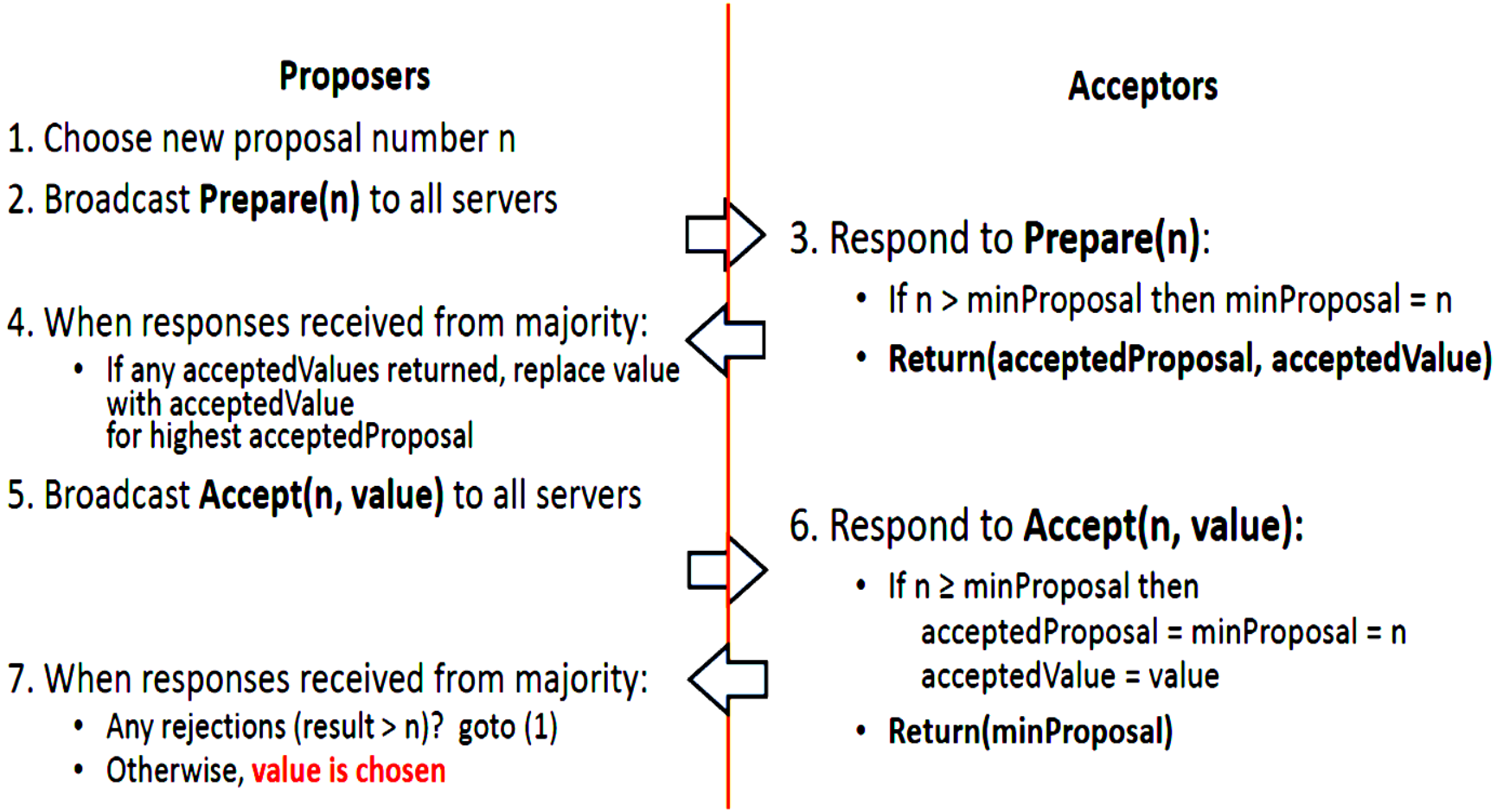
Paxos: Problematic case example



Paxos: Problematic case example



Basic Paxos lifecycle

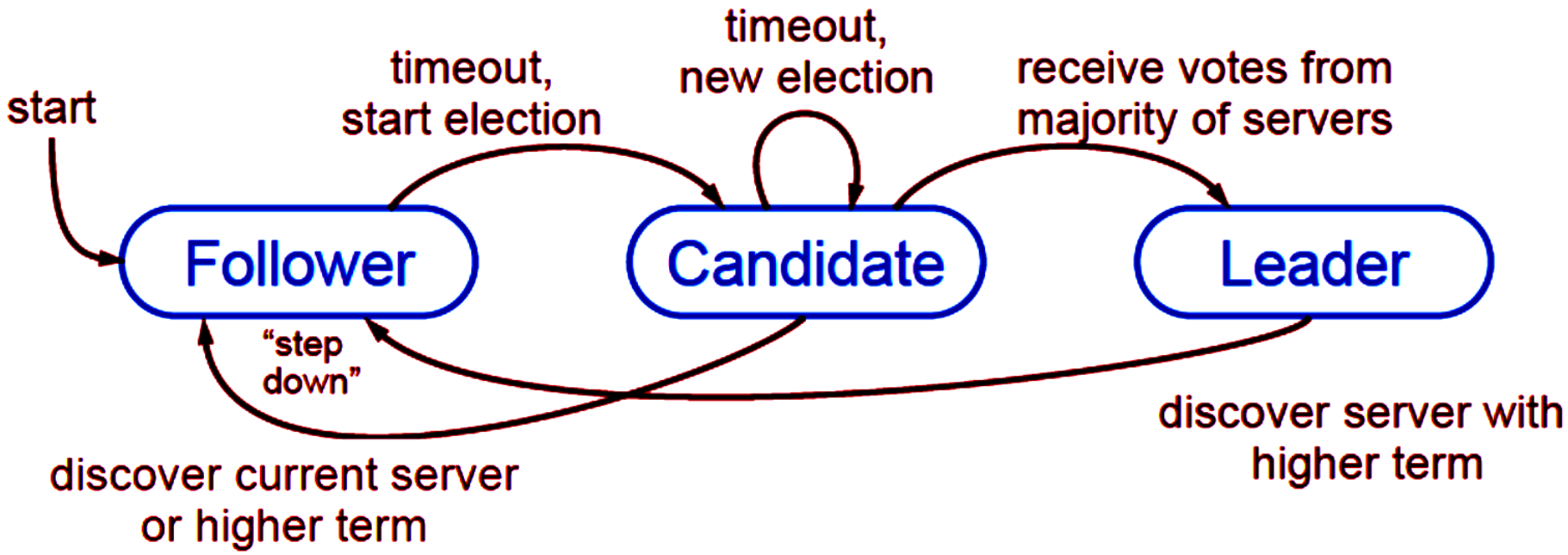


Acceptors must record minProposal, acceptedProposal, and acceptedValue on stable storage (disk)

Raft

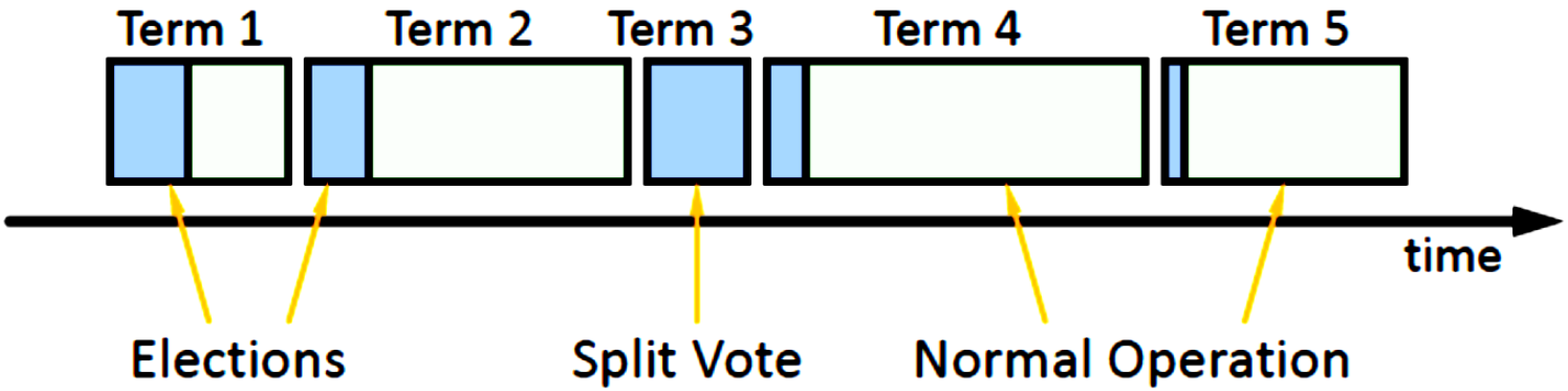
- Simplifies normal operation
- Raft server states:
- At any given time, each server is either:
  - Leader: handles all client interactions, log replication
    - At most 1 viable leader at a time
  - Follower: completely passive (issues no RPCs, responds to incoming RPCs)
  - Candidate: used to elect a new leader
  - Normal operation: 1 leader, N-1 followers

Raft: States


start
timeout,
start election
timeout,
new election
receive votes from
majority of servers
Follower
Candidate
Leader
"step
down"
discover server with
higher term
discover current server
or higher term

Raft: Terms

- Time divided into terms:
  - Election
  - Normal operation under a single leader
- At most one leader per term
- Some terms have no leader (failed election)
- Each server maintains the **current term** value
- Key role of terms: identify obsolete information



Heartbeat and Timeouts

- Servers start-up as followers
- Followers expect to receive RPCs from leaders or candidates
- Leaders must send **heartbeats** (empty Append Entries RPCs) to maintain authority
- If **election Timeout** elapses with no RPCs:
  - Follower assumes leader has crashed
  - Follower starts new election
  - Timeouts typically 100-500ms

Raft: Election basics

- Increment current term
- Change to a Candidate state
- Vote for self
- Send Request Vote via RPCs to all other servers, retry until either:
  - Receive votes from a majority of servers:
    - Become leader
    - Send Append Entries heartbeats to all other servers
  - Receive RPC from a valid leader:
    - Return to the follower state
  - No one wins election (election timeout elapses):
    - Increment term, start a new election

Raft: Election basics (cont.)

