Diskless Stable Storage

Ellis Michael

Administrivia

- Lab 4 home stretch!
- Problem Set 3 due Wednesday.
- Fill out course evaluation!

This Week

- New research!
- Work being done at UW by Arvind and me!
  - Diskless stable storage (Today)
  - Datacenter networking and Speculative Paxos (Wednesday)
  - Programmable networks and NOPaxos and Eris (Friday)

We've Been Living in a Fantasy World

For a long time, we've assumed that no more than f servers fail (usually $f<n/2$ or f<n/3 for Byzantine failures).

We've Been Living in a Fantasy World

For a long time, we've assumed that no more than f servers fail (usually $f<n/2$ or $f<n/3$ for Byzantine failures).

We can't assume that some servers never fail. We need a way to recover from failures.

We've Been Living in a Fantasy World

For a long time, we've assumed that no more than

f servers … yzantine

failures).

VALAR MORGHULIS

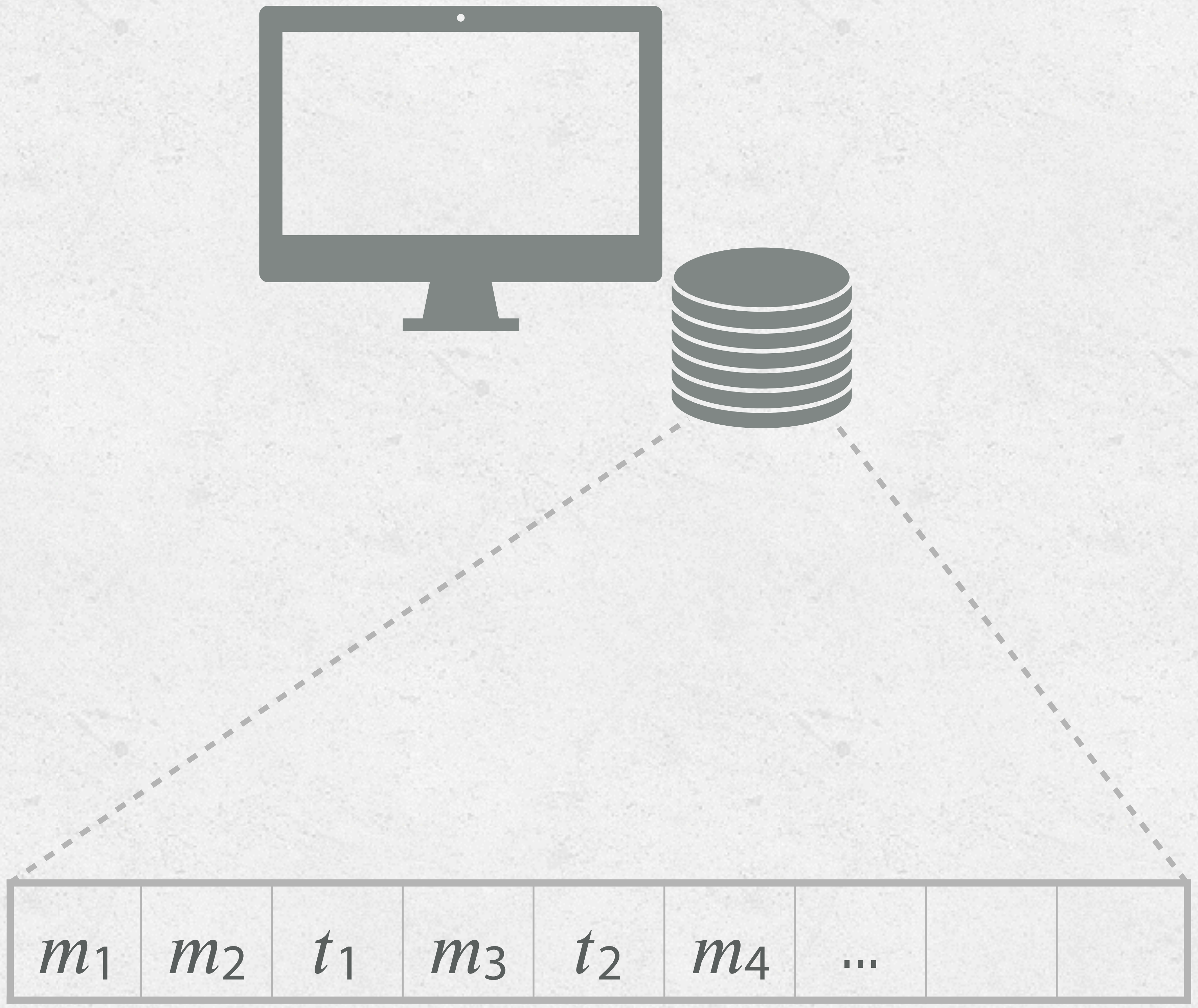
We can't … fail. We

need a way to recover from failures.

What if We Have Stable Storage?

Simple method:

- Each node has an attached disk.
- Node synchronously writes message (or timer) to disk before handling it.
- After node fails, upon restart replay log to restore server to previous state since handlers are deterministic.
- Can periodically write full state to disk to clear event log and free space.
- This method works for *any* distributed protocol.



Problem #1: Synchronous writes to disk are slow (really slow) (yes, even if you use an SSD)

Not Everything Needs to Be Saved

Take PMMC as an example.

Not Everything Needs to Be Saved

Take PMMC as an example.

We can skip logging the P1B and P2B messages. Recovering leaders read from a quorum to discover any previously used ballot number, use a higher one in the future. Everything else on the leader is **soft state** that can be safely discarded.

Not Everything Needs to Be Saved

Take PMMC as an example.

We can skip logging the P1B and P2B messages. Recovering leaders read from a quorum to discover any previously used ballot number, use a higher one in the future. Everything else on the leader is **soft state** that can be safely discarded.

We can even skip logging everything except P1A messages. Then, recovering acceptors must either recover all proposals from the latest leader or wait for a new leader to take over.

Replica State

- Should save state to disk before garbage collection.
- Everything else can be rebuilt by re-learning accepted values.

Replica State

- Should save state to disk before garbage collection.
- Everything else can be rebuilt by re-learning accepted values.

Leader State

- Only invariant a leader needs stable storage to enforce is avoiding using the same ballot with different values.
- Solution: learn about any ballot used in a P2A by querying P1As stored at acceptors, use a ballot larger than max seen.

Replica State

- Should save state to disk before garbage collection.
- Everything else can be rebuilt by re-learning accepted values.

Leader State

- Only invariant a leader needs stable storage to enforce is avoiding using the same ballot with different values.
- Solution: learn about any ballot used in a P2A by querying P1As stored at acceptors, use a ballot larger than max seen.

Acceptor State

- Write P1A messages synchronously to disk.
- Need to learn about accepted values.
- Solution: learn about missing P2As from the recent leader (or a future one which reads from a quorum not including this node).

Replica State

- Should save state to disk before garbage collection.
- Everything else can be rebuilt by re-learning accepted values.

Leader State

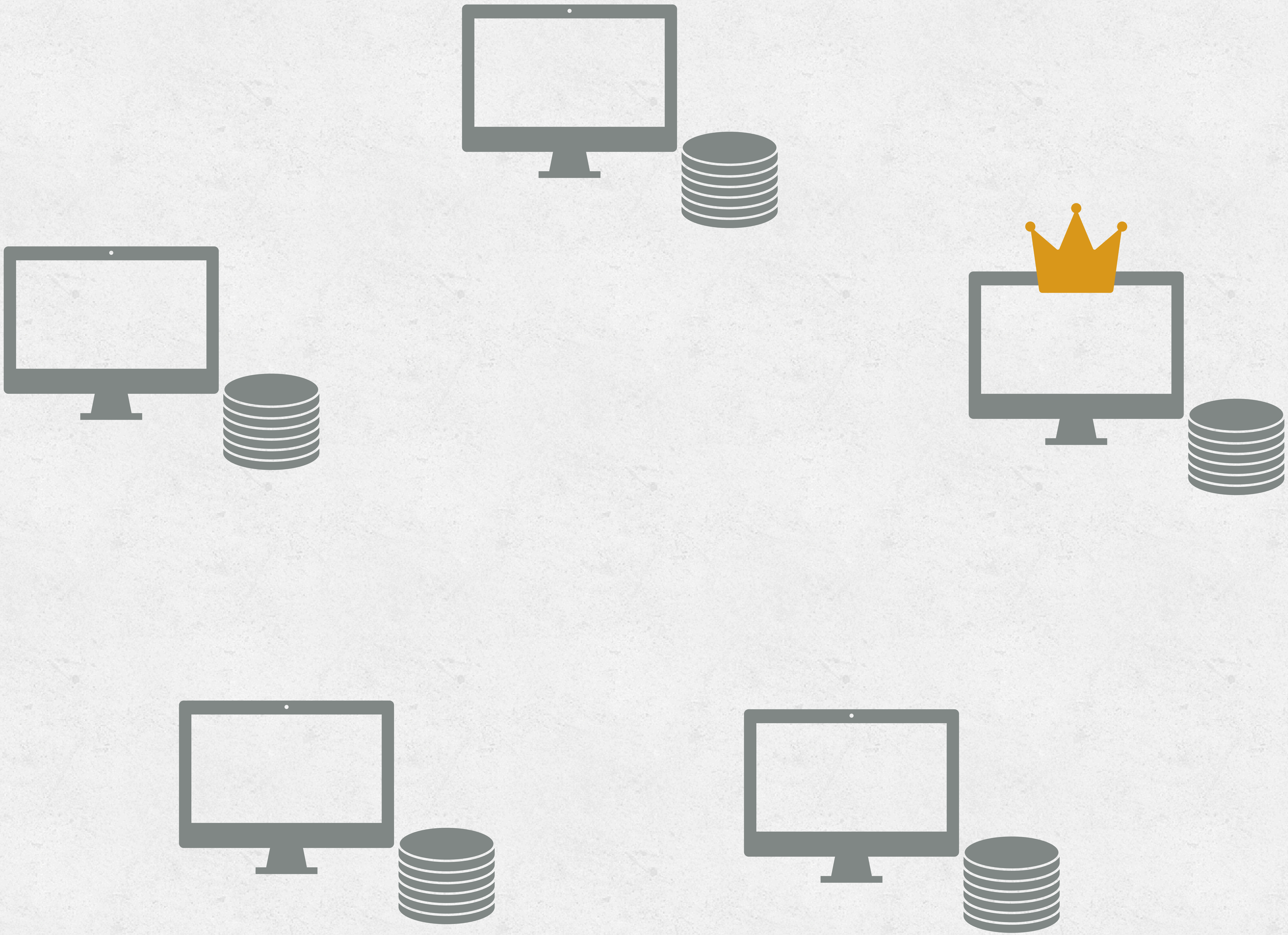
querying P1As stored at acceptors, use a ballot larger than max seen.

Allows acceptors to "forget" certain P2As but still safe. (Why?)

Acceptor State

- Write P1A messages synchronously to disk.
- Need to learn about accepted values.
- Solution: learn about missing P2As from the recent leader (or a future one which reads from a quorum not including this node).


If all messages are logged, each server can recover independently. Failures don't cause permanent deadlock.




Certain storage and recovery schemes yield weaker liveness conditions.

Problem #2: Disks Can Fail!

Need For A Diskless Recovery Mechanism

Need For A Diskless Recovery Mechanism

- Disk failures are by far the most common hardware failures. Relying on disks for recovery can be self-defeating.

Need For A Diskless Recovery Mechanism

- Disk failures are by far the most common hardware failures. Relying on disks for recovery can be self-defeating.

- **Reconfiguration** protocols exist but are costly. They typically rely on consensus and cause the normal protocol to temporarily halt.

Need For A Diskless Recovery Mechanism

- Disk failures are by far the most common hardware failures. Relying on disks for recovery can be self-defeating.
- **Reconfiguration** protocols exist but are costly. They typically rely on consensus and cause the normal protocol to temporarily halt.
- However, getting diskless recovery right is tricky.

Viewstamped Replication (Revisited)

VR is a state-machine replication protocol akin to Paxos.

A later version contained a **diskless recovery protocol** that was supposed to allow nodes to recover even when they didn't write any proposals or leader change messages to disk.

VR Diskless Recovery

Upon recovery, a node:

1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.

2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.

3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.

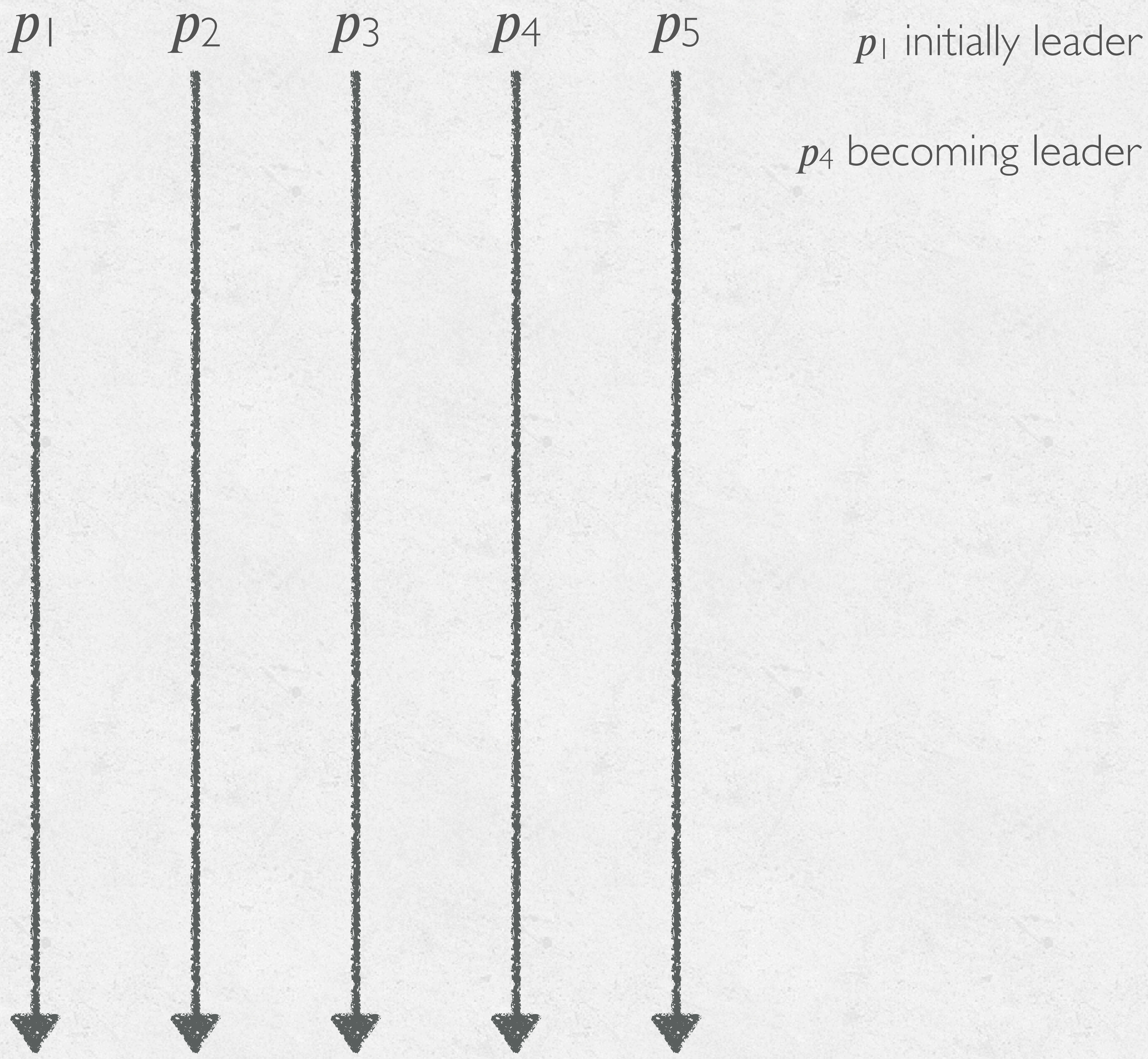
VR Diskless Recovery

Upon recovery, a node:

1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.
2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.
3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.

p_1 initially leader

VR Diskless Recovery



Upon recovery, a node:

1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.

2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.

3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.

