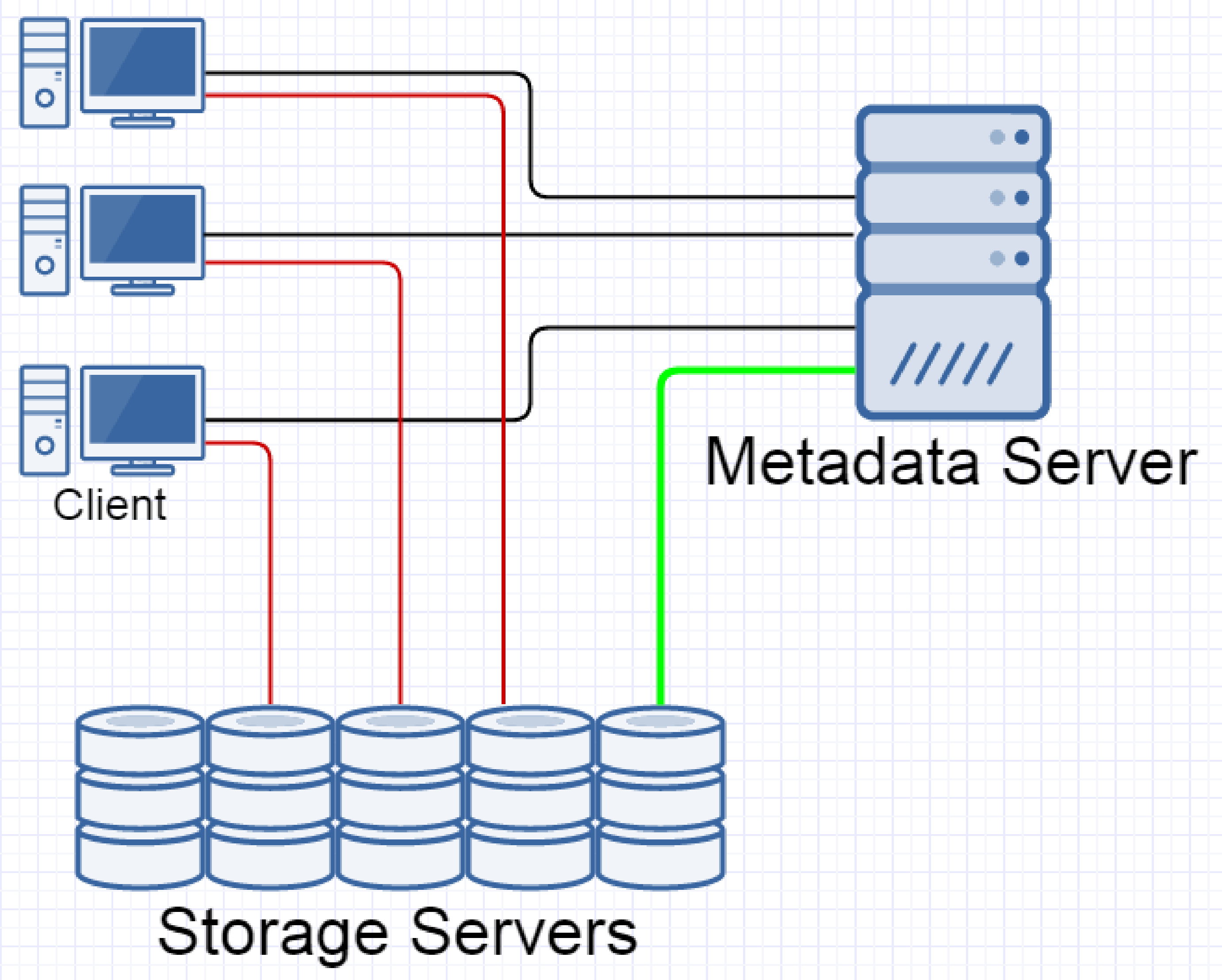
The basics of file systems

Neapolis University Pafos

Consensus in a distributed system

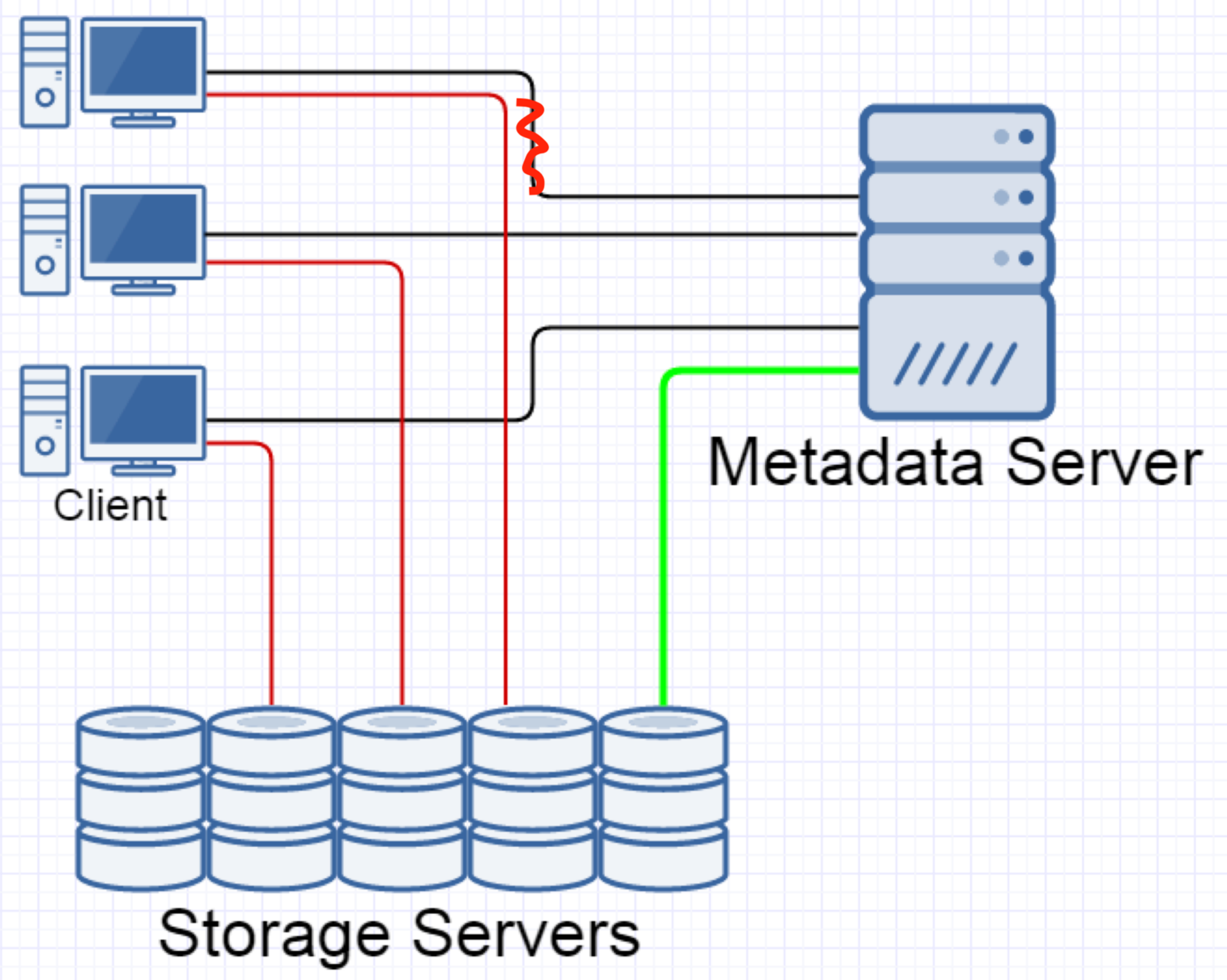
Today we are going to see how to implement a reliable distributed FSM.

A reminder about Parallel NFS



When a client opens a file, the metadata server sends it a "file layout". It described which storage server contains the file, and what protocol that server uses.