- **Safety**: allow at most one winner per term
  - Each server gives out only one vote per term (persist on disk)
  - Two different candidates can't accumulate majorities in the same term

- **Liveness**: some candidate must eventually win
  - Choose election timeouts randomly in [T, 2T]
  - One server usually times out and wins the election before others wake up

Agreement

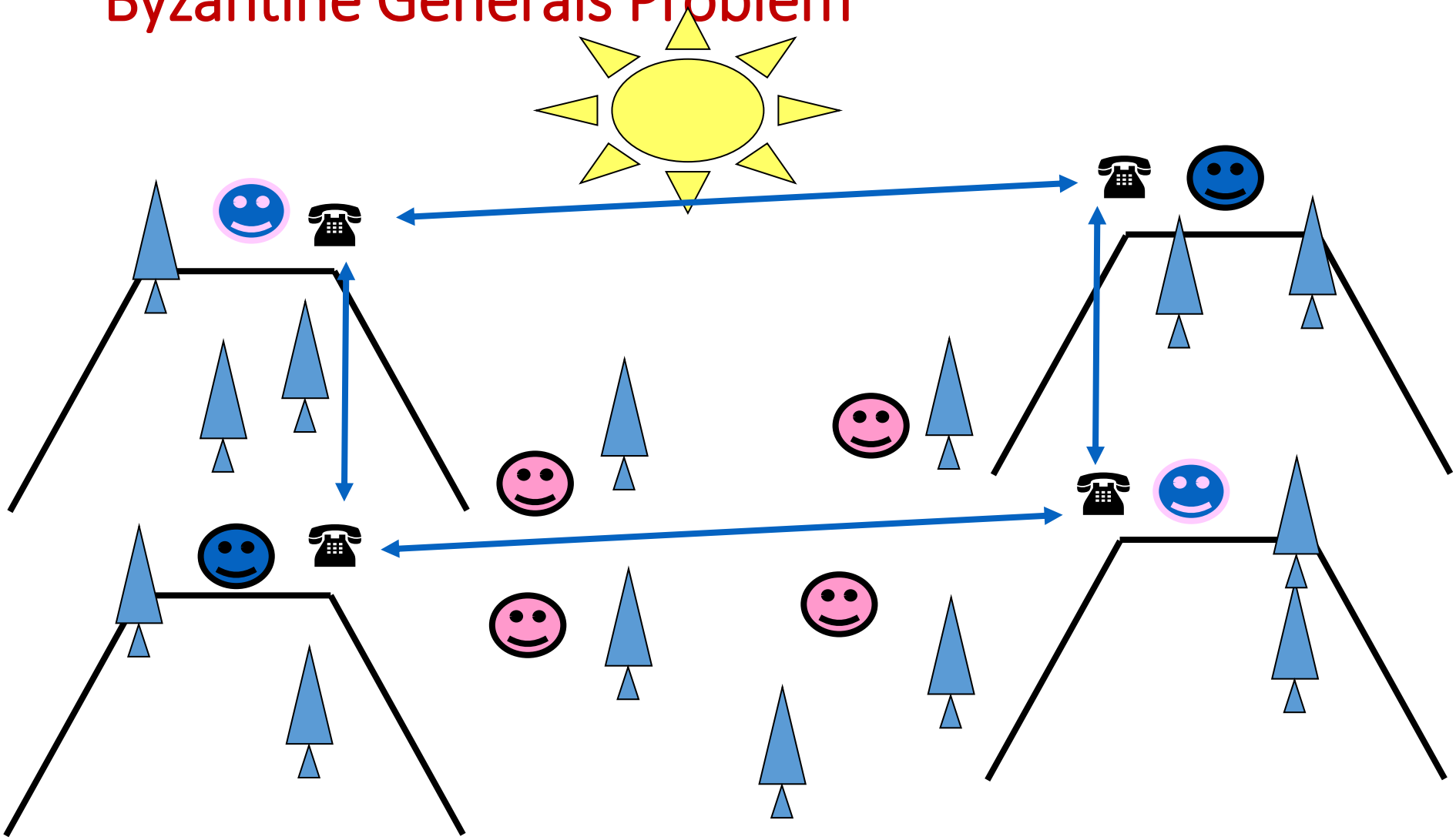
- Need an agreement in DS:
  - Leader, commit and synchronize

- **Distributed Agreement algorithm**: all non-faulty processes achieve consensus in a finite number of steps