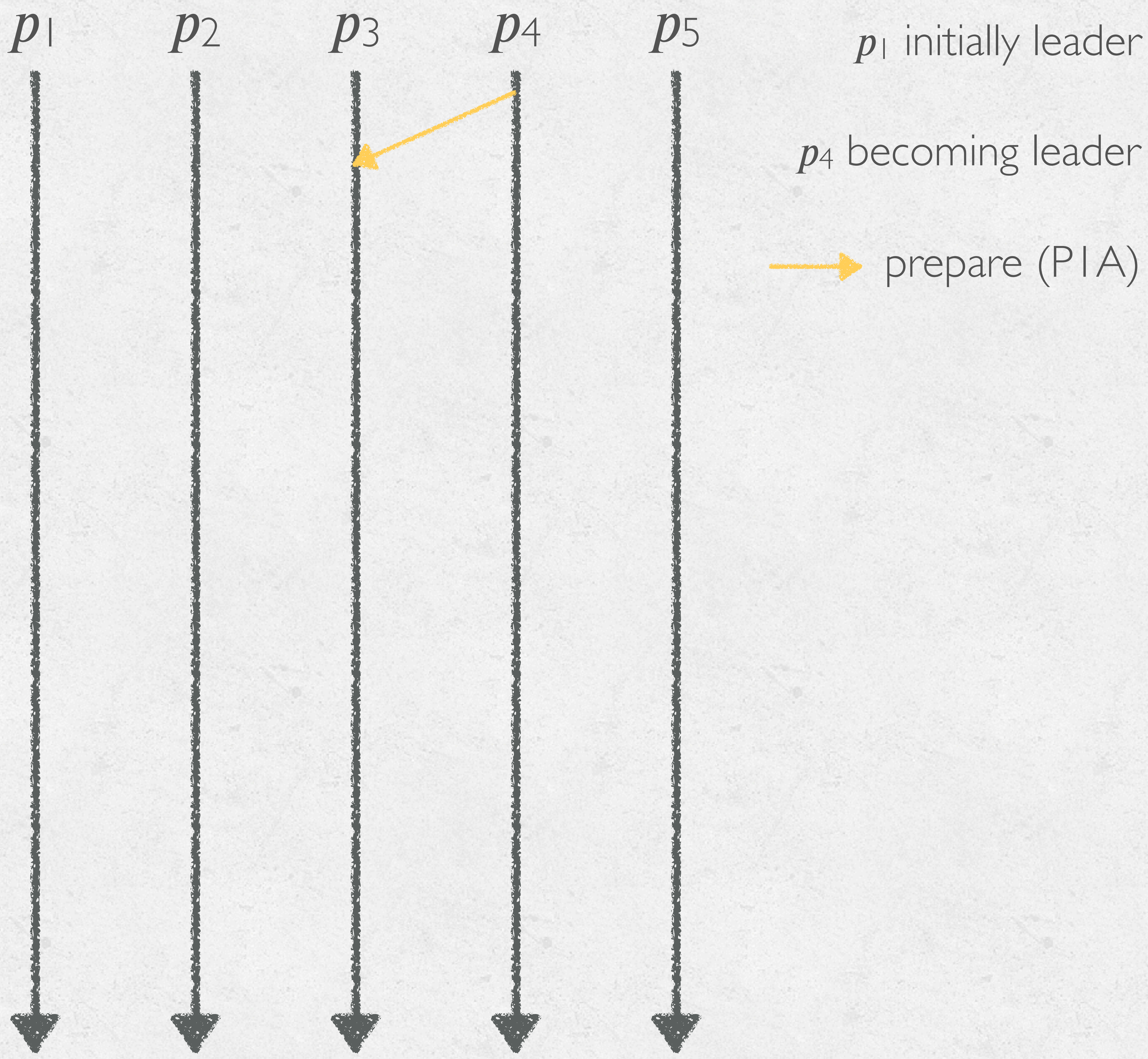
VR Diskless Recovery

Upon recovery, a node:

1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.

2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.

3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.



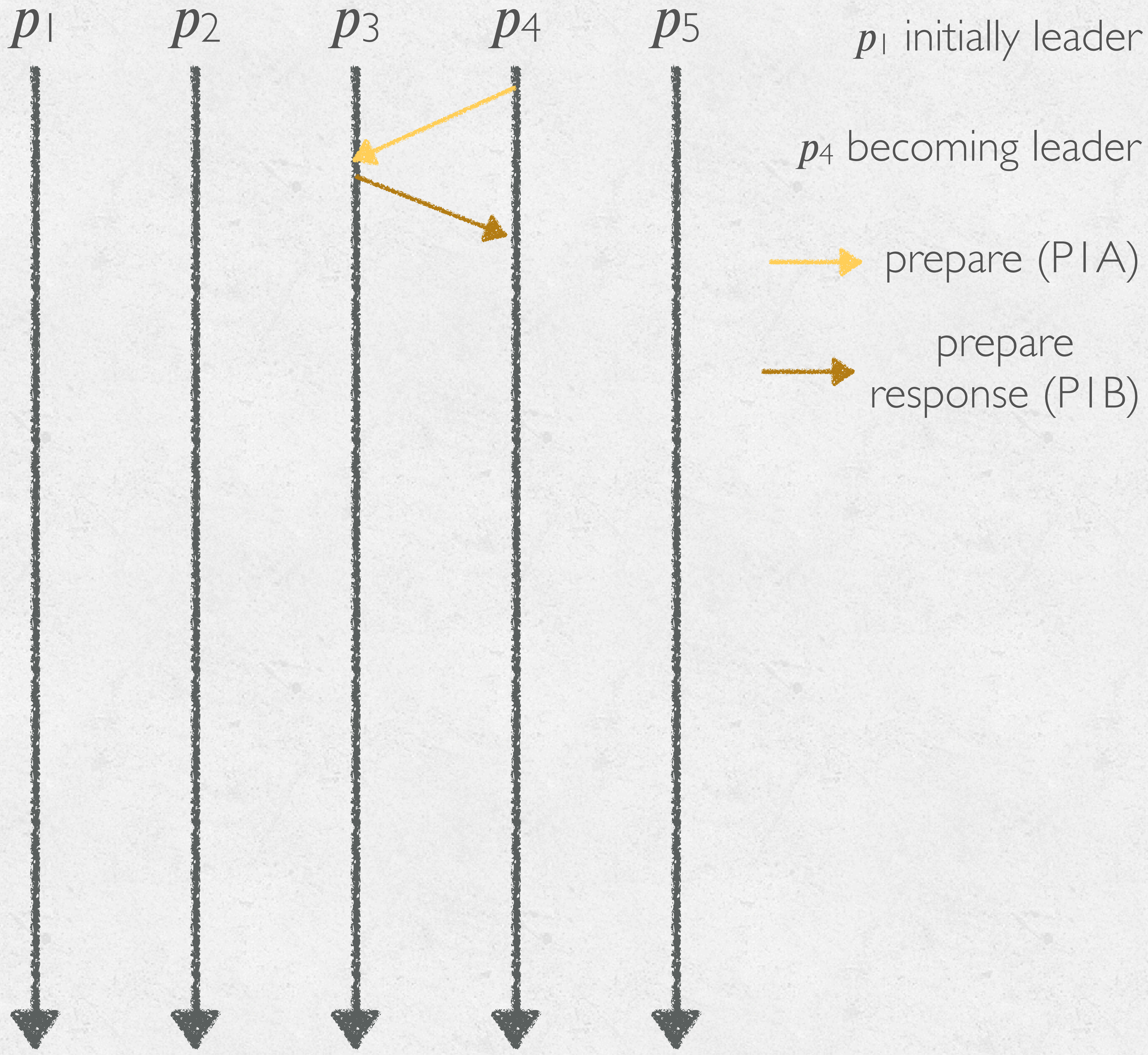
VR Diskless Recovery

Upon recovery, a node:

1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.

2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.

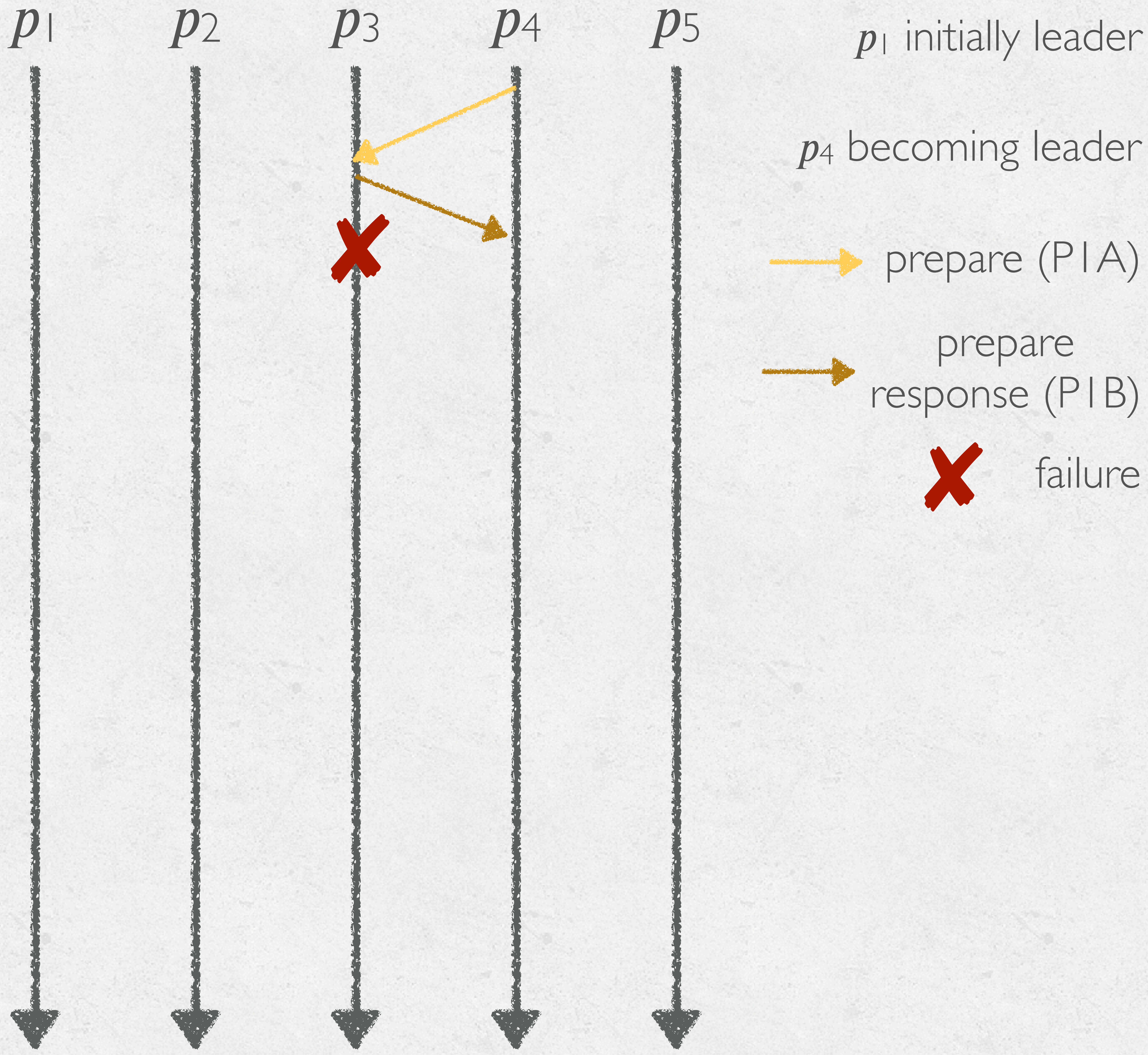
3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.



VR Diskless Recovery

Upon recovery, a node:

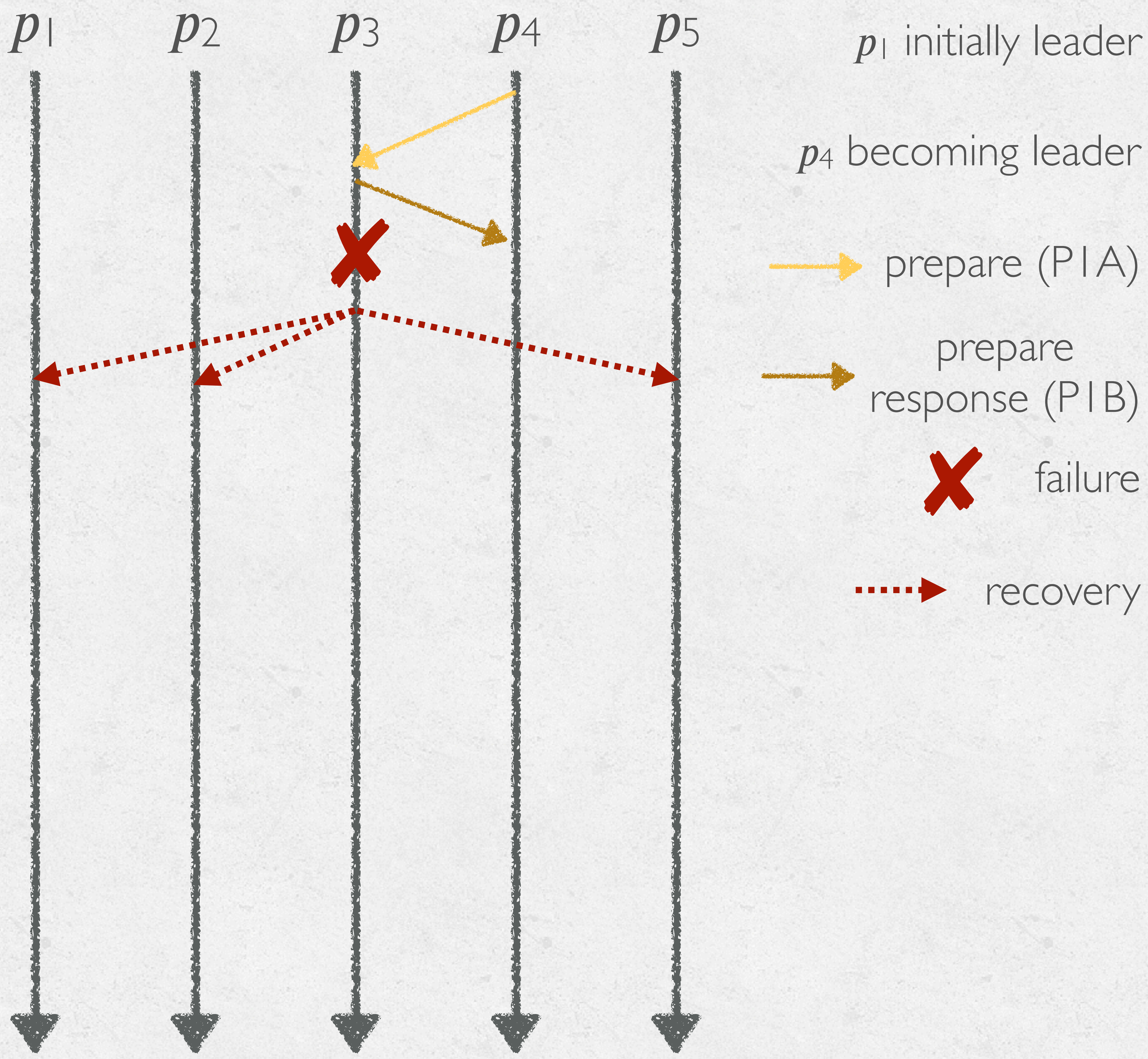
1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.
2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.
3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.