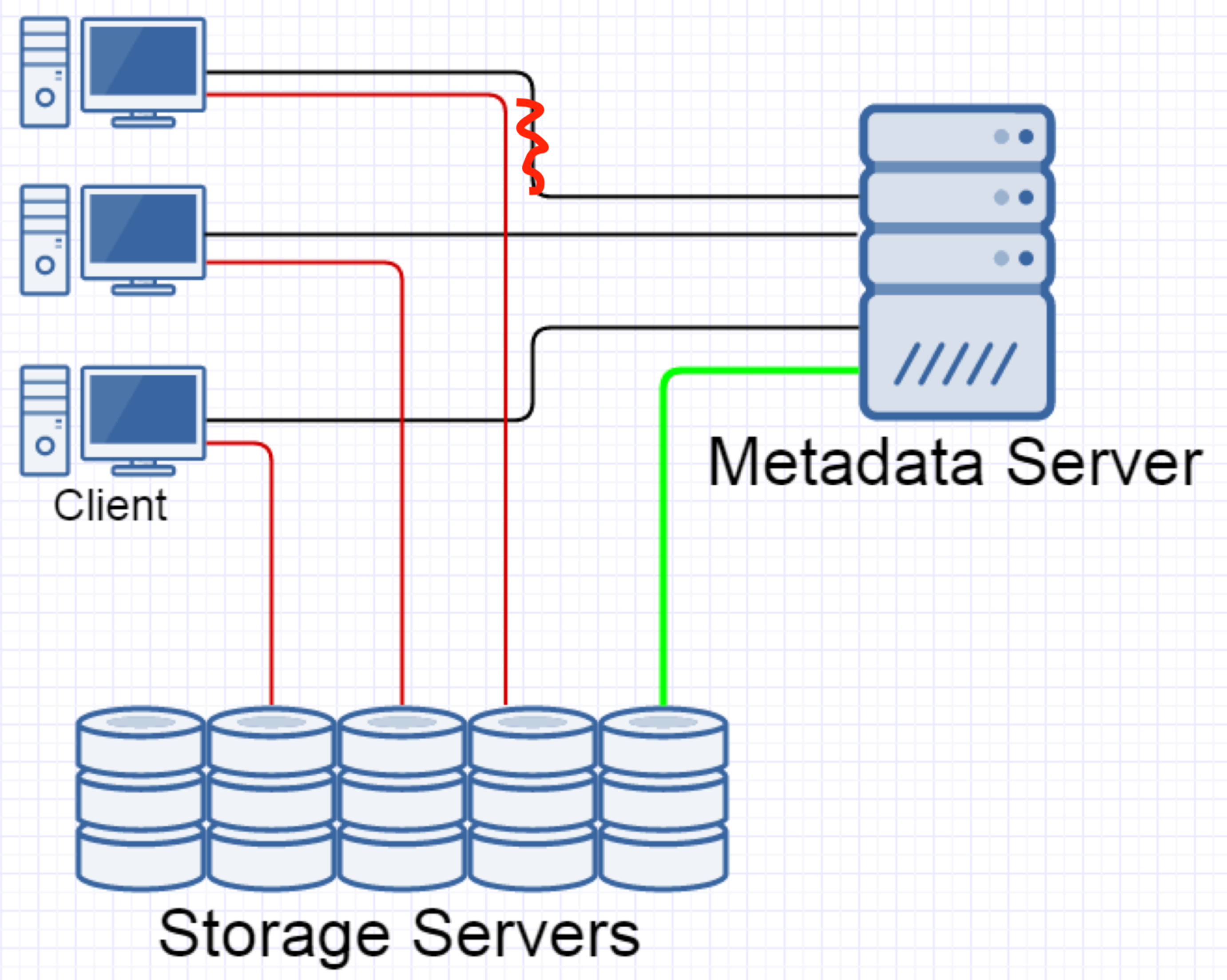
A reminder about Parallel NFS



When a client opens a file, the metadata server sends it a “file layout”. It described which storage server contains the file, and what protocol that server uses.

Question: how do we handle a client that lost connection to the metadata server but can still communicate to the