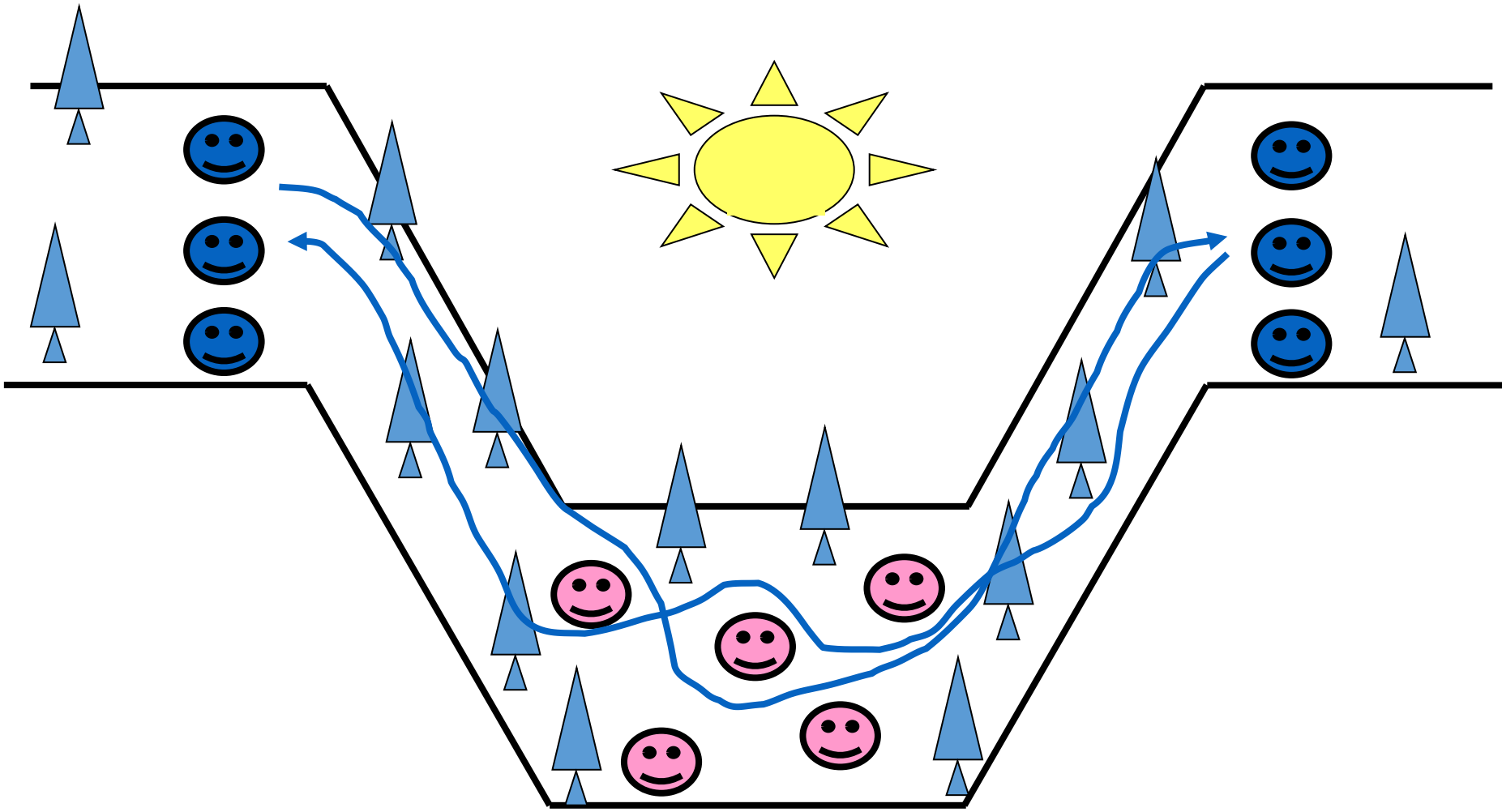
- Faulty processes, perfect channels: Byzantine generals

- Perfect processes, faulty channels: two-army

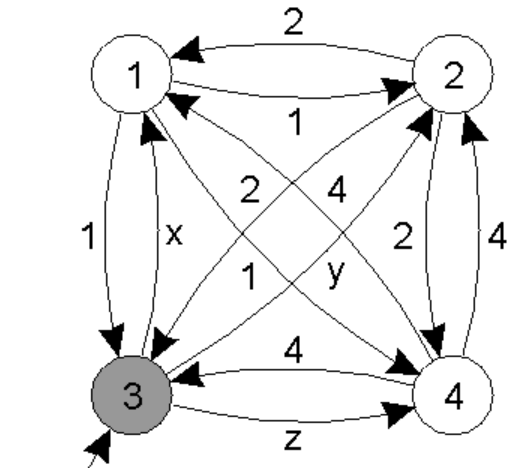
Byzantine Generals Problem



Two-Army Problem



Byzantine Generals -Example



(a)