VR Diskless Recovery

Upon recovery, a node:

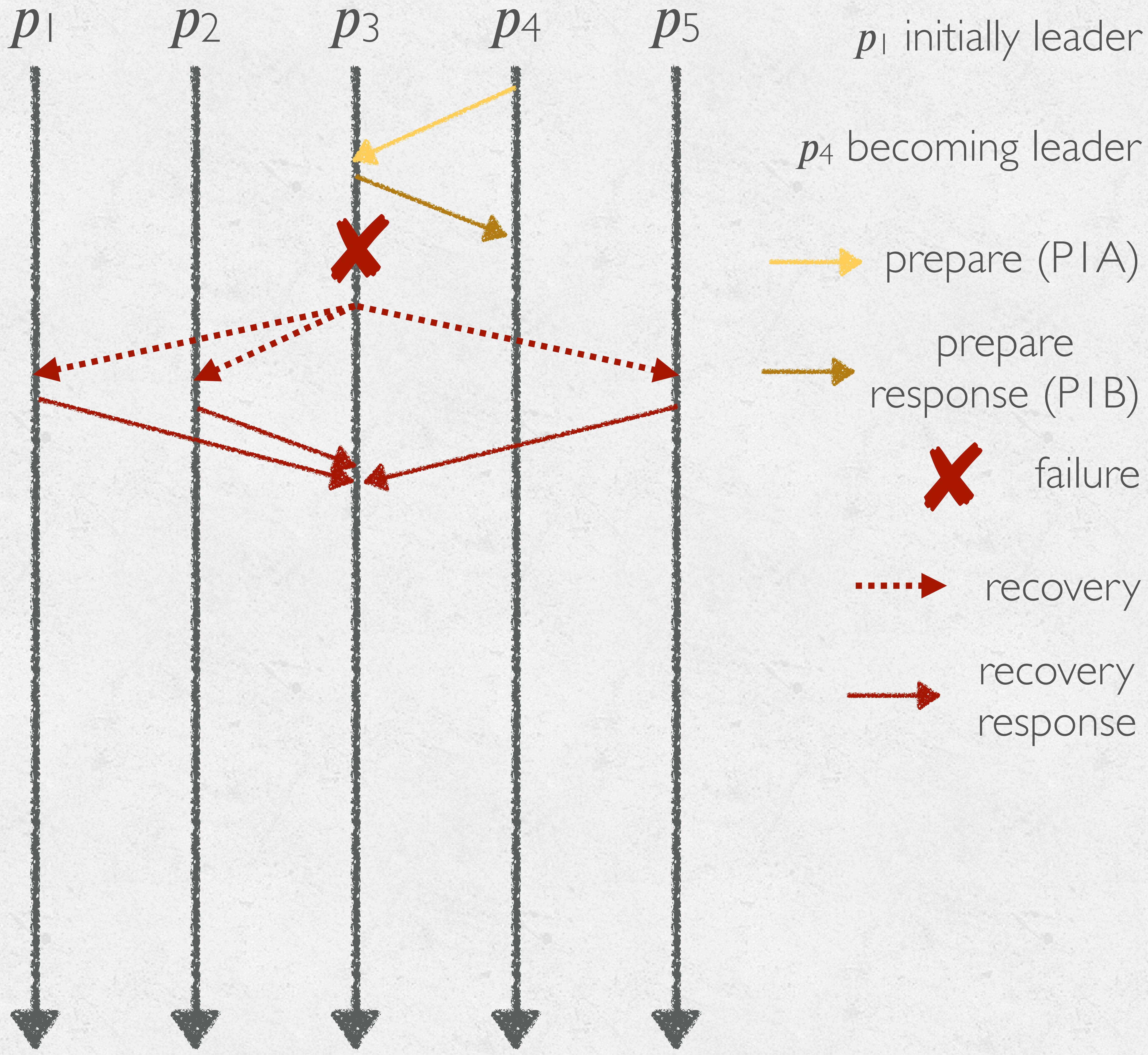
1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.
2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.
3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.



VR Diskless Recovery

Upon recovery, a node:

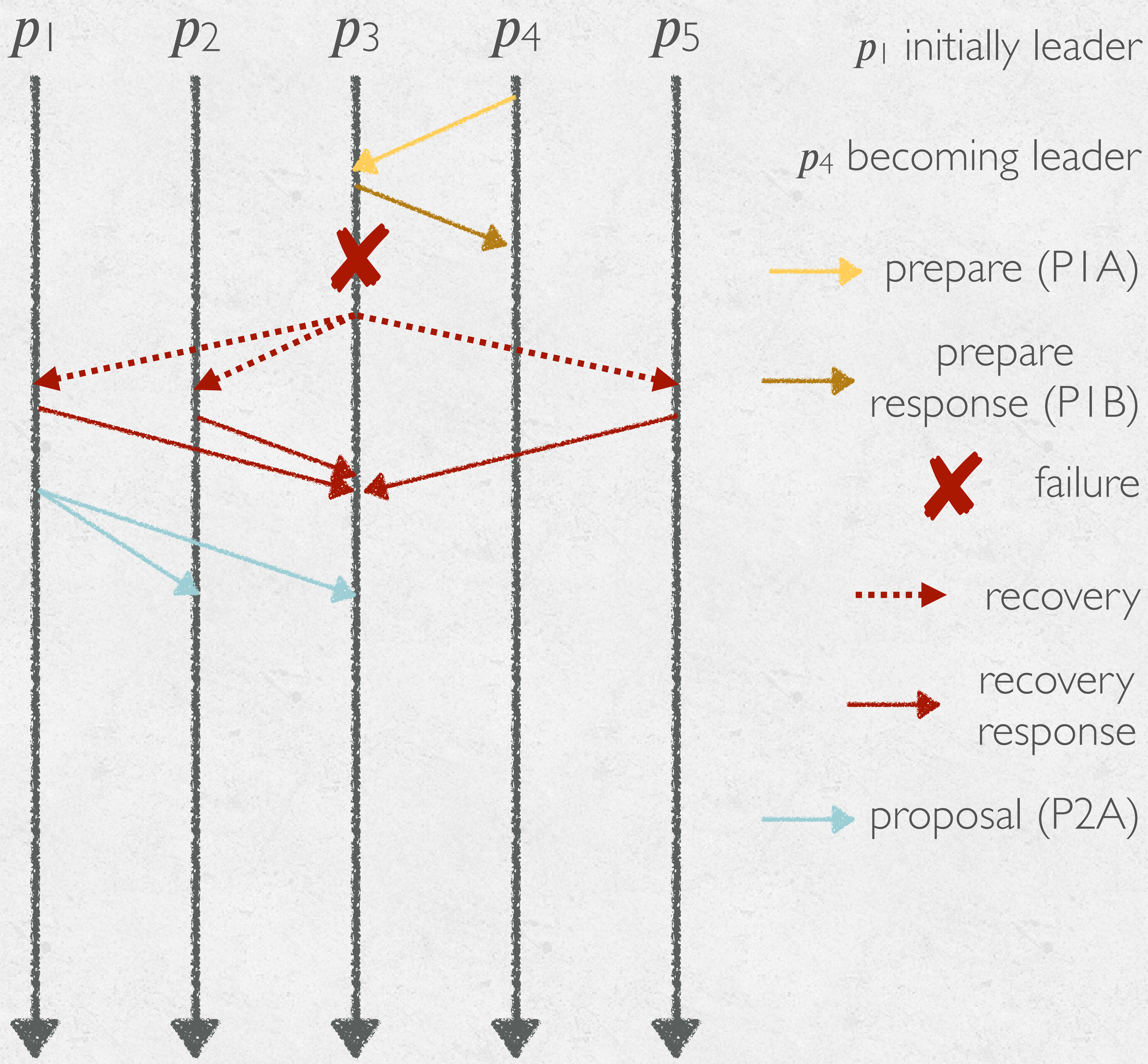
1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.
2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.
3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.



VR Diskless Recovery

Upon recovery, a node:

1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.
2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.
3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.



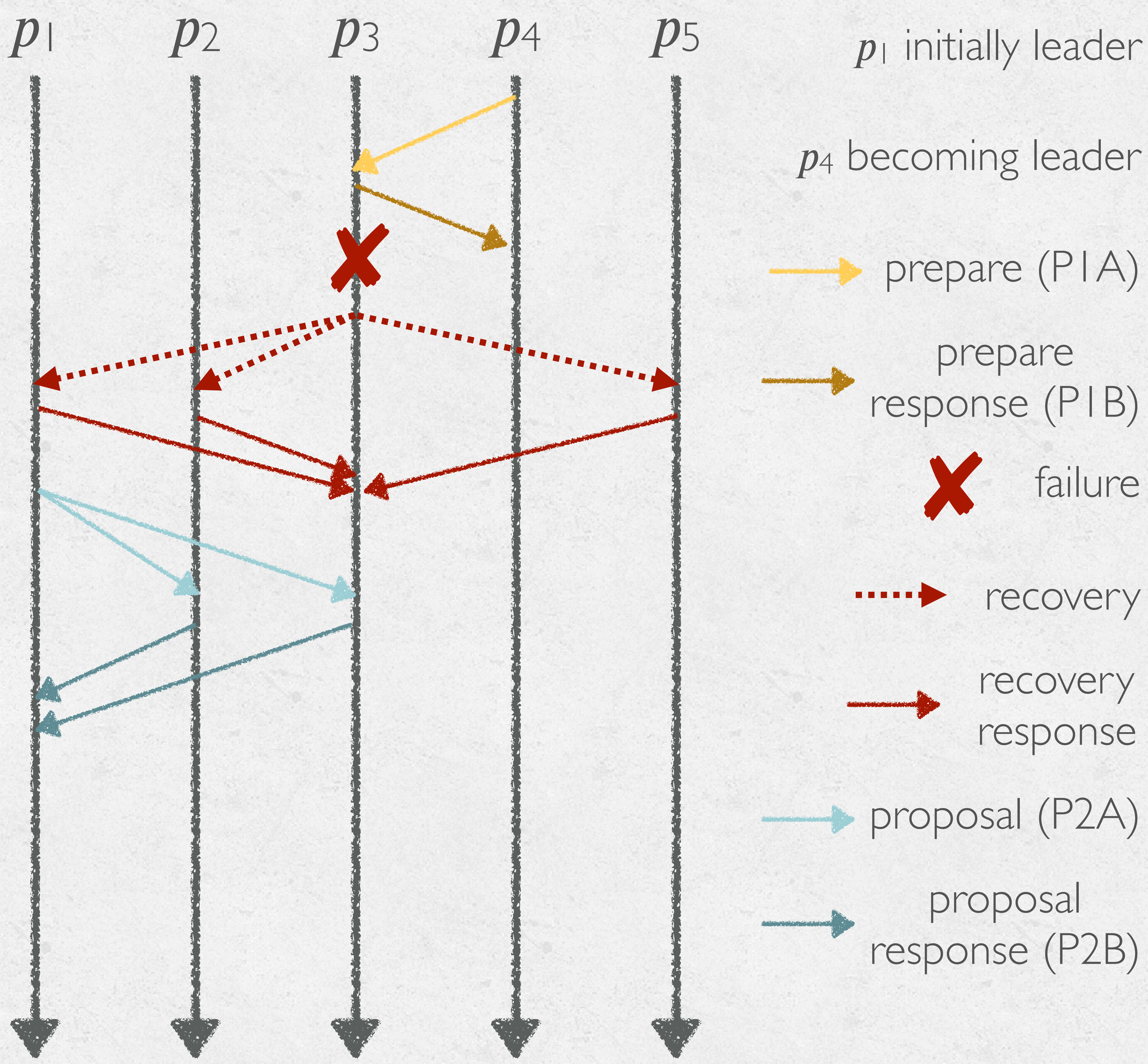
VR Diskless Recovery

Upon recovery, a node:

1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.

2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.

3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.



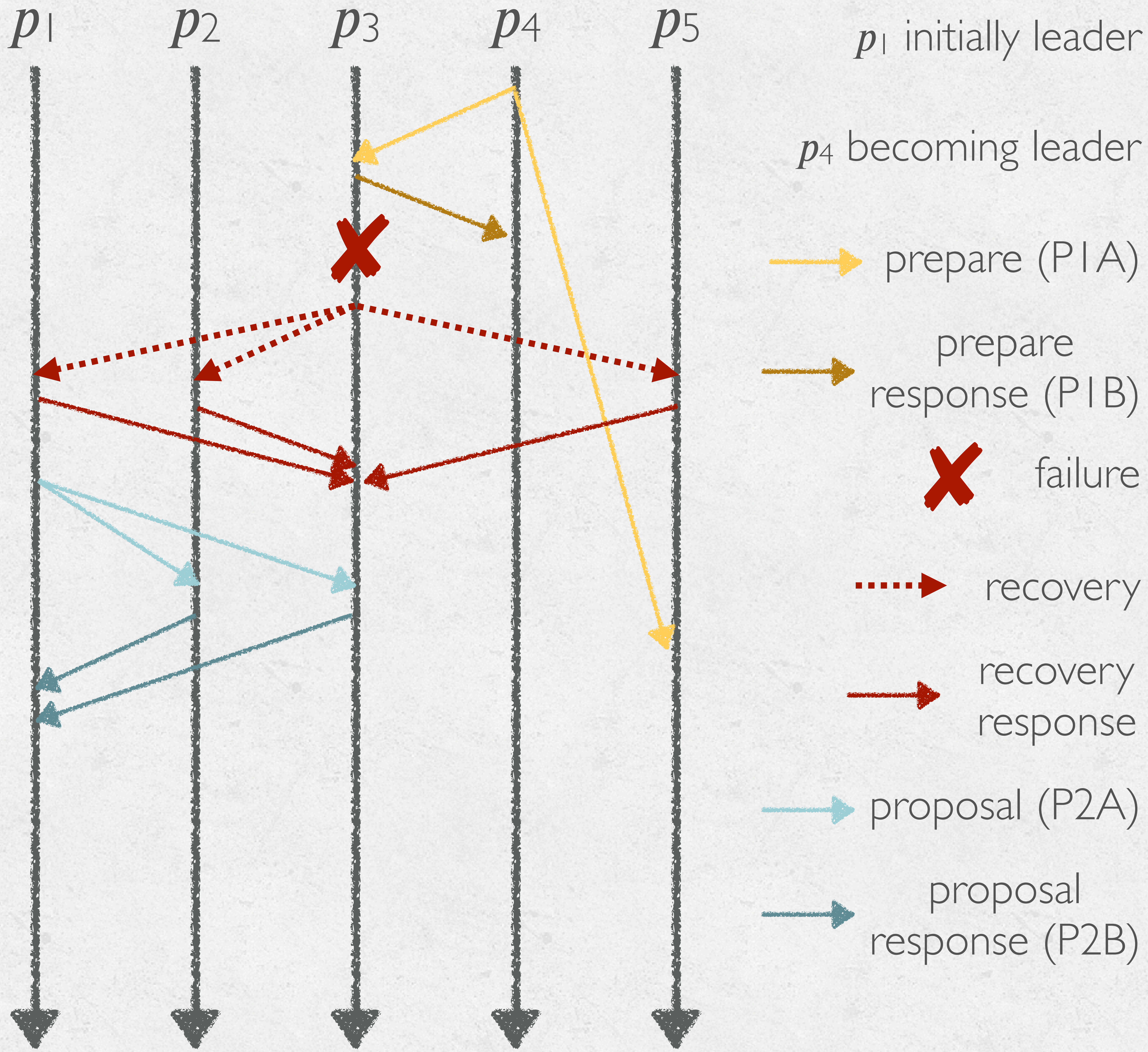
VR Diskless Recovery

Upon recovery, a node:

1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.

2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.

3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.