storage server? It is not admissible to have two clients modify the same file without synchronising with each other.

A reminder about Parallel NFS

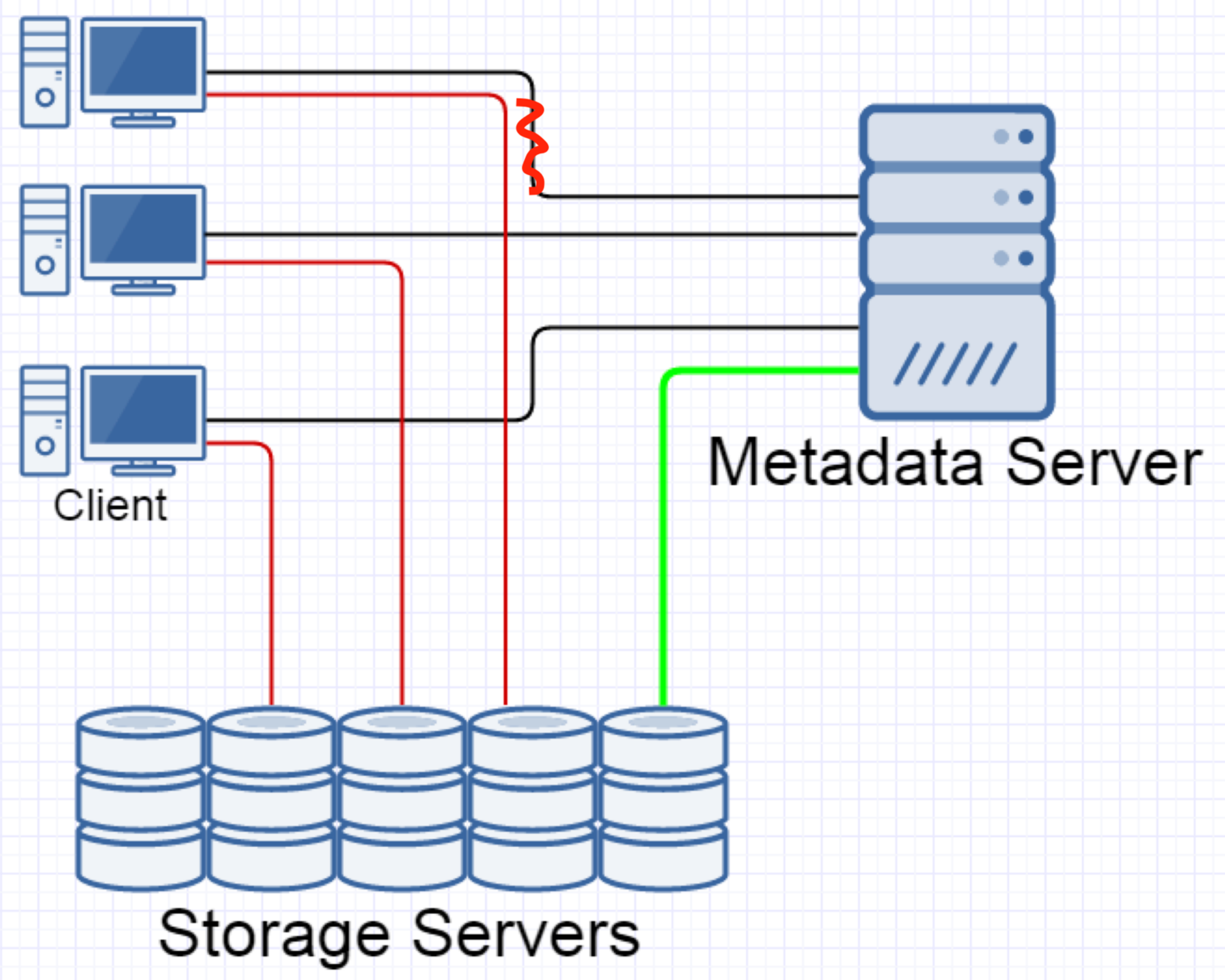


When a client opens a file, the metadata server sends it a "file layout". It described which storage server contains the file, and what protocol that server uses.