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

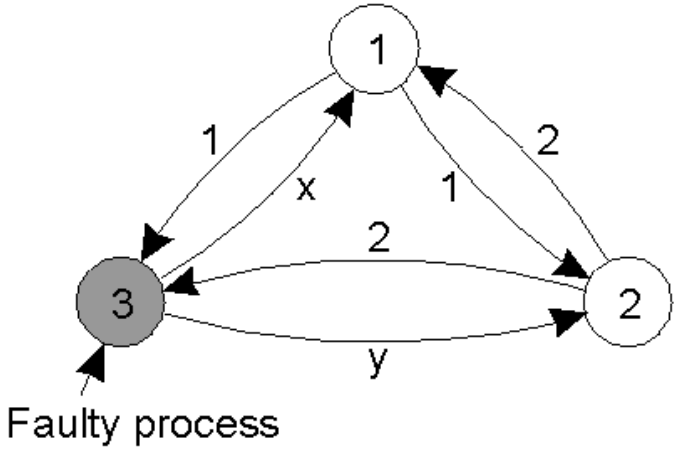
The Byzantine general's problem for 3 loyal generals and1 traitor.

a) The generals announce the time to launch the attack (by messages marked by their ids).

b) The vectors that each general assembles based on (a)

c) The vectors that each general receives in step, where every general passes his vector from (b) to every other general.

Byzantine Generals –Example



(a)

1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

(b)

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

The same as in previous slide, except now with 2 loyal generals and one traitor.

Byzantine Generals

- Given three processes, if one fails, consensus is impossible
- Given N processes, if F processes fail, consensus is impossible if $N \leq 3F$

Distributed Commit

- **Goal**: Either **all** members of a group decide to perform an operation, or **none** of them perform the operation

- **Atomic transaction**: a transaction that happens completely or not at all

- **Failures:**
  - Crash failures that can be recovered
  - Communication failures detectable by timeouts

- Notes:
  - Commit requires a set of processes to agree…
  - similar to the Byzantine generals problem…
    - but the solution much simpler because stronger assumptions

Two-phase commit (2PC)

- The client who initiated the computation acts as coordinator; processes required to commit are the participants

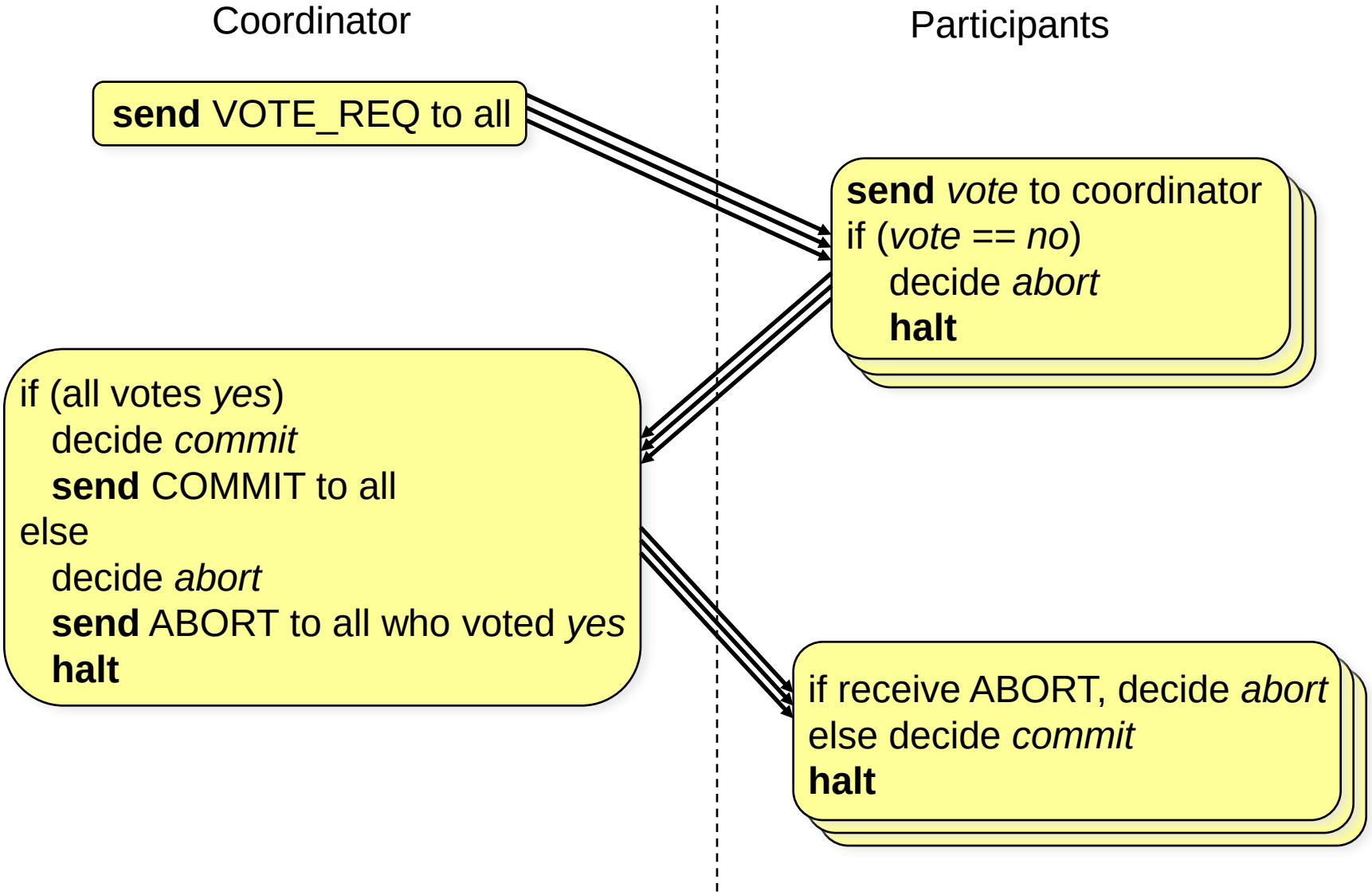
- **Phase 1a**: The coordinator sends vote-request to participants (also called a pre-write)