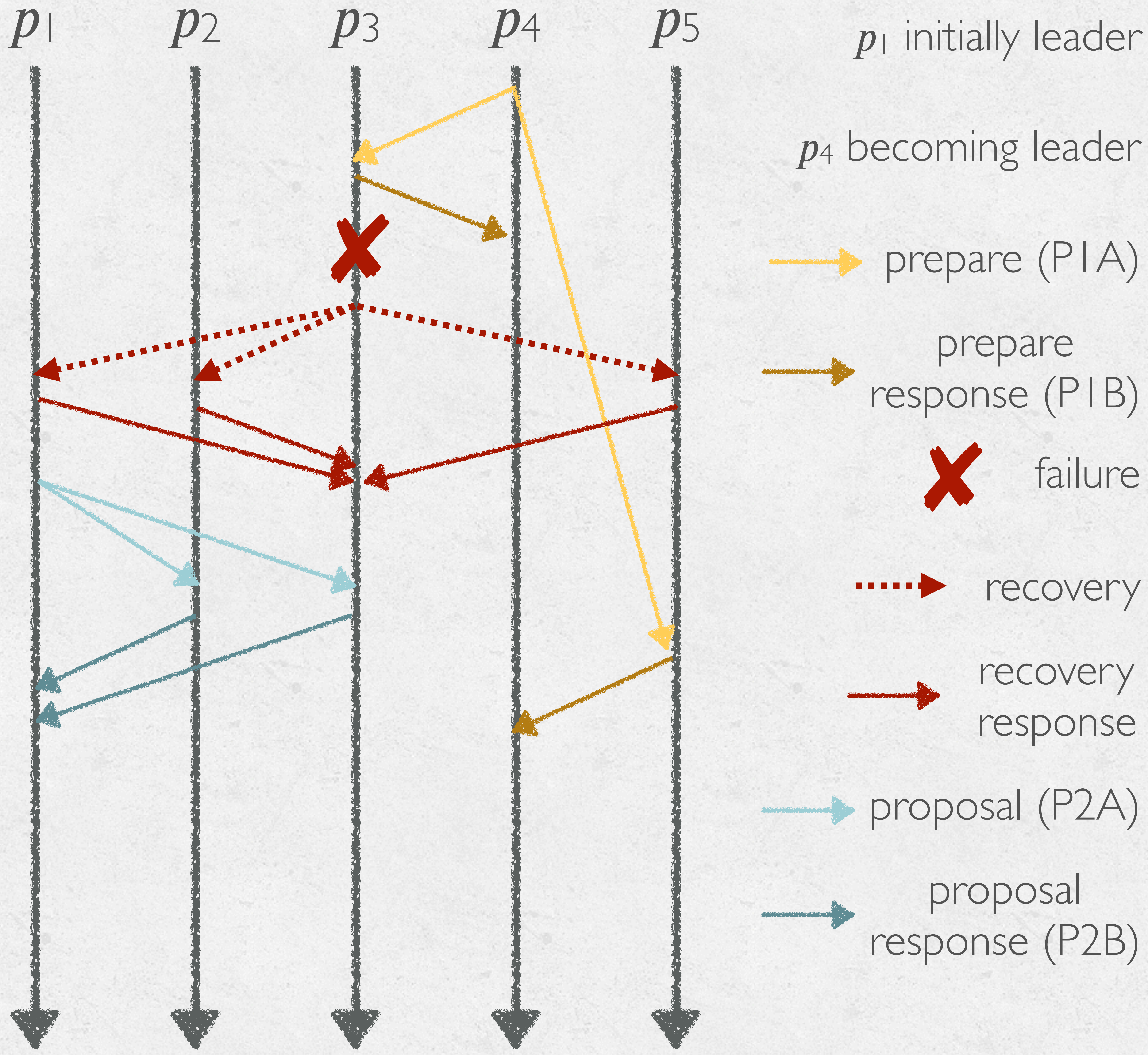
VR Diskless Recovery

Upon recovery, a node:

1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.

2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.

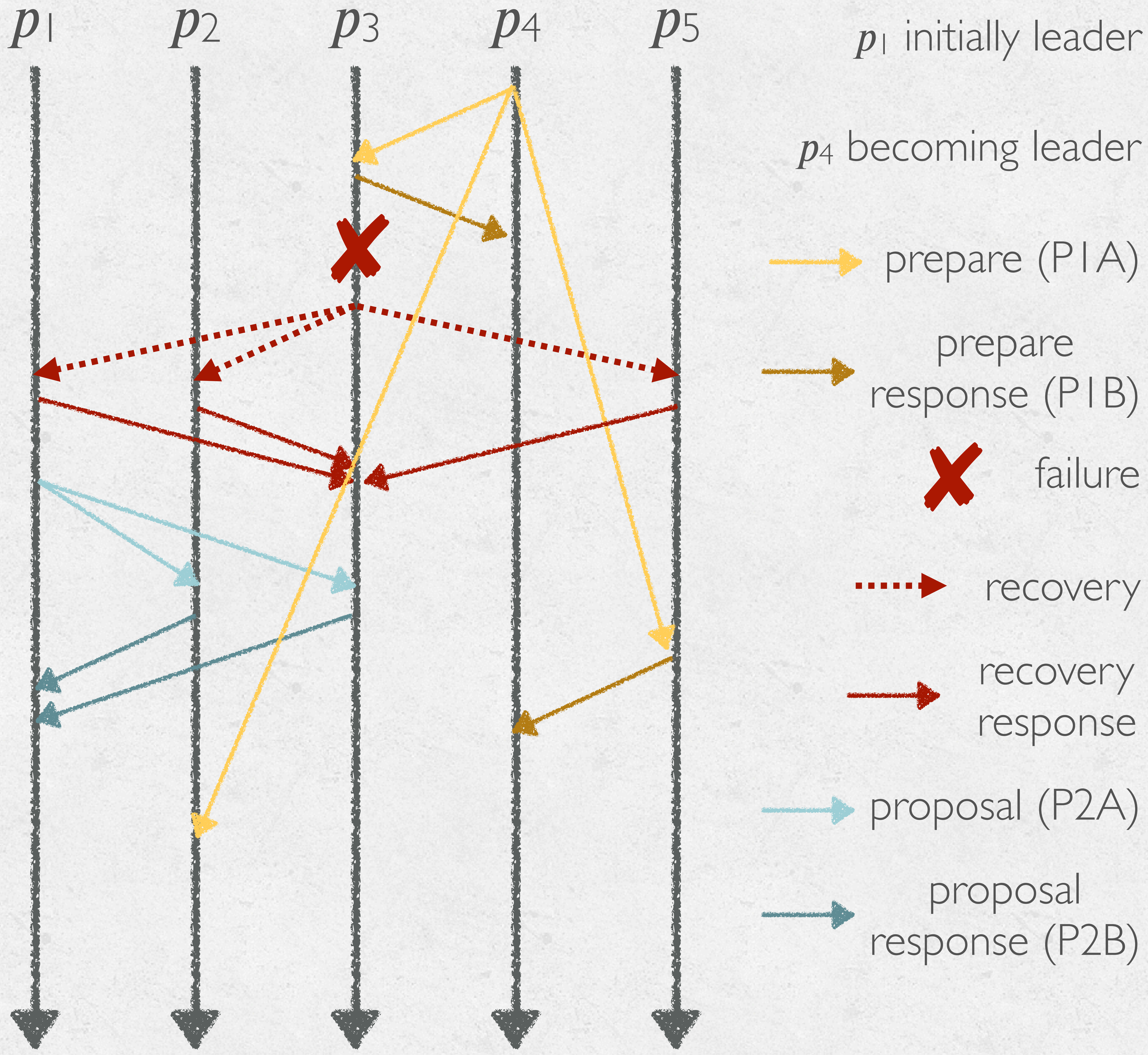
3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.



VR Diskless Recovery

Upon recovery, a node:

1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.
2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.
3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.



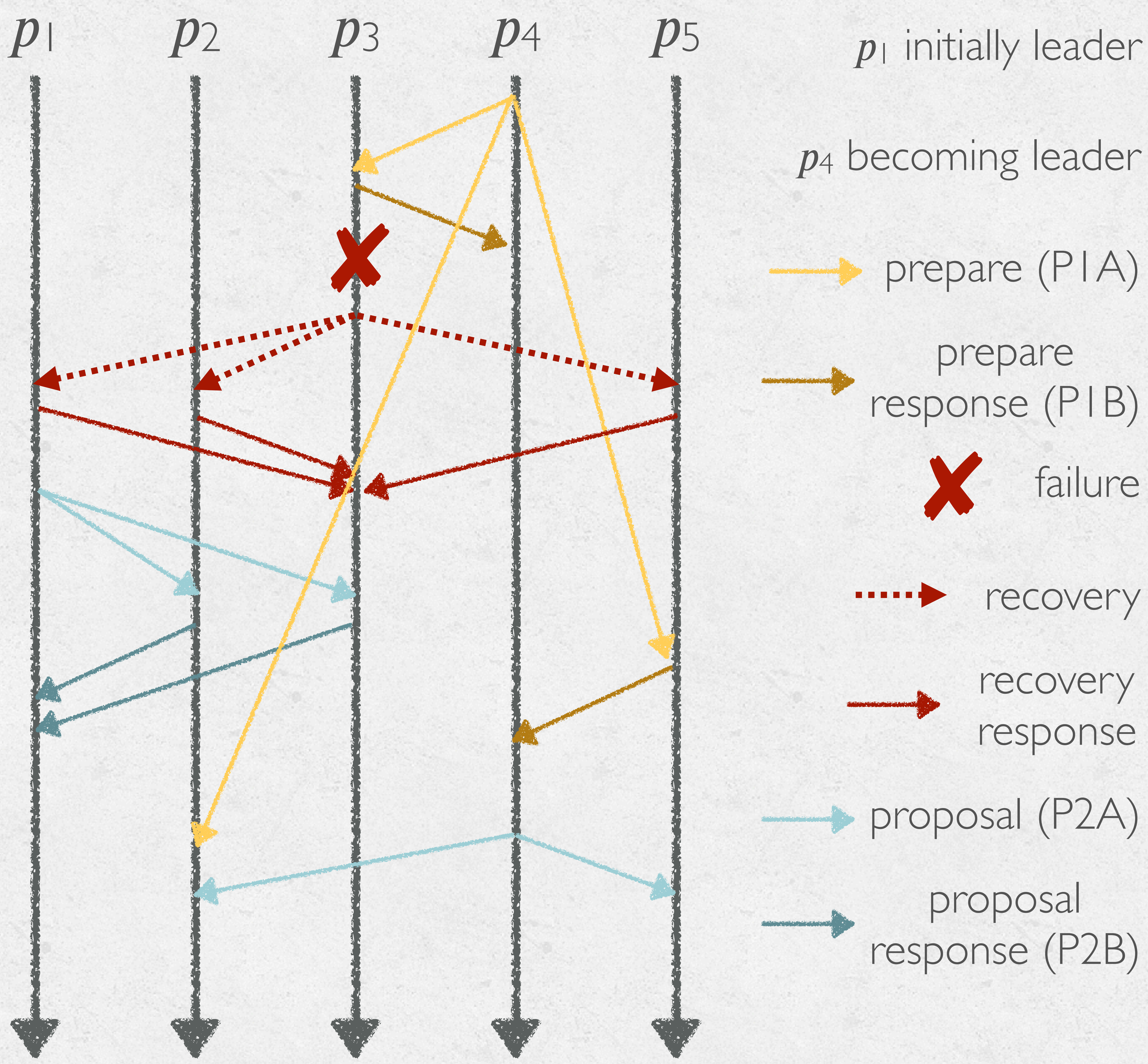
VR Diskless Recovery

Upon recovery, a node:

1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.

2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.

3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.



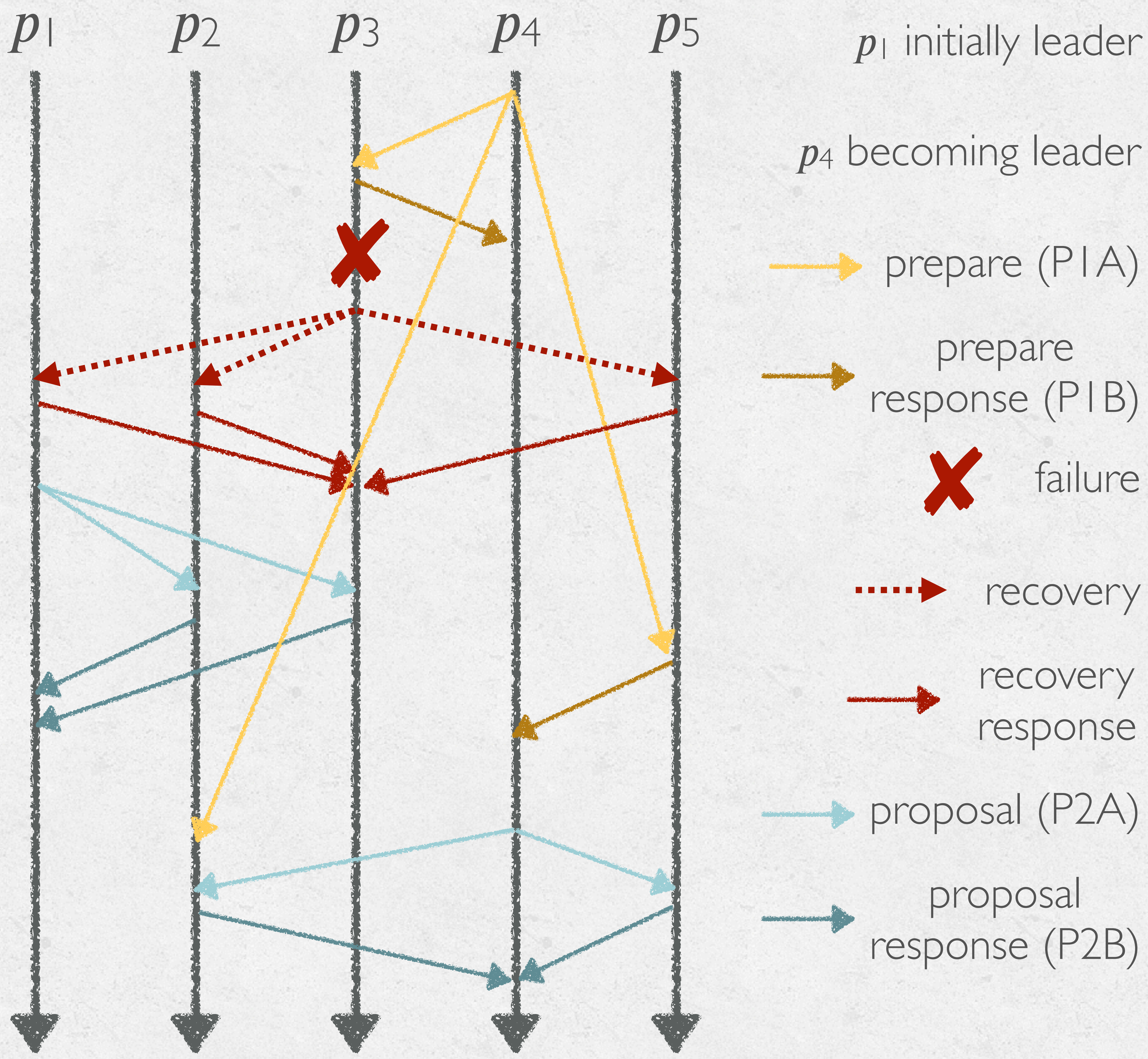
VR Diskless Recovery

Upon recovery, a node:

1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.

2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.

3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.



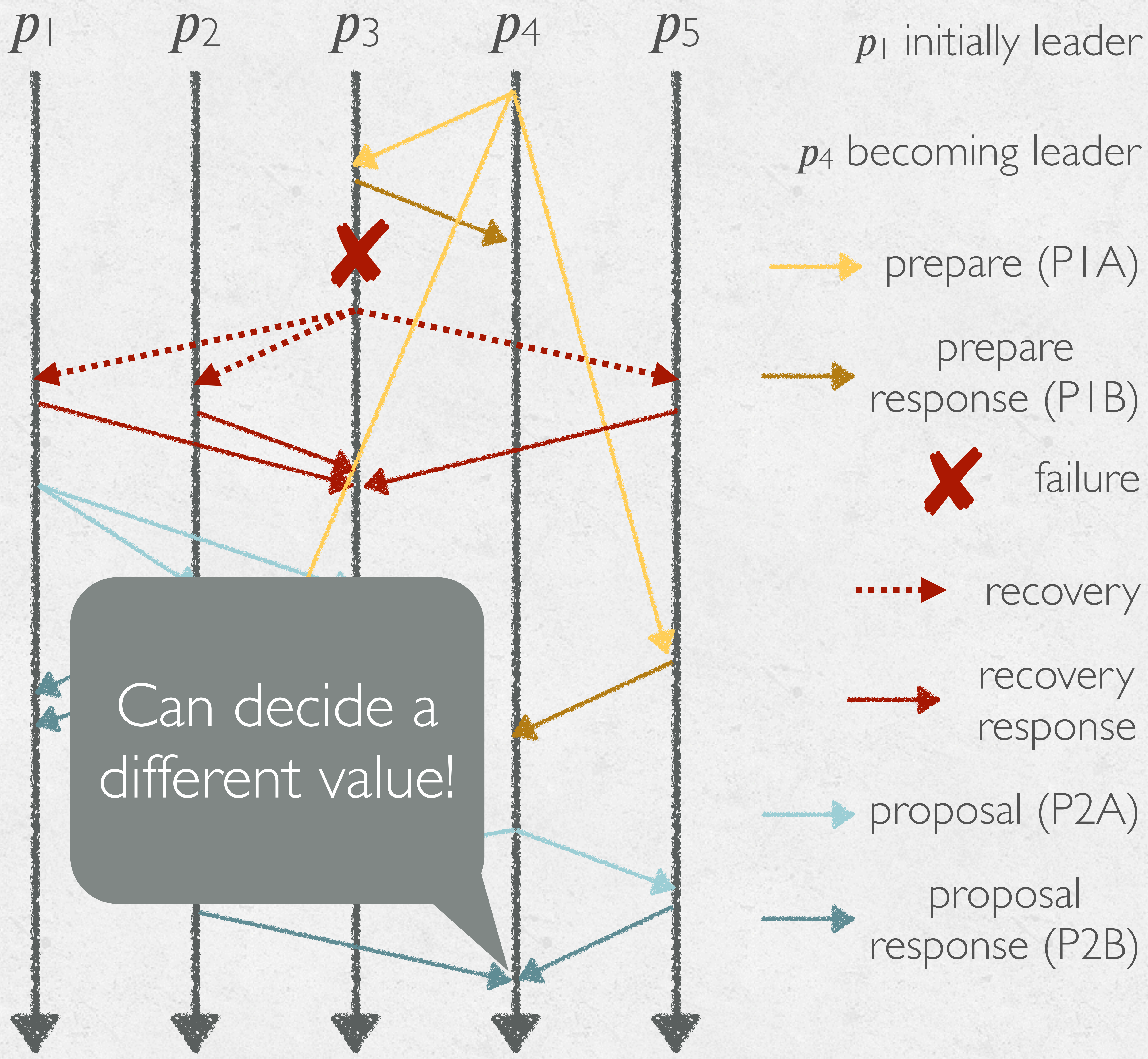
VR Diskless Recovery

Upon recovery, a node:

1. Sends ⟨RECOVERY, x⟩ to all other nodes, where x is a unique value to guarantee freshness.

2. Nodes that are currently OPERATIONAL (not recovering) respond with ⟨RECOVERY-RESPONSE, x, b, l⟩, where b is its ballot number and l is its log.