Question: how do we handle a client that lost connection to the metadata server but can still communicate to the storage server? It is not admissible to have two clients modify the same file without synchronising with each other.

A solution by NFSv4.1: the metadata server assigns a layout lease to each file layout. A layout lease has a period when the client may use the layout. If the client does not renew its layout lease, then the storage server stops accepting requests that the stale layout.

A reminder about Parallel NFS



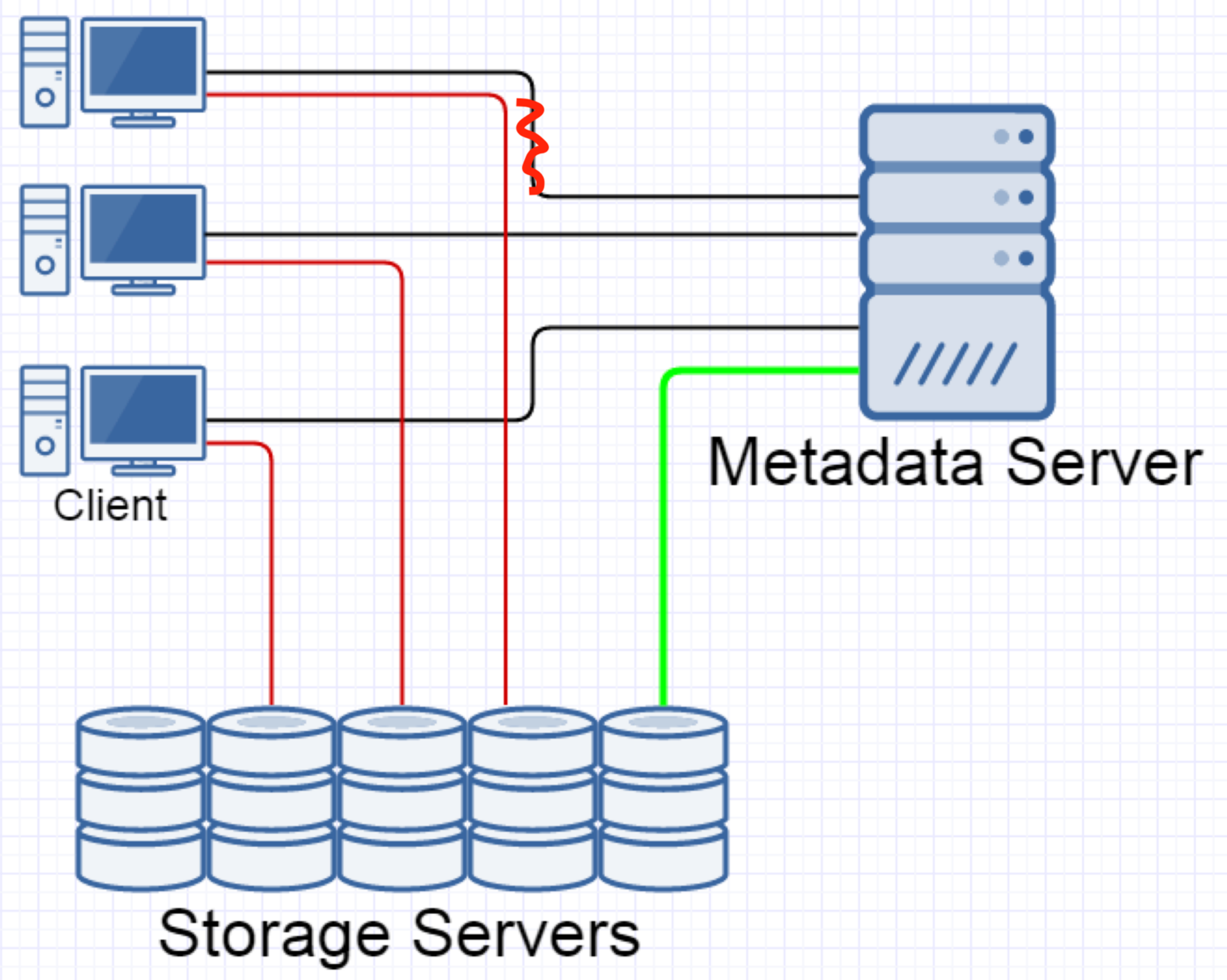
When a client opens a file, the metadata server sends it a “file layout”. It described which storage server contains the file, and what protocol that server uses.

Question: how do we handle a client that lost connection to the metadata server but can still communicate to the storage server? It is not admissible to have two clients modify the same file without synchronising with each other.

A solution by NFSv4.1: the metadata server assigns a layout lease to each file layout. A layout lease has a period when the client may use the layout. If the client does not renew its layout lease, then the storage server stops accepting requests that the stale layout.