- **Phase 1b**: When the participant receives a vote request it returns either vote-commit or vote-abort to the coordinator. If it sends vote-abort, it aborts its local computation

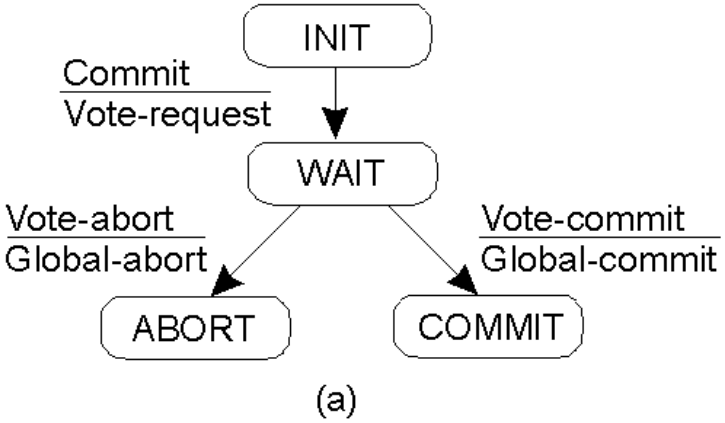
- **Phase 2a**: The coordinator collects all votes; if all are vote-commit, it sends global-commit to all participants, otherwise it sends global-abort

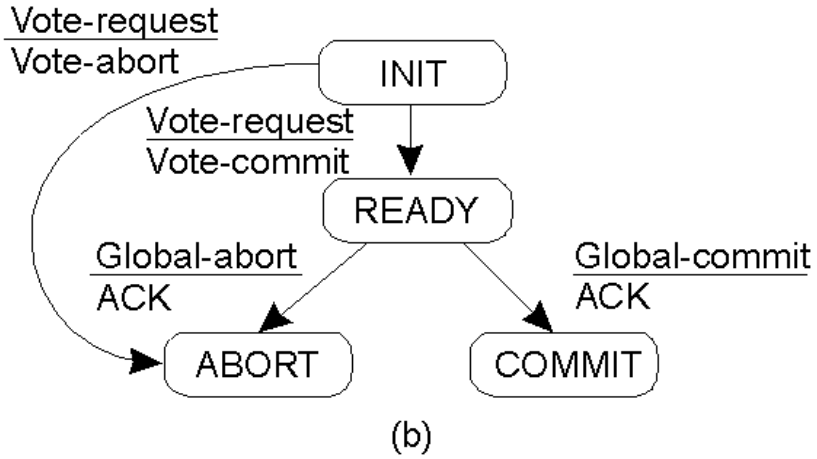
- **Phase 2b**: Each participant waits for global commit or global abort and handles accordingly.

Two Phase Commit (2PC)



Two-Phase Commit





a) The finite state machine for the coordinator in 2PC.

b) The finite state machine for a participant.

Two-phase commit-Failing participant

- Participant crashes in state S, and recovers to S'
  - **Initial state:** No problem: a participant was unaware of the protocol
  - **Ready state:** Participant is waiting to either commit or abort. After recovery, the participant needs to know which state transition he should make (log the coordinator's decision)
  - **Abort state:** Merely make entry into abort state idempotent, e.g., removing the workspace of results
  - **Commit state:** Also make entry into commit state idempotent, e.g., copying workspace to storage.

Two-Phase Commit

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

- When recovery is needed to READY state, check the state of other participants -> no need to log the coordinator's decision.

- Recovering participant P contacts another participant Q.