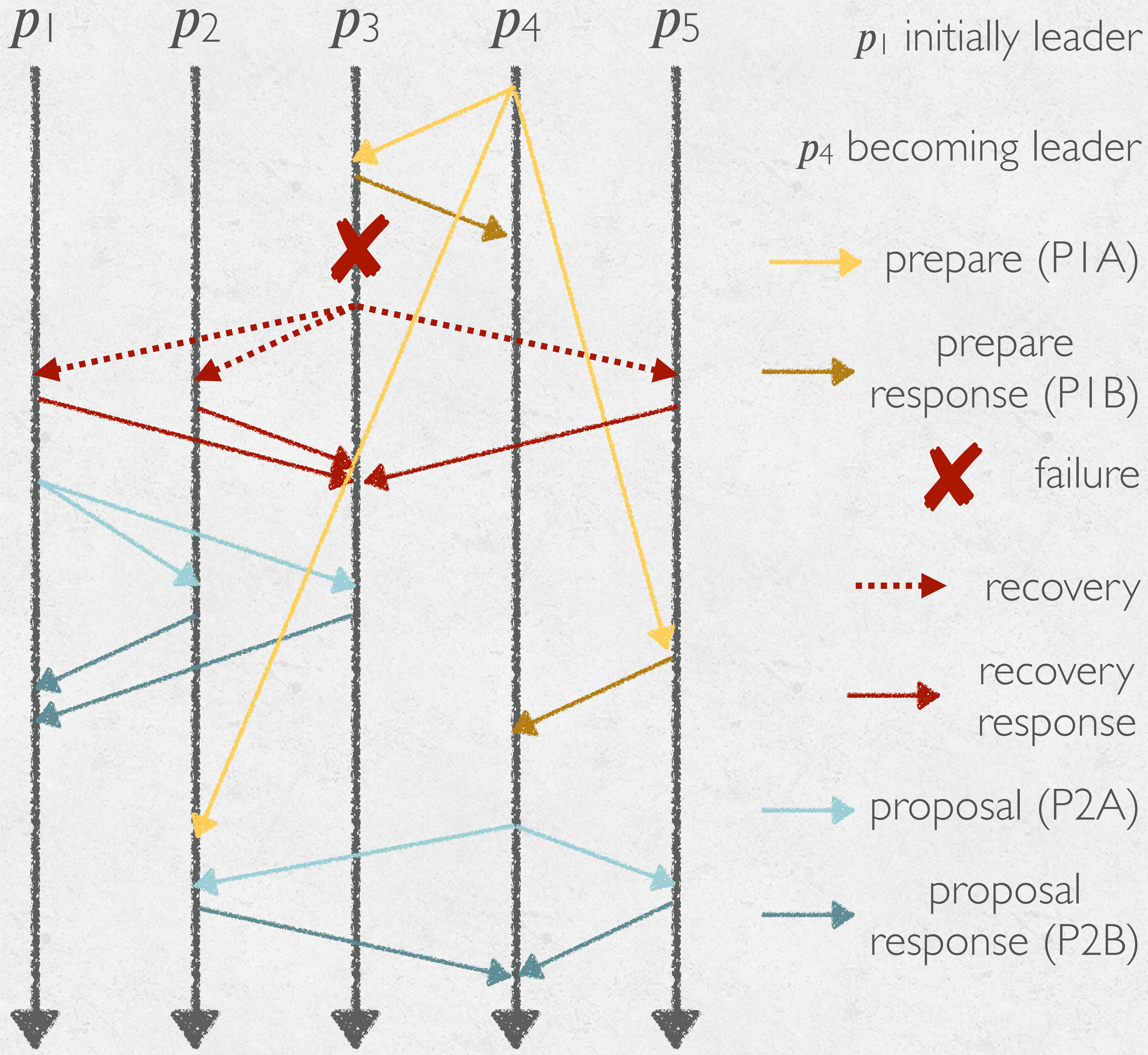
3. The recovering node waits for responses from a majority, including one from the leader node associated with the largest ballot seen. It updates its own log and returns its status to OPERATIONAL.



Problem: Unstable Quorums

An **unstable quorum** is one in which some of the participants have already crashed and restarted and no longer remember the relevant information.

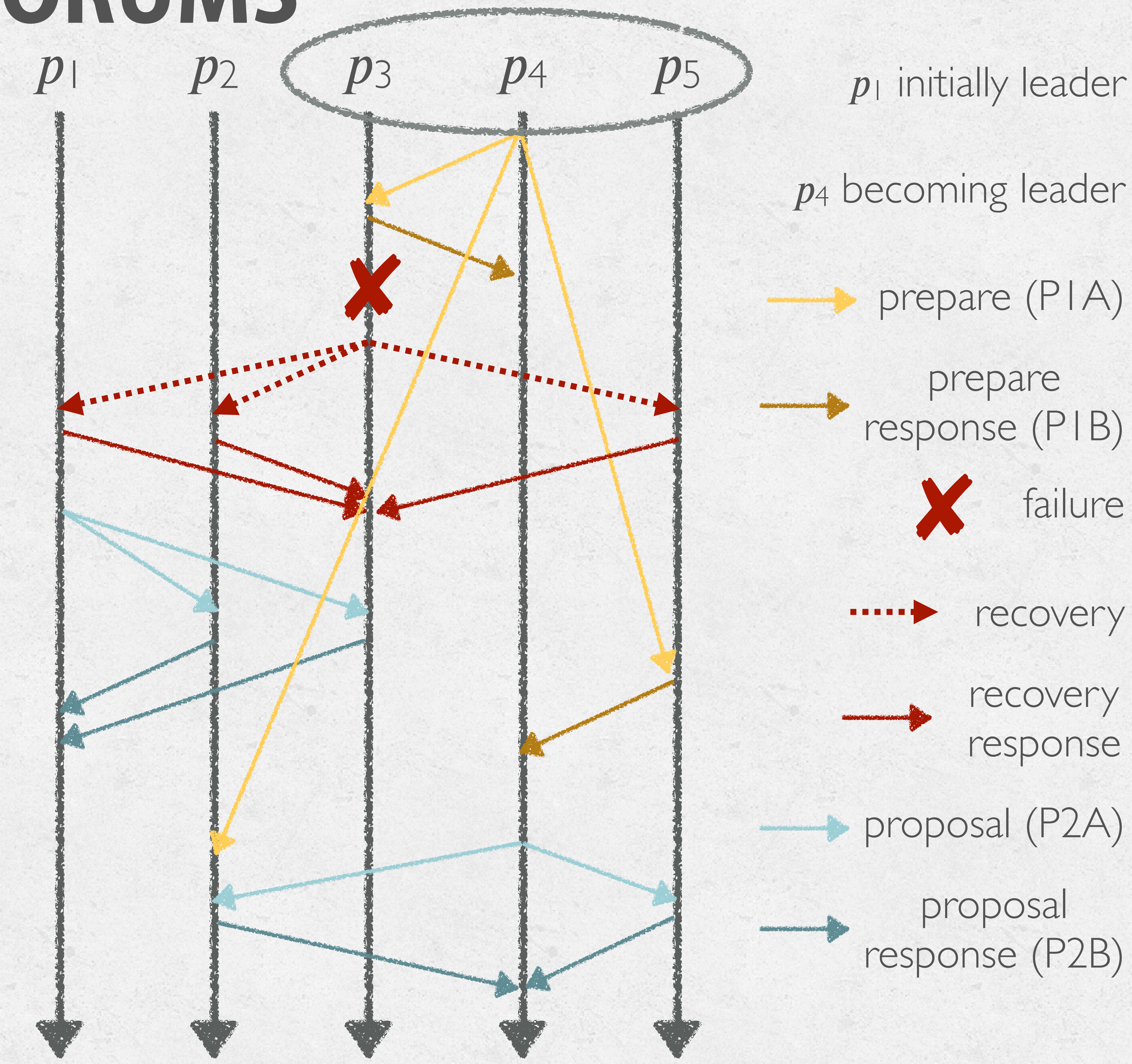
Unstable quorums can forget key information, causing the system to violate safety.



PROBLEM: UNSTABLE QUORUMS

An **unstable quorum** is one in which some of the participants have already crashed and restarted and no longer remember the relevant information.

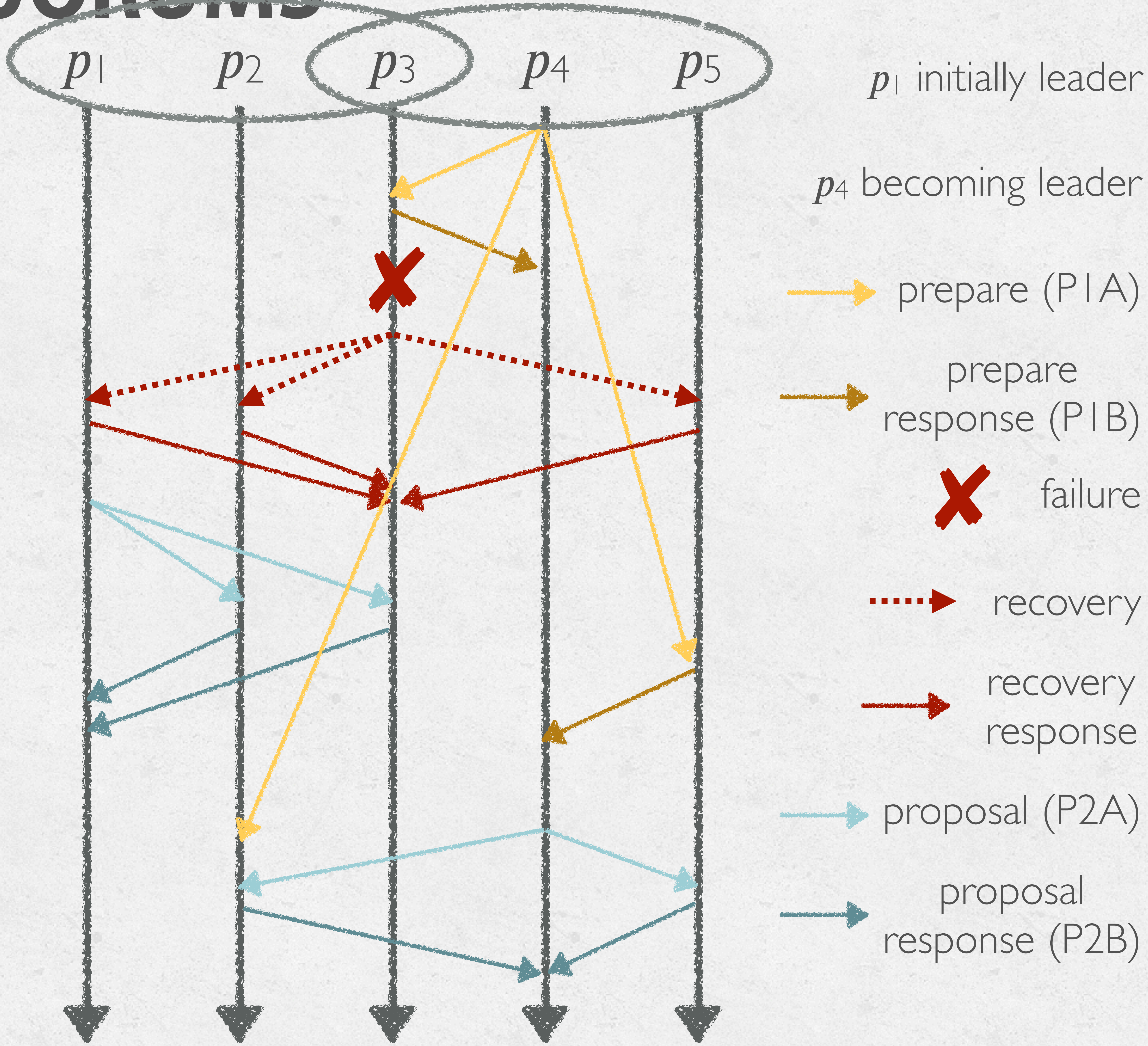
Unstable quorums can forget key information, causing the system to violate safety.



PROBLEM: UNSTABLE QUORUMS

An **unstable quorum** is one in which some of the participants have already crashed and restarted and no longer remember the relevant information.

Unstable quorums can forget key information, causing the system to violate safety.



Goal: Emulate Stable Storage

- Our goal is to provide each node with a **set** that it can ***add values to*** and ***read from***. This set need only support a single writer/reader.
- We want to guarantee that values written are later returned and that completed reads are monotonic (i.e., that a value in the set returned by one read is returned by later reads).
- We want these guarantees to hold even across node failures.
- Nodes can use this set just like stable storage. Therefore, any protocol that is safe with stable storage will still be safe when stable storage is replaced by our diskless storage.

Amnesia Fault Tolerance

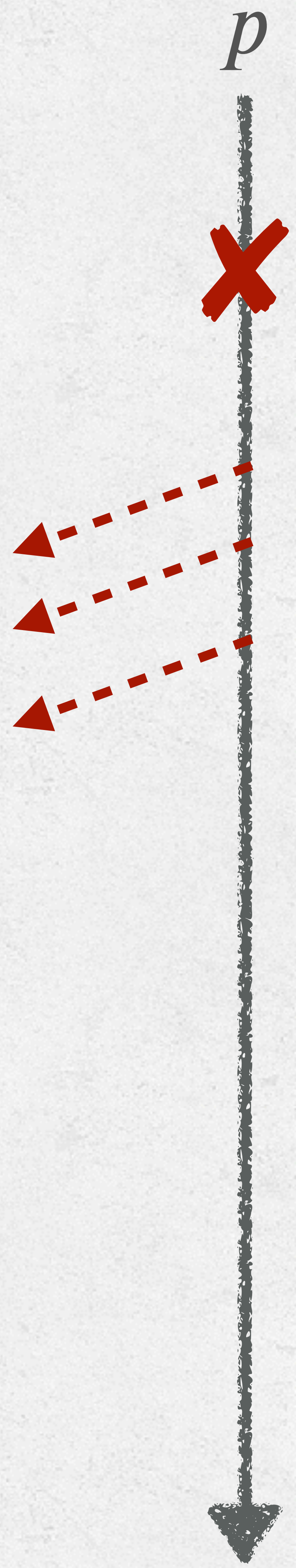
- Nodes can crash and restart.
- Upon restart, they lose all of their local state.
- Restarting nodes run a *recovery protocol* to re-learn any necessary information.
- How can we ever make progress when nodes can forget what we've told them at any moment?