Quiz: layout lease may not rely on clock of different clients being synchronised. How can we implement that?

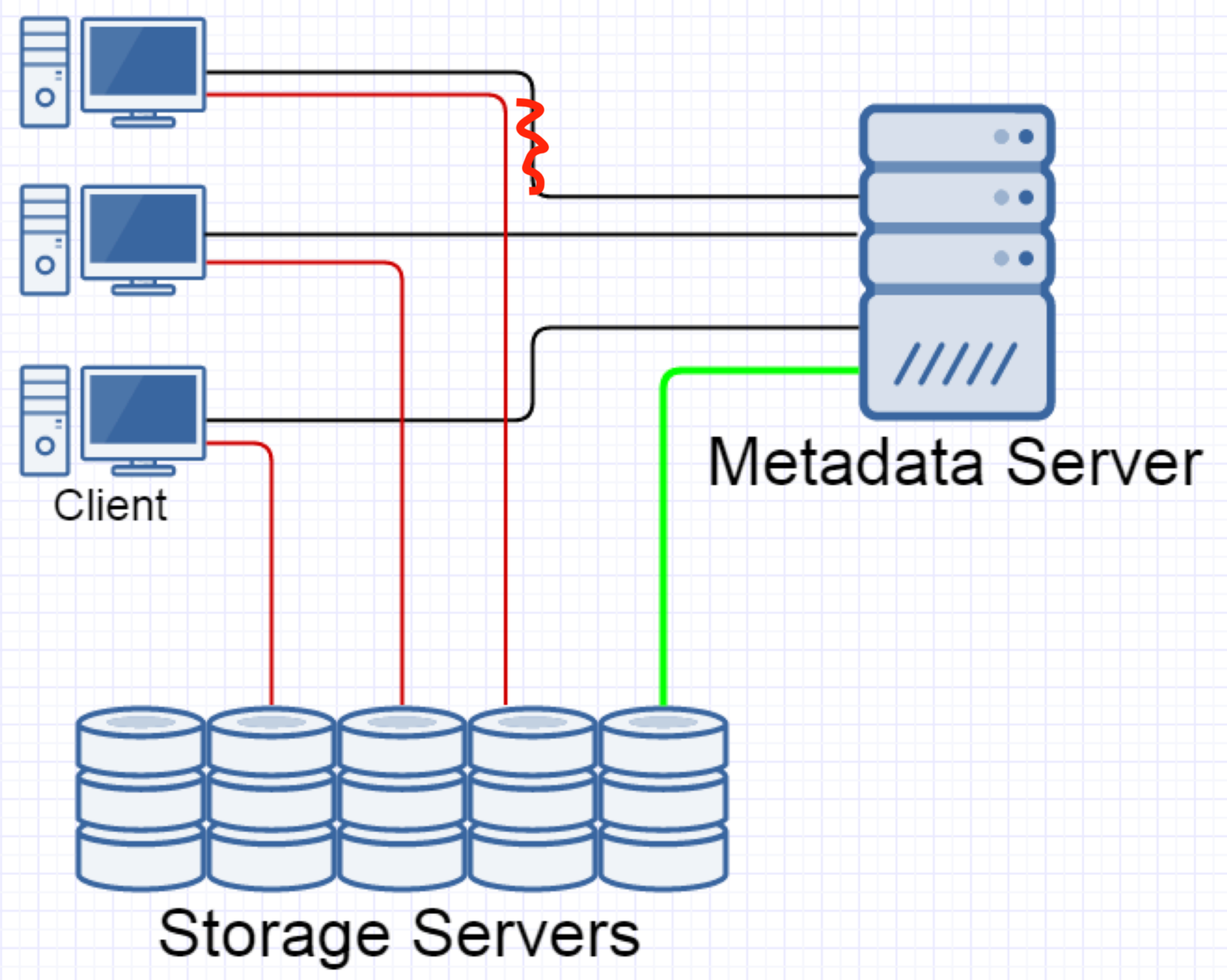
A reminder about Parallel NFS



When a client opens a file, the metadata server sends it a “file layout”. It described which storage server contains the file, and what protocol that server uses.

Quiz: validating layout leases in storage servers appears to be an unnecessary complication. A NFS client may stop issuing requests once its lease time out. Is this right?