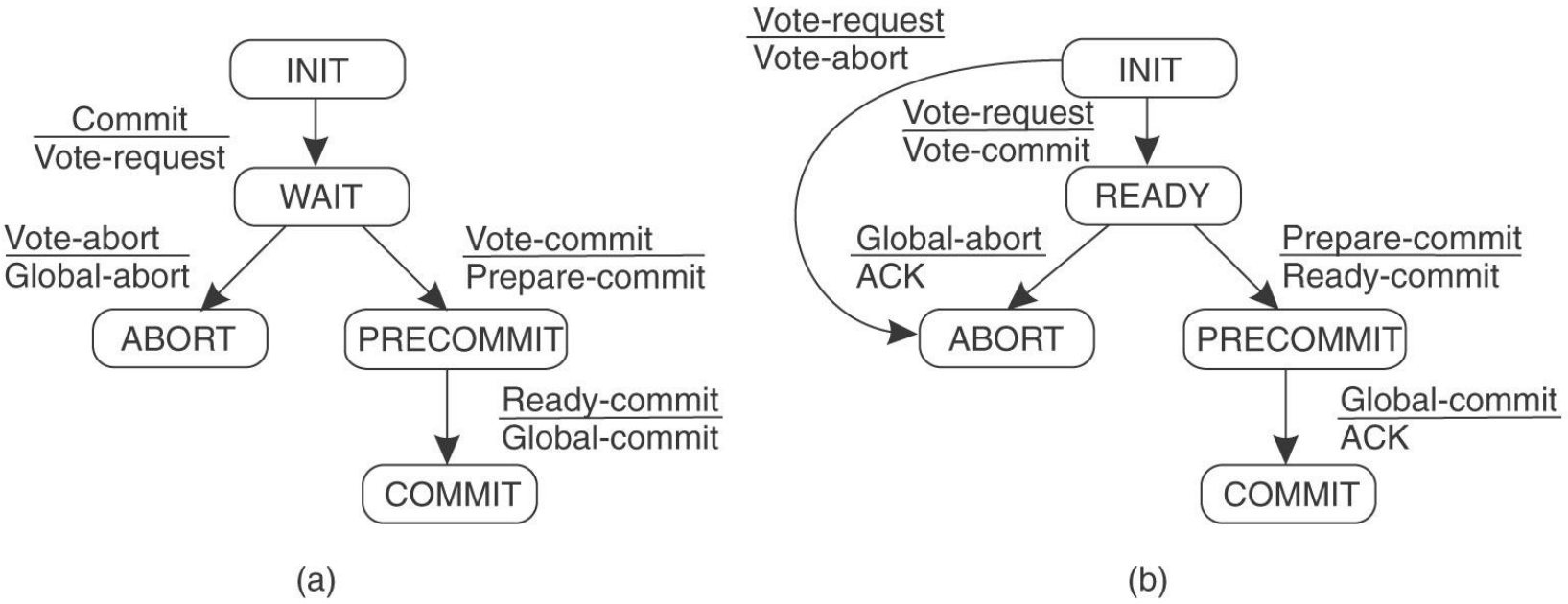
Two-Phase Commit

- When all participants are in the ready states, no final decision can be reached
- Apparently, the coordinator is failing
- Two-phase commit is a blocking commit protocol

Two-phase commit-Failing coordinator

- The real problem lies in the fact that the coordinator's final decision may not be available for some time (or actually lost).
- Let a participant P in the READY state timeout when it hasn't received the coordinator's decision; P tries to find out what other participants know (as discussed).
- Essence of the problem is that a recovering participant cannot make a local decision: it is dependent on other (possibly failed) processes

Three-Phase Commit



- non-blocking commit protocol

Three-Phase Commit

- Coordinator sends *Vote_Request* (as before)
- If all participants respond affirmatively,
  - Put *Precommit* state into the log on stable storage
  - Send out *Prepare_to_Commit* message to all
- After all participants acknowledge,
  - Put *Commit* state in log
  - Send out *Global_Commit*
- Coordinator blocked in *Wait* state
  - Safe to abort transaction
- Coordinator blocked in *Precommit* state
  - Safe to issue *Global_Commit*
  - Any crashed or partitioned participants will commit when recovered