Amnesia Fault Tolerance

- Nodes can crash and restart.
- Upon restart, they lose all of their local state.
- Restarting nodes run a *recovery protocol* to re-learn any necessary information.
- How can we ever make progress when nodes can forget what we've told them at any moment?

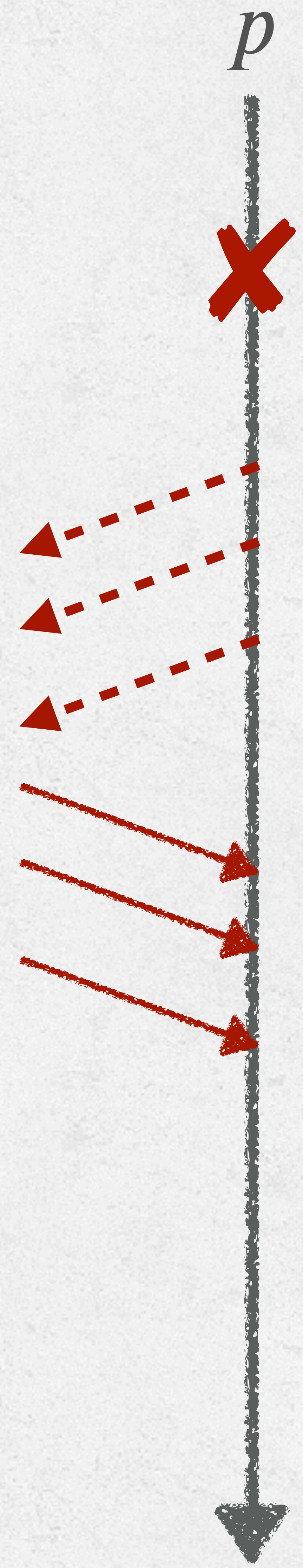
Amnesia Fault Tolerance

- Nodes can crash and restart.
- Upon restart, they lose all of their local state.
- Restarting nodes run a *recovery protocol* to re-learn any necessary information.
- How can we ever make progress when nodes can forget what we've told them at any moment?



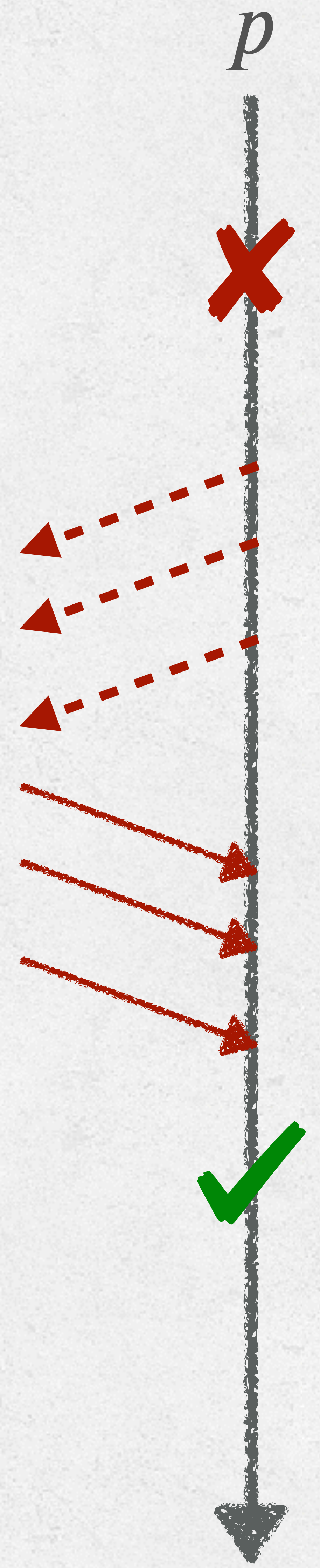
Amnesia Fault Tolerance

- Nodes can crash and restart.
- Upon restart, they lose all of their local state.
- Restarting nodes run a *recovery protocol* to re-learn any necessary information.
- How can we ever make progress when nodes can forget what we've told them at any moment?



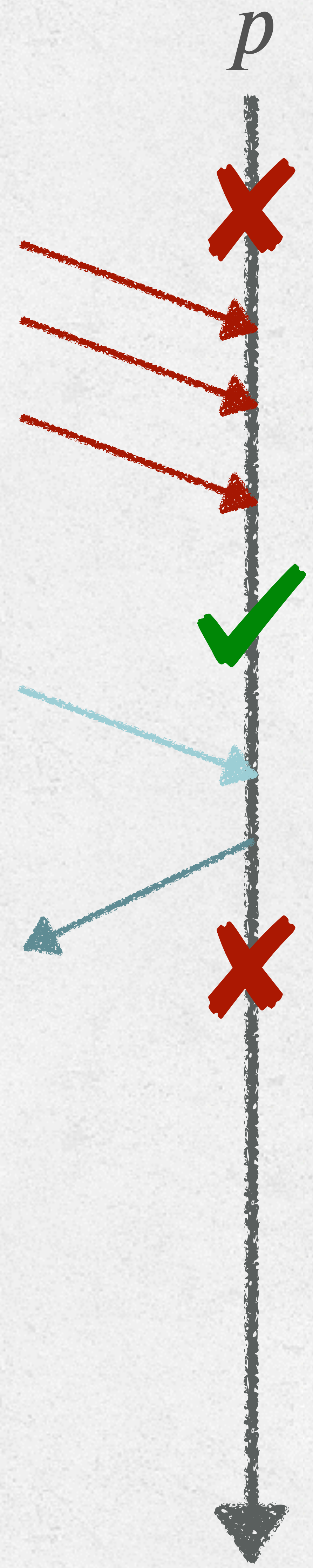
Amnesia Fault Tolerance

- Nodes can crash and restart.
- Upon restart, they lose all of their local state.
- Restarting nodes run a *recovery protocol* to re-learn any necessary information.
- How can we ever make progress when nodes can forget what we've told them at any moment?



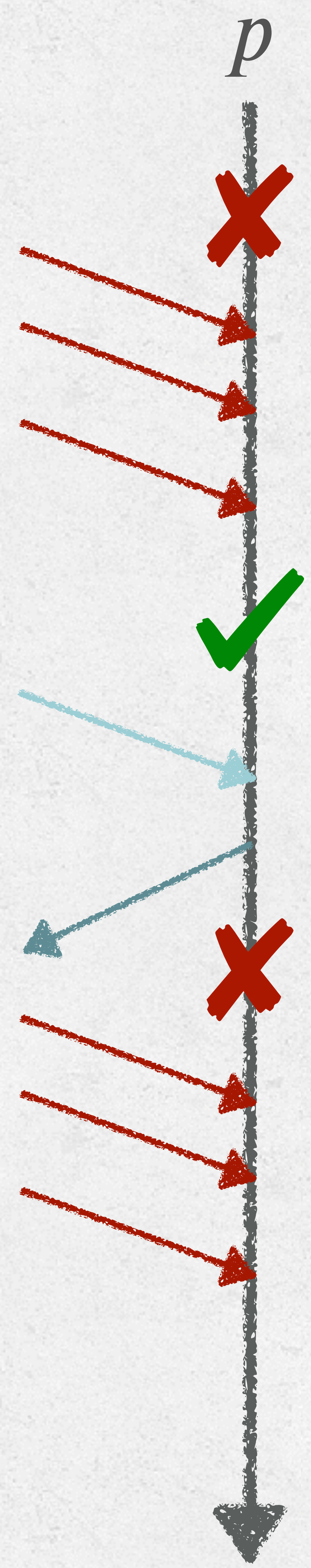
What Do Amnesiacs Know?

- If we don't assume that the initial states of nodes are different across recoveries, we can't do anything.



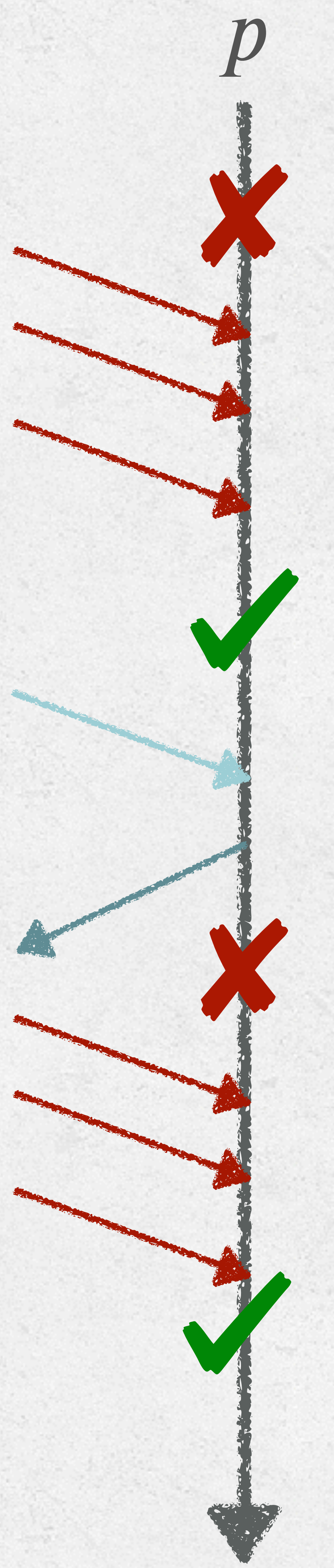
What Do Amnesiacs Know?

- If we don't assume that the initial states of nodes are different across recoveries, we can't do anything.



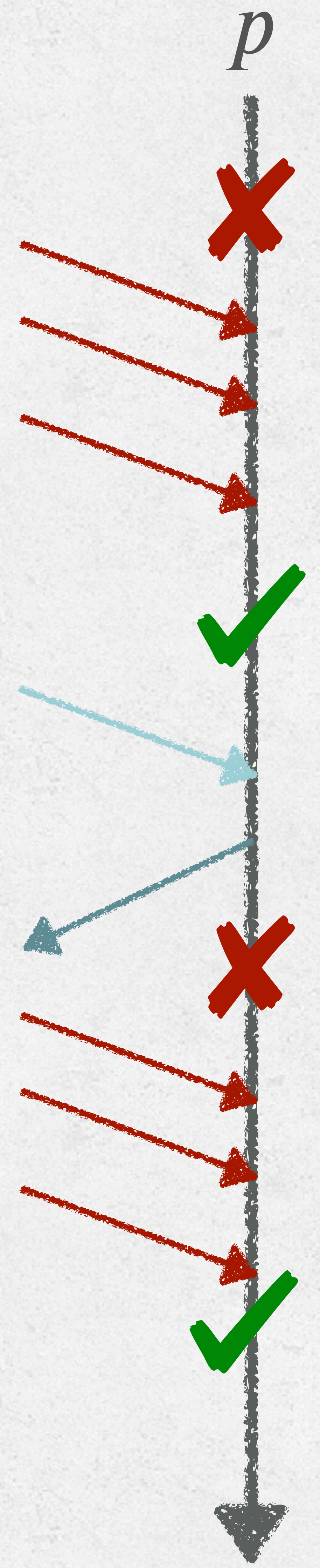
What Do Amnesiacs Know?

- If we don't assume that the initial states of nodes are different across recoveries, we can't do anything.



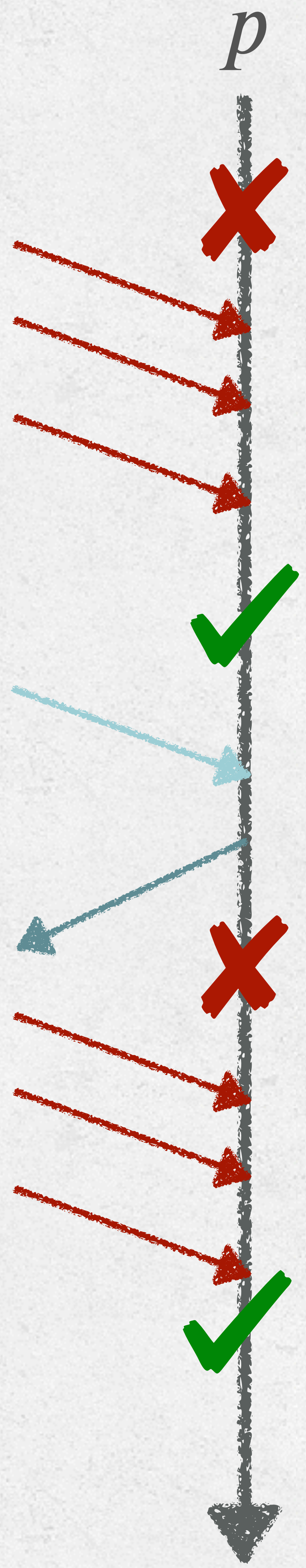
What Do Amnesiacs Know?

- If we don't assume that the initial states of nodes are different across recoveries, we can't do anything.
- Like VR, we could assume that nodes can generate unique values (e.g., by generating a large random number).



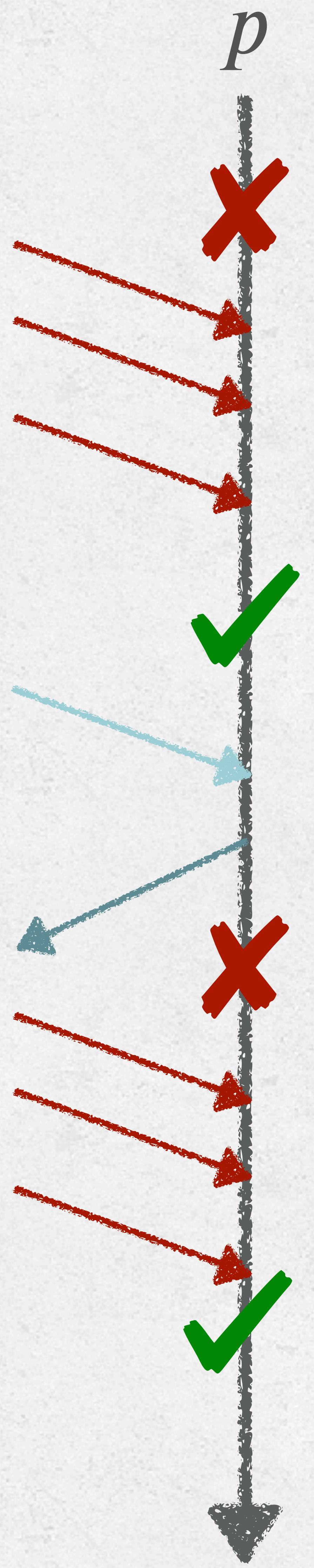
What Do Amnesiacs Know?

- If we don't assume that the initial states of nodes are different across recoveries, we can't do anything.
- Like VR, we could assume that nodes can generate unique values (e.g., by generating a large random number).
- For simplicity, we'll assume that nodes have access to a **non-decreasing clock** (upon recovery, they wait for at least one tick).



What Do Amnesiacs Know?

- If we don't assume that the initial states of nodes are different across recoveries, we can't do anything.
- Like VR, we could assume that nodes can generate unique values (e.g., by generating a large random number).
- For simplicity, we'll assume that nodes have access to a **non-decreasing clock** (upon recovery, they wait for at least one tick).
- Nodes record this clock value upon beginning recovery. This is called the node's **incarnation ID**.