A reminder about Parallel NFS

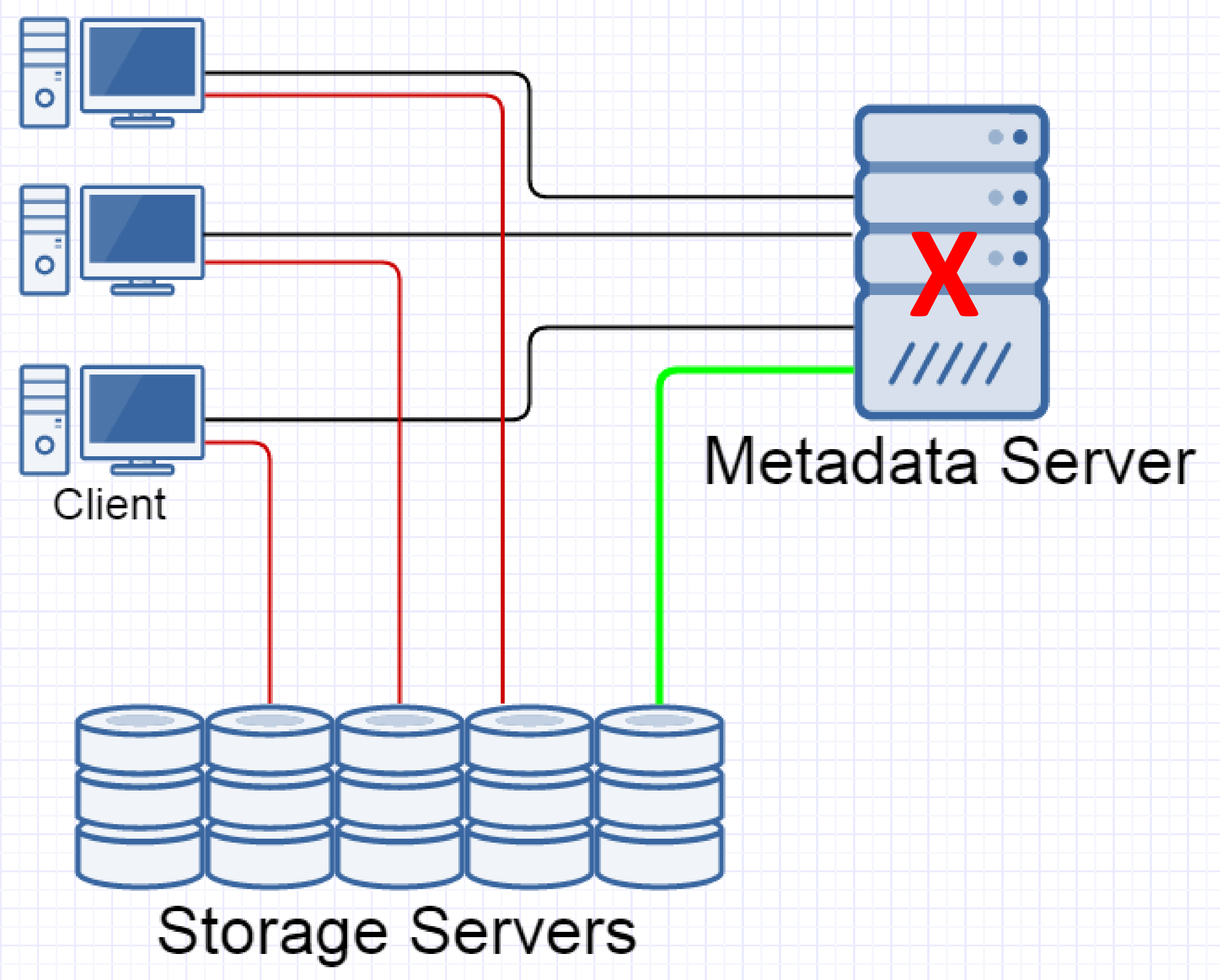


When a client opens a file, the metadata server sends it a "file layout". It described which storage server contains the file, and what protocol that server uses.

Quiz: validating layout leases in storage servers appears to be an unnecessary complication. A NFS client may stop issuing requests once its lease time out. Is this right?

See also: client fencing.

A reminder about Parallel NFS



The metadata server is a ”single point of failure” of this system.

Question: how can we replicate the state of the metadata server to multiple machines and have them all agree on ”the state” of the metadata server?

Consensus in a distributed system

Original problem: how to replicate the state of the metadata server to multiple nodes so that all nodes agree on the same state?

Consensus in a distributed system

Original problem: how to replicate the state of the metadata server to multiple nodes so that all nodes agree on the same state?

- We may regard the metadata server as a deterministic FSM. Its states are FS images, and its state transitions are operations like “create a file”, “delete a file”, etc.

Consensus in a distributed system

Original problem: how to replicate the state of the metadata server to multiple nodes so that all nodes agree on the same state?

- We may regard the metadata server as a deterministic FSM. Its states are FS images, and its state transitions are operations like “create a file”, “delete a file”, etc.
- At every step, the replicas of the metadata server need to agree on a state transition that they perform. Every replica must do the same transition.

Consensus in a distributed system

Original problem: how to replicate the state of the metadata server to multiple nodes so that all nodes agree on the same state?

- We may regard the metadata server as a deterministic FSM. Its states are FS images, and its state transitions are operations like “create a file”, “delete a file”, etc.
- At every step, the replicas of the metadata server need to agree on a state transition that they perform. Every replica must do the same transition.
- The state of a deterministic FSM is uniquely defined by a log of state transition. Thus, to replicate an FSM it suffices to replicate a journal.

Consensus in a distributed system

Original problem: how to replicate the state of the metadata server to multiple nodes so that all nodes agree on the same state?

- We may regard the metadata server as a deterministic FSM. Its states are FS images, and its state transitions are operations like “create a file”, “delete a file”, etc.
- At every step, the replicas of the metadata server need to agree on a state transition that they perform. Every replica must do the same transition.
- The state of a deterministic FSM is uniquely defined by a log of state transition. Thus, to replicate an FSM it suffices to replicate a journal.
- To replicate a journal, cluster nodes need to agree on values to append to the log at every step.