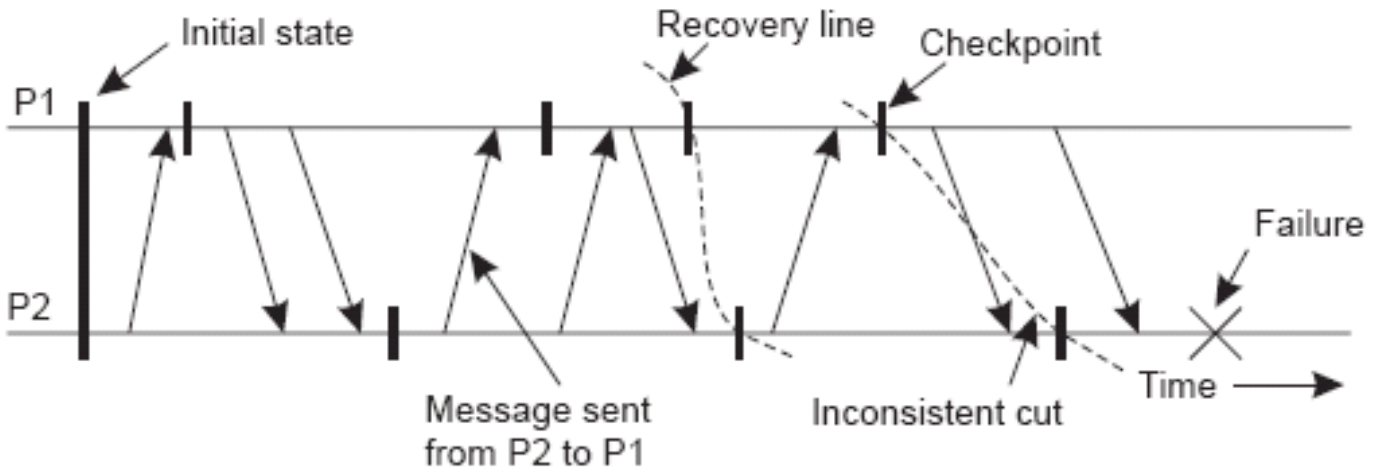
Recovery

A process that exhibits a failure must be able to recover to a correct state

Checkpointing

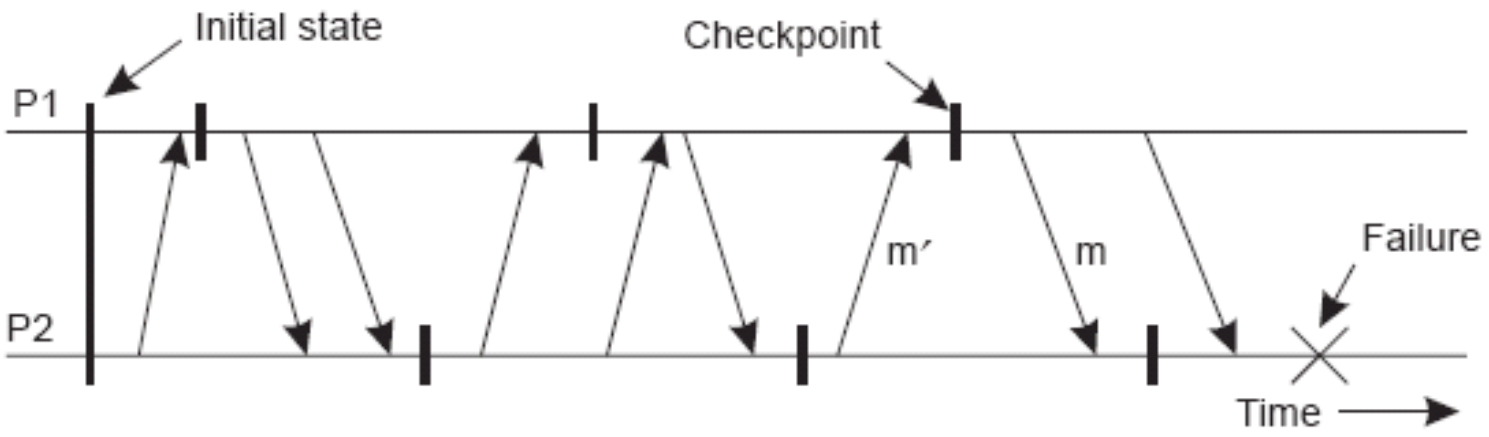
- Every message that has been received is also shown to have been sent in the state of the sender.

- Assuming processes regularly checkpoint their state, the most recent **consistent global checkpoint is** called the **recovery line**



Independent Checkpointing

- It is often difficult to find a recovery line in a system where every process just records its local state every so often –
  - a *domino effect* or cascading rollback can result



Independent Checkpointing

- Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.
- Let CP[i](m) denote m-th checkpoint of process Pi and INT[i](m) the interval between CP[i](m-1) and CP[i](m)
- When process Pi sends a message in interval INT[i](m), it piggybacks (i;m)
- When process Pj receives a message in interval INT[j](n), it records the dependency INT[i](m)!INT[j](n)
- The dependency INT[i](m)!INT[j](n) is saved to stable storage when taking checkpoint CP[j](n)

Coordinated Checkpointing

- To solve this problem, systems can implement *coordinated checkpointing*

- Each process takes a checkpoint after a globally coordinated action

Coordinated Checkpointing

- Make sure that processes are synchronized when doing the checkpoint

- Two-phase blocking protocol
  1. Coordinator multicasts *CHECKPOINT_REQUEST*
  2. Processes take a local checkpoint
     - Delay further sends
     - Acknowledge to the coordinator
     - Send state
  3. Coordinator multicasts CHECKPOINT_DONE

References

- Distributed systems: principles and paradigms, Andrew S. Tanenbaum, Maarten Van Steen
- Paxos presentation by Hein Meling, Associate professor at University Stavanger
- Paxos and Raft study by Diego Ongaroand John Ousterhout, Stanford University 2013
- http://thesecretlivesofdata.com/raft/