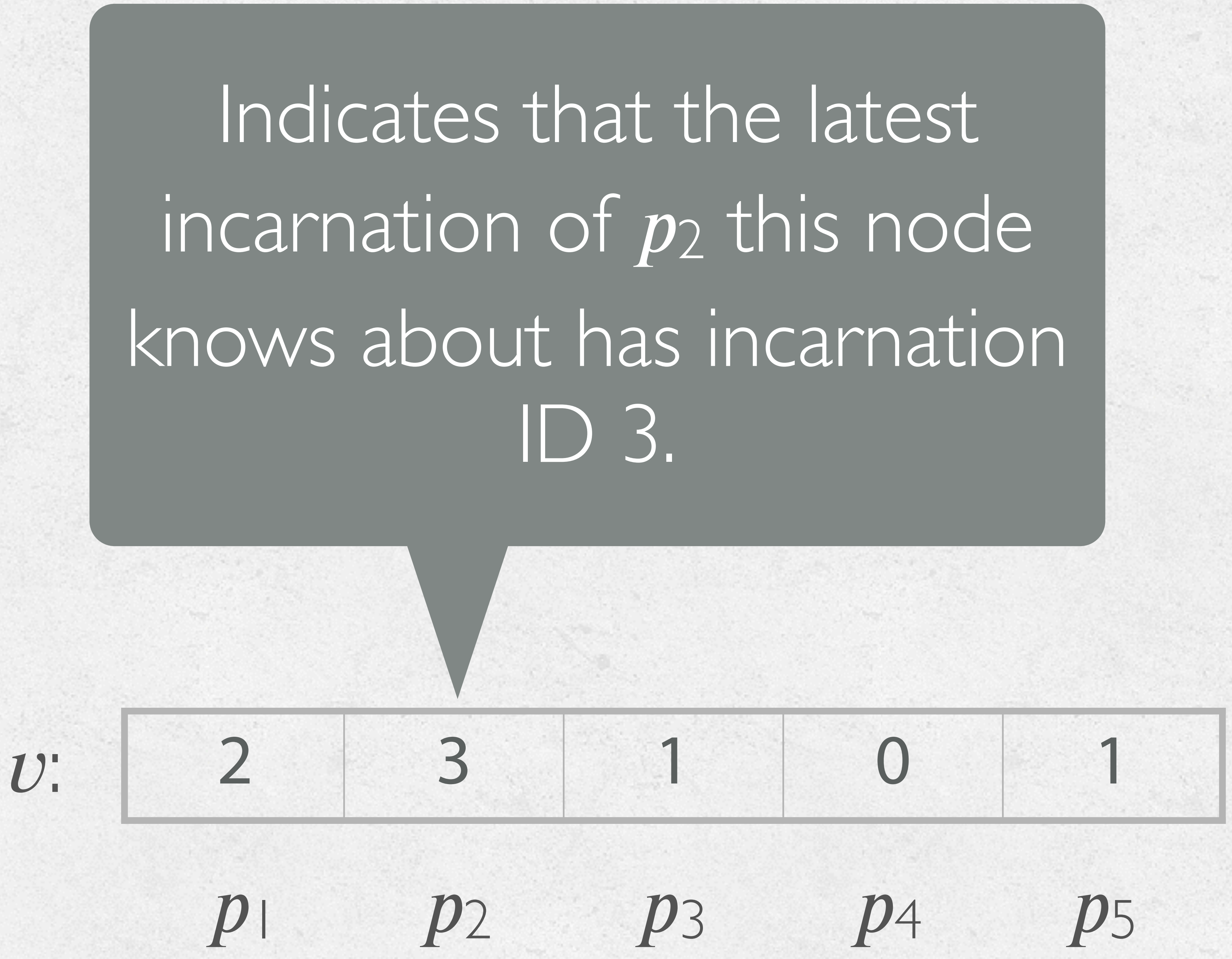
Crash Vectors

- Each node maintains a **crash vector**, with one entry per node containing the largest incarnation ID known for that node.
- Crash vectors are attached to set operation messages and recovery messages.
- Upon receiving messages, nodes update each entry in their local crash vector, taking the maximum of that entry and the one in the message.

	p_1	p_2	p_3	p_4	p_5
v:	2	3	1	0	1

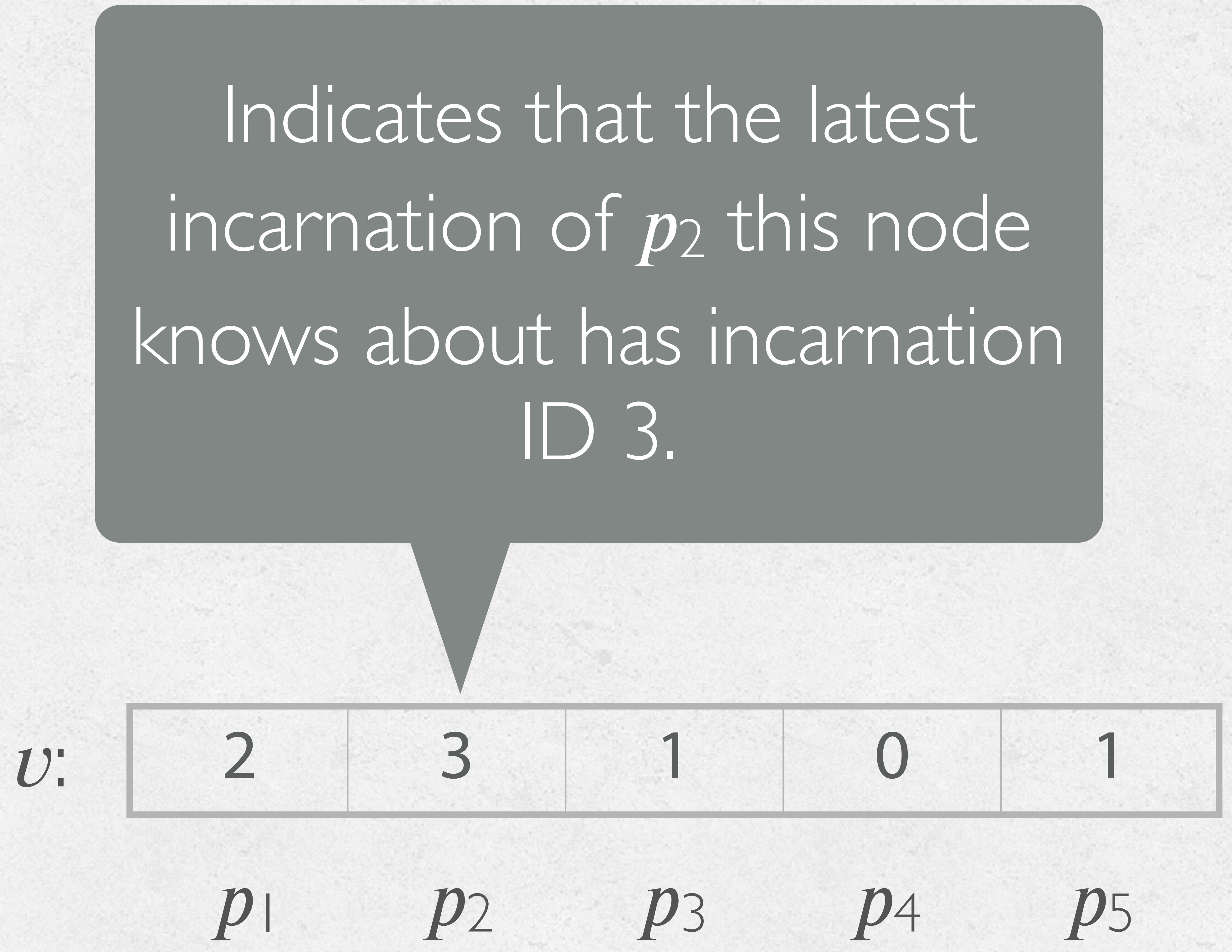
Crash Vectors

- Each node maintains a **crash vector**, with one entry per node containing the largest incarnation ID known for that node.
- Crash vectors are attached to set operation messages and recovery messages.
- Upon receiving messages, nodes update each entry in their local crash vector, taking the maximum of that entry and the one in the message.



Crash Vectors

- Each node maintains a **crash vector**, with one entry per node containing the largest incarnation ID known for that node.
- Crash vectors are attached to set operation messages and recovery messages.
- Upon receiving messages, nodes update each entry in their local crash vector, taking the maximum of that entry and the one in the message.
- *This works just like a vector clock!*



CRASH CONSISTENCY

- Two messages with crash vectors v_1 and v_2, sent by p_1 and p_2, respectively, are **crash-consistent** if:

  $$v_1[p_2] \leq v_2[p_2] \text{ and } v_2[p_1] \leq v_1[p_1]$$

  That is, if p_1 doesn't know about a later incarnation of p_2 and vice-versa.

- Similarly, a set of messages is crash-consistent if they are pairwise crash-consistent.

- Again, this works exactly like a vector clock.

STATUS

At any given time, each node has one of three statuses: **OPERATIONAL**, **DOWN**, or **RECOVERING**.

- A node that is DOWN has crashed and has not yet begun recovery.
- A node that is RECOVERING has restarted but hasn't yet finished the recovery procedure.
- A node that is OPERATIONAL has either never crashed or has finished recovering from its most recent crash.

Protocol in a Slide

Persistent local state:

n (number of nodes)

i (node ID)

Volatile local state:

v (local crash vector)

σ (current status)

S (local set)

- Nodes update their crash vectors when receiving a message.
- Nodes can match requests with replies.

Protocol in a Slide

write(s)

Persistent local state:

n (number of nodes)

i (node ID)

Volatile local state:

v (local crash vector)

σ (current status)

S (local set)

- Nodes update their crash vectors when receiving a message.
- Nodes can match requests with replies.

Protocol in a Slide

write(s)

- Sends $\langle$WRITE, v, $s\rangle$ to all nodes

Persistent local state:

n (number of nodes)

i (node ID)

Volatile local state:

v (local crash vector)

σ (current status)

S (local set)

- Nodes update their crash vectors when receiving a message.
- Nodes can match requests with replies.

Protocol in a Slide

write(s)

- Sends $\langle$WRITE, v, $s\rangle$ to all nodes
- Currently operational nodes add s to their local set, respond with $\langle$WRITE-REPLY, v, $s\rangle$

Persistent local state:

n (number of nodes)

i (node ID)

Volatile local state:

v (local crash vector)

σ (current status)

S (local set)

- Nodes update their crash vectors when receiving a message.
- Nodes can match requests with replies.

Protocol in a Slide

write(s)

- Sends $\langle$WRITE, v, $s\rangle$ to all nodes
- Currently operational nodes add s to their local set, respond with $\langle$WRITE-REPLY, v, $s\rangle$
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

Persistent local state:

n (number of nodes)

i (node ID)

Volatile local state:

v (local crash vector)

σ (current status)

S (local set)

- Nodes update their crash vectors when receiving a message.
- Nodes can match requests with replies.

Protocol in a Slide

write(s)

- Sends ⟨WRITE, v, s⟩ to all nodes
- Currently operational nodes add s to their local set, respond with ⟨WRITE-REPLY, v, s⟩
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

read()

Persistent local state:

n (number of nodes)

i (node ID)

Volatile local state:

v (local crash vector)

σ (current status)

S (local set)

- Nodes update their crash vectors when receiving a message.
- Nodes can match requests with replies.

Protocol in a Slide

write(s)

- Sends $\langle$WRITE, v, $s\rangle$ to all nodes
- Currently operational nodes add s to their local set, respond with $\langle$WRITE-REPLY, v, $s\rangle$
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

read()

- Simply returns its local set

Persistent local state:

n (number of nodes)

i (node ID)

Volatile local state:

v (local crash vector)

σ (current status)

S (local set)

- Nodes update their crash vectors when receiving a message.
- Nodes can match requests with replies.

Protocol in a Slide

write(s)

- Sends ⟨WRITE, v, s⟩ to all nodes
- Currently operational nodes add s to their local set, respond with ⟨WRITE-REPLY, v, s⟩
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

read()

- Simply returns its local set

Upon Recovery (all nodes)

Persistent local state:

n (number of nodes)

i (node ID)

Volatile local state:

v (local crash vector)

σ (current status)

S (local set)

- Nodes update their crash vectors when receiving a message.
- Nodes can match requests with replies.

Protocol in a Slide

write(s)

- Sends $\langle$WRITE, v, $s\rangle$ to all nodes
- Currently operational nodes add s to their local set, respond with $\langle$WRITE-REPLY, v, $s\rangle$
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

read()

- Simply returns its local set

Upon Recovery (all nodes)

- Sets its entry in its local crash vector to be its new incarnation ID

Persistent local state:

n (number of nodes)

i (node ID)

Volatile local state:

v (local crash vector)

σ (current status)

S (local set)

- Nodes update their crash vectors when receiving a message.
- Nodes can match requests with replies.

PROTOCOL IN A SLIDE

write(s)

- Sends $\langle$WRITE, v, $s\rangle$ to all nodes
- Currently operational nodes add s to their local set, respond with $\langle$WRITE-REPLY, v, $s\rangle$
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

read()

- Simply returns its local set

Upon Recovery (all nodes)

- Sets its entry in its local crash vector to be its new incarnation ID
- Sends $\langle$RECOVER, $v\rangle$ to all nodes.

Persistent local state:

n (number of nodes)

i (node ID)

Volatile local state:

v (local crash vector)

σ (current status)

S (local set)

- Nodes update their crash vectors when receiving a message.
- Nodes can match requests with replies.

Protocol in a Slide

write(s)

- Sends ⟨WRITE, v, s⟩ to all nodes
- Currently operational nodes add s to their local set, respond with ⟨WRITE-REPLY, v, s⟩
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

read()

- Simply returns its local set

Upon Recovery (all nodes)

- Sets its entry in its local crash vector to be its new incarnation ID
- Sends ⟨RECOVER, v⟩ to all nodes.
- Operational nodes reply with ⟨RECOVER-REPLY, v, S⟩.

Persistent local state:

n (number of nodes)

i (node ID)

Volatile local state:

v (local crash vector)

σ (current status)

S (local set)

- Nodes update their crash vectors when receiving a message.
- Nodes can match requests with replies.

PROTOCOL IN A SLIDE

write(s)

- Sends ⟨WRITE, v, s⟩ to all nodes
- Currently operational nodes add s to their local set, respond with ⟨WRITE-REPLY, v, s⟩
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

read()

- Simply returns its local set

Upon Recovery (all nodes)

- Sets its entry in its local crash vector to be its new incarnation ID
- Sends ⟨RECOVER, v⟩ to all nodes.
- Operational nodes reply with ⟨RECOVER-REPLY, v, S⟩.
- Waits for a crash-consistent set of replies from a majority, re-sending when necessary. Then sets its local S to be the union of the replies and uses the **write** procedure to write-back the read values (for monotonicity).

Persistent local state:

n (number of nodes)

i (node ID)

Volatile local state:

v (local crash vector)

σ (current status)

S (local set)

- Nodes update their crash vectors when receiving a message.
- Nodes can match requests with replies.

Protocol in a Slide

write(s)

- Sends $\langle$WRITE, v, $s\rangle$ to all nodes
- Currently operational nodes add s to their local set, respond with $\langle$WRITE-REPLY, v, $s\rangle$
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

read()

- Simply returns its local set

Upon Recovery (all nodes)

- Sets its entry in its local crash vector to be its new incarnation ID
- Sends $\langle$RECOVER, $v\rangle$ to all nodes.
- Operational nodes reply with $\langle$RECOVER-REPLY, v, $S\rangle$.
- Waits for a crash-consistent set of replies from a majority, re-sending when necessary. Then sets its local S to be the union of the replies and uses the **write** procedure to write-back the read values (for monotonicity).
- Finally, sets σ to OPERATIONAL, declaring the end of recovery.

Persistent local state:

n (number of nodes)

i (node ID)

Volatile local state:

v (local crash vector)

σ (current status)

S (local set)

- Nodes update their crash vectors when receiving a message.
- Nodes can match requests with replies.

Is it Correct?

Quorum Knowledge

- We say that quorum Q **knows** X if, for all nodes $p \in Q$:

  1. p is DOWN,
  2. p is OPERATIONAL and knows X, or
  3. p is RECOVERING and *guaranteed to know* X *upon finishing recovery* (if it doesn't crash again first).

- We are concerned with two kinds of quorum knowledge: knowledge of **written values** and knowledge of **incarnation IDs**.

Persistence of Quorum Knowledge

Suppose quorum Q knows X (either an incarnation ID or written value). We will show this knowledge persists by induction.

write(s)

- Sends $\langle$WRITE, v, $s\rangle$ to all nodes
- Currently operational nodes add s to their local set, respond with $\langle$WRITE-REPLY, v, $s\rangle$
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

read()

- Simply returns its local set

Upon Recovery (all nodes)

- Sets its entry in its local crash vector to be its new incarnation ID
- Sends $\langle$RECOVER, v, $s\rangle$ to all nodes.
- Operational nodes reply with $\langle$RECOVER-REPLY, v, $S\rangle$.
- Waits for a crash-consistent set of replies from a majority, re-sending when necessary. Then sets its local S to be the union of the replies and uses the **write** procedure to write-back the read values (for monotonicity).
- Finally, sets σ to OPERATIONAL, declaring the end of recovery.

PERSISTENCE OF QUORUM KNOWLEDGE

Suppose quorum Q knows X (either an incarnation ID or written value). We will show this knowledge persists by induction.