Consensus in a distributed system

Original problem: how to replicate the state of the metadata server to multiple nodes so that all nodes agree on the same state?

- We may regard the metadata server as a deterministic FSM. Its states are FS images, and its state transitions are operations like “create a file”, “delete a file”, etc.
- At every step, the replicas of the metadata server need to agree on a state transition that they perform. Every replica must do the same transition.
- The state of a deterministic FSM is uniquely defined by a log of state transition. Thus, to replicate an FSM it suffices to replicate a journal.
- To replicate a journal, cluster nodes need to agree on values to append to the log at every step.
- It suffices to solve the following problem: nodes of a cluster must agree on a value to append to a journal.

The problem statement

The original problem: nodes of a cluster must agree on a value to append to a journal.

The problem statement

The original problem: nodes of a cluster must agree on a value to append to a journal.

The problem, restated: *choose** one value from a set of *proposed** values.

* *We will define the words "proposed" and "choose" later.*

The problem statement

The original problem: nodes of a cluster must agree on a value to append to a journal.

The problem, restated: *choose** one value from a set of *proposed** values.

Participants:
1. proposers,
2. acceptors,
3. learners.

** We will define the words "proposed" and "choose" later.*

The problem statement

The original problem: nodes of a cluster must agree on a value to append to a journal.

The problem, restated: *choose** one value from a set of *proposed** values.

Participants:
1. proposers,
2. acceptors,
3. learners.

Proposer processes send `propose()` requests. Their arguments are called *proposed* values.

Acceptor processes receive `propose()` requests from proposers.

Acceptor processes send `accept()` requests to learners. Their arguments are called *accepted* values.

Typically, every cluster member plays all three roles.

The problem statement

The original problem: nodes of a cluster must agree on a value to append to a journal.

The problem, restated: *choose** one value from a set of *proposed** values.

Participants:

1. proposers,
2. acceptors,
3. learners.

Proposer processes may be regarded as the source of requests “at step N, create a file F” or “at step N, grant a lease on file F to client C”.

The problem statement

The original problem: nodes of a cluster must agree on a value to append to a journal.

The problem, restated: *choose** one value from a set of *proposed** values.

Participants:
1. proposers,
2. acceptors,
3. learners.

The model of the network and process failure modes:
1. participant processes may work at arbitrary speed,
2. processes may crash and restart at any moment,
3. messages may be delayed (in particular, reordered), lost, or duplicated, but they cannot be corrupted.

Motivation for definitions and rules

We will consider a value chosen if it was *accepted* by a majority of acceptors.

Motivation for definitions and rules

We will consider a value chosen if it was *accepted* by a majority of acceptors.

This definition makes sense if the following property holds:

Requirement 0: an acceptor accepts at most one value.

In a set with N elements any two subsets with $\lfloor {}^{N}/_{2} \rfloor + 1$ elements have a non-empty intersection. Suppose M_0 and M_1 are majorities of acceptors that have accepted v_0 and v_1, respectively. An acceptor that belongs to $M_0 \cap M_1$ has accepted both values. Thus, $v_0 = v_1$ and all acceptors in $M_0 \cup M_1$ have accepted this value.

Motivation for definitions and rules

If only one proposer proposed a value, then this value must be chosen by the system.

Motivation for definitions and rules

If only one proposer proposed a value, then this value must be chosen by the system.

In this situation every acceptor receives only one `propose()` request.

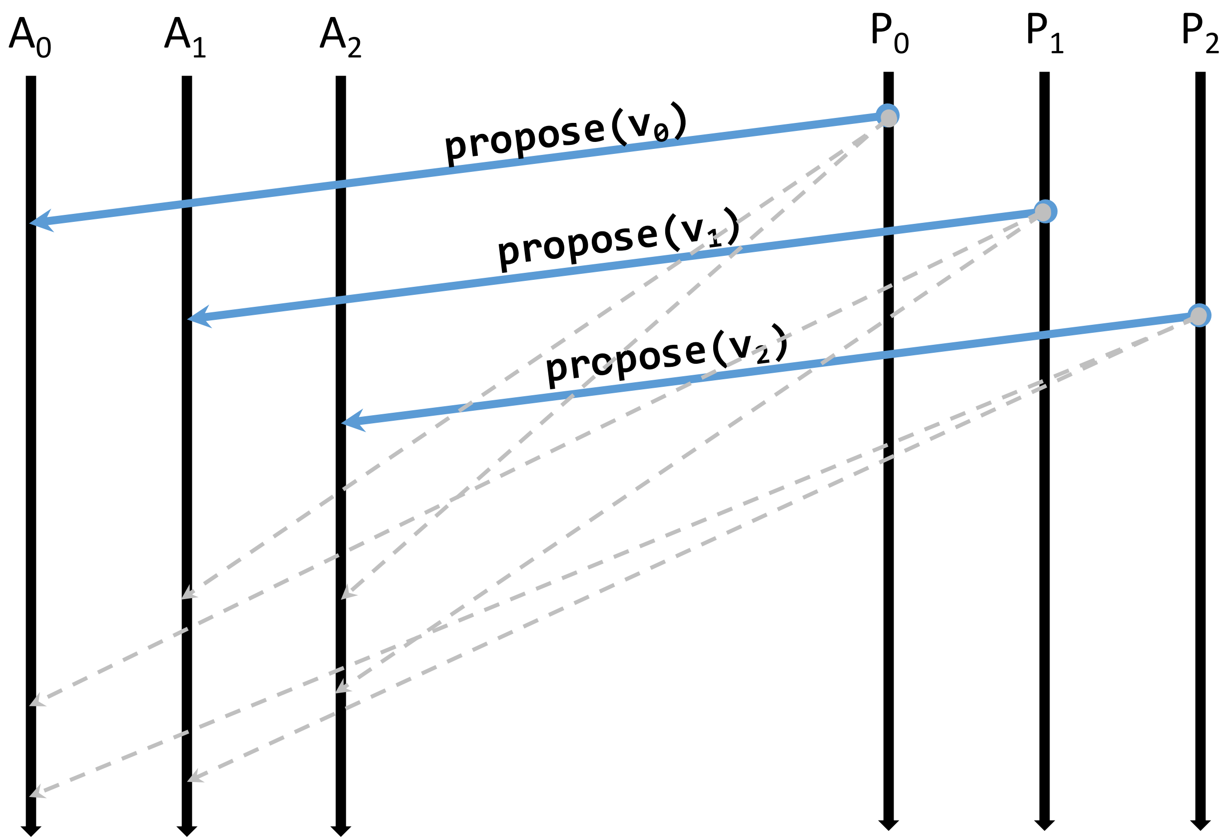
Requirement 1: an acceptor must accept the first proposed value.