- The only step a node can take to falsify our invariant is beginning recovery.

write(s)

- Sends $\langle$WRITE, v, $s\rangle$ to all nodes
- Currently operational nodes add s to their local set, respond with $\langle$WRITE-REPLY, v, $s\rangle$
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

read()

- Simply returns its local set

Upon Recovery (all nodes)

- Sets its entry in its local crash vector to be its new incarnation ID
- Sends $\langle$RECOVER, v, $s\rangle$ to all nodes.
- Operational nodes reply with $\langle$RECOVER-REPLY, v, $S\rangle$.
- Waits for a crash-consistent set of replies from a majority, re-sending when necessary. Then sets its local S to be the union of the replies and uses the **write** procedure to write-back the read values (for monotonicity).
- Finally, sets σ to OPERATIONAL, declaring the end of recovery.

PERSISTENCE OF QUORUM KNOWLEDGE

Suppose quorum Q knows X (either an incarnation ID or written value). We will show this knowledge persists by induction.

- The only step a node can take to falsify our invariant is beginning recovery.
- If that node manages to eventually recover, it must use a quorum of OPERATIONAL nodes.

write(s)

- Sends $\langle$WRITE, v, $s\rangle$ to all nodes
- Currently operational nodes add s to their local set, respond with $\langle$WRITE-REPLY, v, $s\rangle$
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

read()

- Simply returns its local set

Upon Recovery (all nodes)

- Sets its entry in its local crash vector to be its new incarnation ID
- Sends $\langle$RECOVER, v, $s\rangle$ to all nodes.
- Operational nodes reply with $\langle$RECOVER-REPLY, v, $S\rangle$.
- Waits for a crash-consistent set of replies from a majority, re-sending when necessary. Then sets its local S to be the union of the replies and uses the **write** procedure to write-back the read values (for monotonicity).
- Finally, sets σ to OPERATIONAL, declaring the end of recovery.

Persistence of Quorum Knowledge

Suppose quorum Q knows X (either an incarnation ID or written value). We will show this knowledge persists by induction.

- The only step a node can take to falsify our invariant is beginning recovery.
- If that node manages to eventually recover, it must use a quorum of OPERATIONAL nodes.
- That quorum will contain at least one node from Q, which by induction knows X.

write(s)

- Sends $\langle$WRITE, v, $s\rangle$ to all nodes
- Currently operational nodes add s to their local set, respond with $\langle$WRITE-REPLY, v, $s\rangle$
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

read()

- Simply returns its local set

Upon Recovery (all nodes)

- Sets its entry in its local crash vector to be its new incarnation ID
- Sends $\langle$RECOVER, v, $s\rangle$ to all nodes.
- Operational nodes reply with $\langle$RECOVER-REPLY, v, $S\rangle$.
- Waits for a crash-consistent set of replies from a majority, re-sending when necessary. Then sets its local S to be the union of the replies and uses the **write** procedure to write-back the read values (for monotonicity).
- Finally, sets σ to OPERATIONAL, declaring the end of recovery.

Persistence of Quorum Knowledge

Suppose quorum Q knows X (either an incarnation ID or written value). We will show this knowledge persists by induction.

- The only step a node can take to falsify our invariant is beginning recovery.
- If that node manages to eventually recover, it must use a quorum of OPERATIONAL nodes.
- That quorum will contain at least one node from Q, which by induction knows X.
- Therefore, our node will learn X if/when it finishes recovery.

write(s)

- Sends $\langle$WRITE, v, $s\rangle$ to all nodes
- Currently operational nodes add s to their local set, respond with $\langle$WRITE-REPLY, v, $s\rangle$
- Writer waits for a *crash-consistent* set of replies from a majority, re-sending when necessary.

read()

- Simply returns its local set

Upon Recovery (all nodes)

- Sets its entry in its local crash vector to be its new incarnation ID
- Sends $\langle$RECOVER, v, $s\rangle$ to all nodes.
- Operational nodes reply with $\langle$RECOVER-REPLY, v, $S\rangle$.
- Waits for a crash-consistent set of replies from a majority, re-sending when necessary. Then sets its local S to be the union of the replies and uses the **write** procedure to write-back the read values (for monotonicity).
- Finally, sets σ to OPERATIONAL, declaring the end of recovery.

Acquisition of Quorum Knowledge (I)

- We want to show that a crash-consistent set of replies implies quorum knowledge.
- For writes, we want to know that the value being written was persisted.
- On recovery, we want to know that our new (larger) incarnation ID was persisted.

Acquisition of Quorum Knowledge (II)

We will prove this by induction.

Invariant 1: If a node receives a crash-consistent set of replies for X from quorum Q, then Q knows X.

Invariant 2: If node p *ever* sent a reply that makes it into a crash-consistent set of replies for X from a quorum, and p is currently OPERATIONAL, then p knows X.

Acquisition of Quorum Knowledge (III)

Suppose p received a crash-consistent set of replies from Q.

Invariant 1: If a node receives a crash-consistent set of replies for X from quorum Q, then Q knows X.

Invariant 2: If node p *ever* sent a reply that makes it into a crash-consistent set of replies for X from a quorum, and p is currently OPERATIONAL, then p knows X.

Acquisition of Quorum Knowledge (III)

Suppose p received a crash-consistent set of replies from Q.

Then, for all nodes in Q, *at the time they sent their replies*, they hadn't yet helped any other node in Q recover to a later incarnation (than the one that sent the reply).

Invariant 1: If a node receives a crash-consistent set of replies for X from quorum Q, then Q knows X.

Invariant 2: If node p *ever* sent a reply that makes it into a crash-consistent set of replies for X from a quorum, and p is currently OPERATIONAL, then p knows X.

Acquisition of Quorum Knowledge (III)

Suppose p received a crash-consistent set of replies from Q.

Then, for all nodes in Q, *at the time they sent their replies*, they hadn't yet helped any other node in Q recover to a later incarnation (than the one that sent the reply).

Otherwise, by induction, it would have known about that later incarnation and the replies wouldn't be crash-consistent.

Invariant 1: If a node receives a crash-consistent set of replies for X from quorum Q, then Q knows X.

Invariant 2: If node p *ever* sent a reply that makes it into a crash-consistent set of replies for X from a quorum, and p is currently OPERATIONAL, then p knows X.

ACQUISITION OF QUORUM KNOWLEDGE (IV)

Now, suppose for the sake of contradiction, that a node in $p \in Q$ is OPERATIONAL but does not know X.

Invariant 1: If a node receives a crash-consistent set of replies for X from quorum Q, then Q knows X.

Invariant 2: If node p *ever* sent a reply that makes it into a crash-consistent set of replies for X from a quorum, and p is currently OPERATIONAL, then p knows X.

Acquisition of Quorum Knowledge (IV)

Now, suppose for the sake of contradiction, that a node in $p \in Q$ is OPERATIONAL but does not know X.

It must have crashed and recovered since sending its reply. At least one of the nodes that helped it recover must have been in Q (by quorum intersection). Call that node r.

Invariant 1: If a node receives a crash-consistent set of replies for X from quorum Q, then Q knows X.

Invariant 2: If node p *ever* sent a reply that makes it into a crash-consistent set of replies for X from a quorum, and p is currently OPERATIONAL, then p knows X.

Acquisition of Quorum Knowledge (IV)

Now, suppose for the sake of contradiction, that a node in $p \in Q$ is OPERATIONAL but does not know X.

It must have crashed and recovered since sending its reply. At least one of the nodes that helped it recover must have been in Q (by quorum intersection). Call that node r.

As we just showed, r couldn't have participated in the recovery before sending its own reply. But then by induction, it would have known X during the recovery, implying that p now knows X.

Invariant 1: If a node receives a crash-consistent set of replies for X from quorum Q, then Q knows X.

Invariant 2: If node p *ever* sent a reply that makes it into a crash-consistent set of replies for X from a quorum, and p is currently OPERATIONAL, then p knows X.

Acquisition of Quorum Knowledge (IV)

Now, suppose for the sake of contradiction, that a node in $p \in Q$ is OPERATIONAL but does not know X.

It must have crashed and recovered since sending its reply. At least one of the nodes that helped it recover must have been in Q (by quorum intersection). Call that node r.

As we just showed, r couldn't have participated in the recovery before sending its own reply. But then by induction, it would have known X during the recovery, implying that p now knows X.

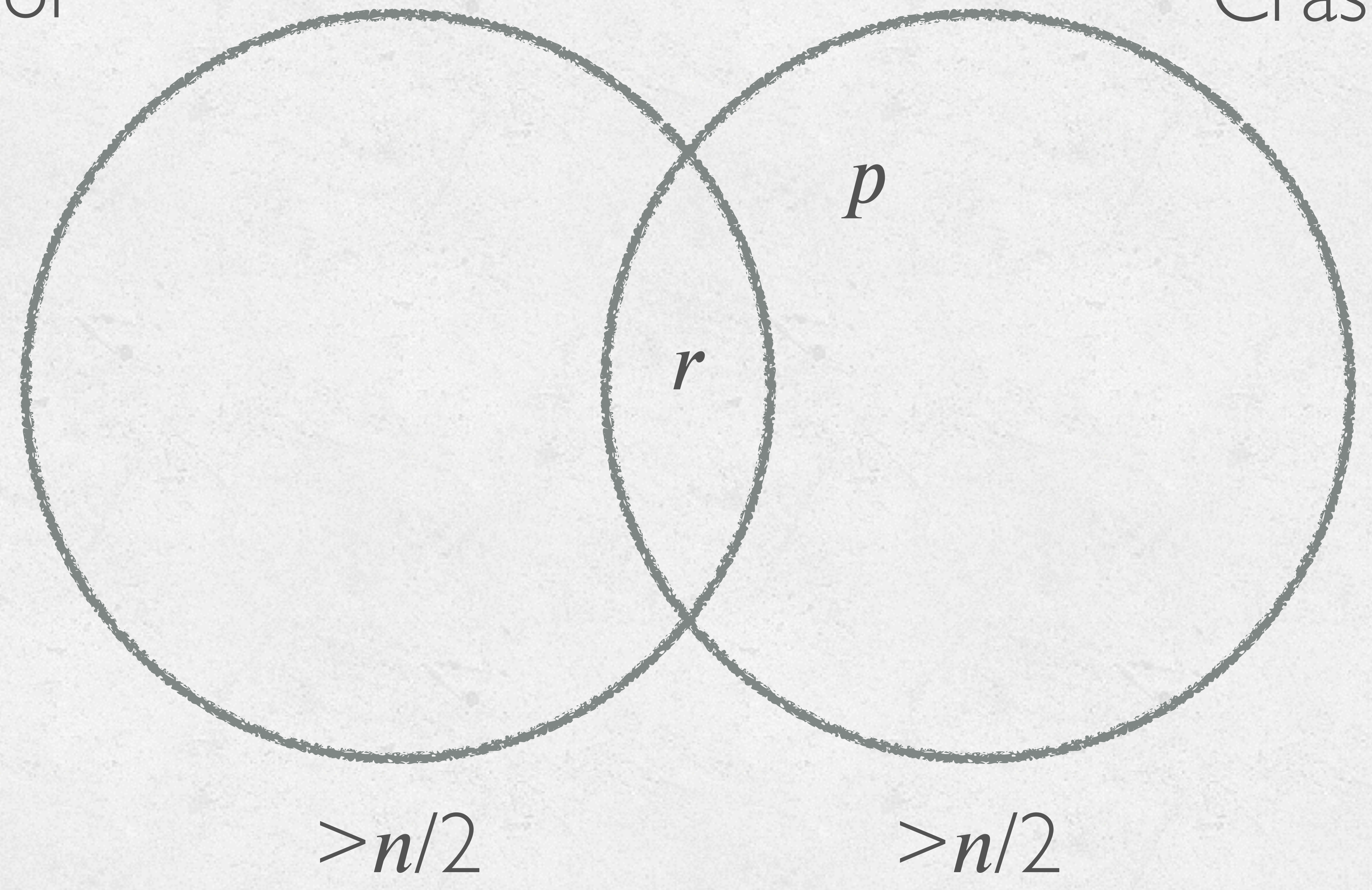
Contradiction. **Invariant 1 holds**.

Invariant 1: If a node receives a crash-consistent set of replies for X from quorum Q, then Q knows X.

Invariant 2: If node p *ever* sent a reply that makes it into a crash-consistent set of replies for X from a quorum, and p is currently OPERATIONAL, then p knows X.

Crash-consistent replies for *p*'s recovery

Crash-consistent replies for *X*

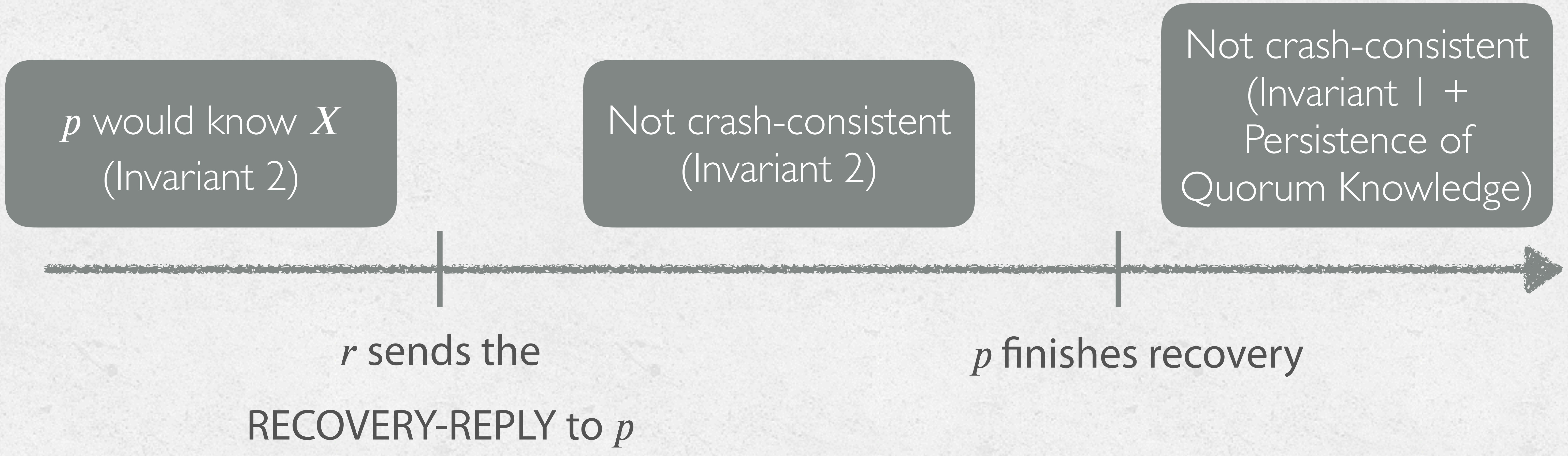


ACQUISITION OF QUORUM KNOWLEDGE (V)

Next, suppose that p sent a reply that will make it into a crash-consistent set of replies and is currently operational but doesn't know X.

Again, it must have crashed and recovered, and there must be some node in the intersection of those that it recovered from and the other nodes that sent replies, r.

When could r send its reply for X?



Invariant 1: If a node receives a crash-consistent set of replies for X from quorum Q, then Q knows X.

Invariant 2: If node p *ever* sent a reply that made it into a crash-consistent set of replies for X from a quorum, and p is currently OPERATIONAL, then p knows X.

Completed writes ensure Quorum Knowledge, which is persistent!

What Did We Accomplish?

- We created a set which a writer can store values in. Upon recovery, the writer will re-learn the values of all completed writes.
- We can run n copies of this protocol in parallel, one for each server.
- ***Any protocol*** which is safe when you assume stable storage will be safe when disk-based storage is replaced with our diskless storage.
- Furthermore, you could recover from disk normally and only recover from diskless storage when disks fail.
- If latency between nodes is small, writing to diskless storage is *faster*.
- But what about liveness?

When Diskless Storage is Live

- Obviously, if *all servers crash simultaneously* and you don't have stable storage, you're stuck. If you have stable storage, you could still recover (depending on what you were storing).
- So, we can't hope for the same liveness guarantees. Moreover, specifying *exactly* when this protocol can make progress is hard.
- If, for all points in time, there is always *some quorum* that is operational, we can make progress.
- If there exists some particular quorum that stays OPERATIONAL long enough, then our diskless stable storage algorithm is **wait-free**! We don't need consensus!

Questions?