Motivation for definitions and rules

If only one proposer proposed a value, then this value must be chosen by the system.

In this situation every acceptor receives only one `propose()` request.

Requirement 1: an acceptor must accept the first proposed value.

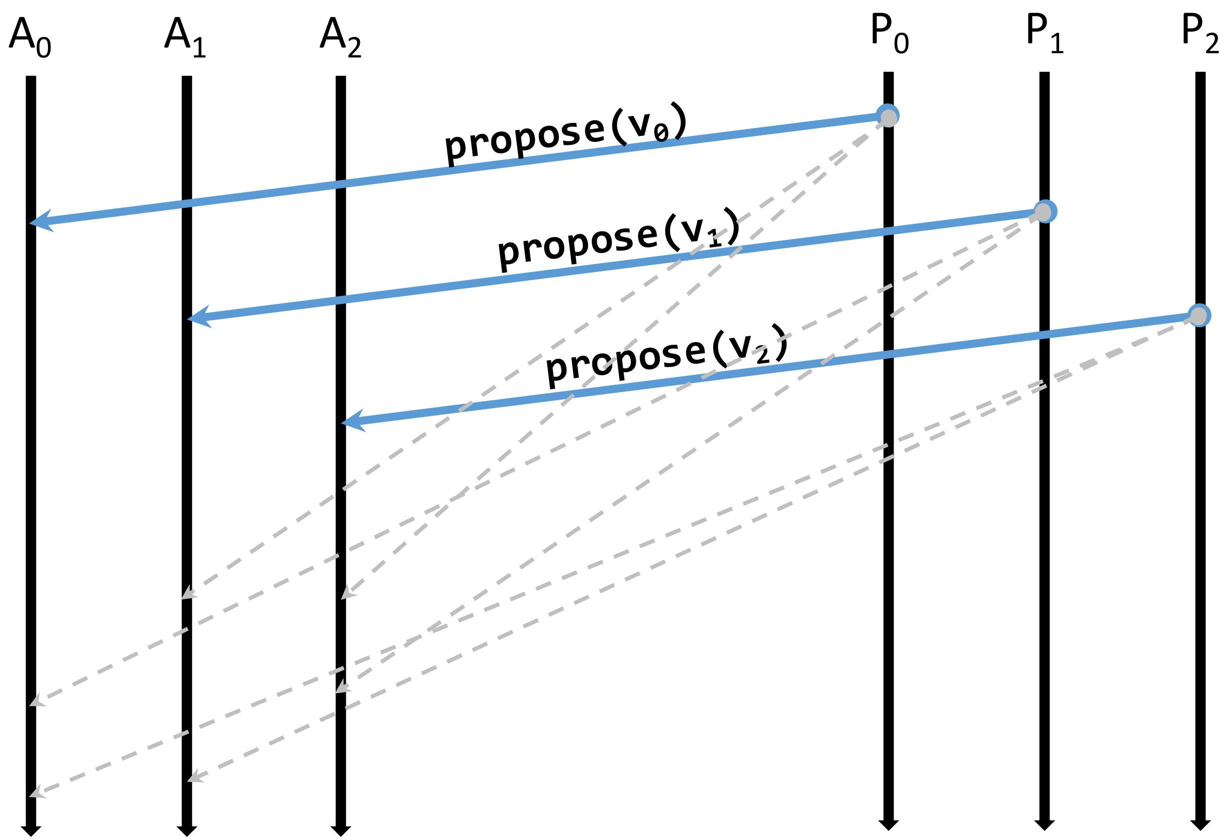


Motivation for definitions and rules

If only one proposer proposed a value, then this value must be chosen by the system.

In this situation every acceptor receives only one `propose()` request.

Requirement 1: an acceptor must accept the first proposed value.



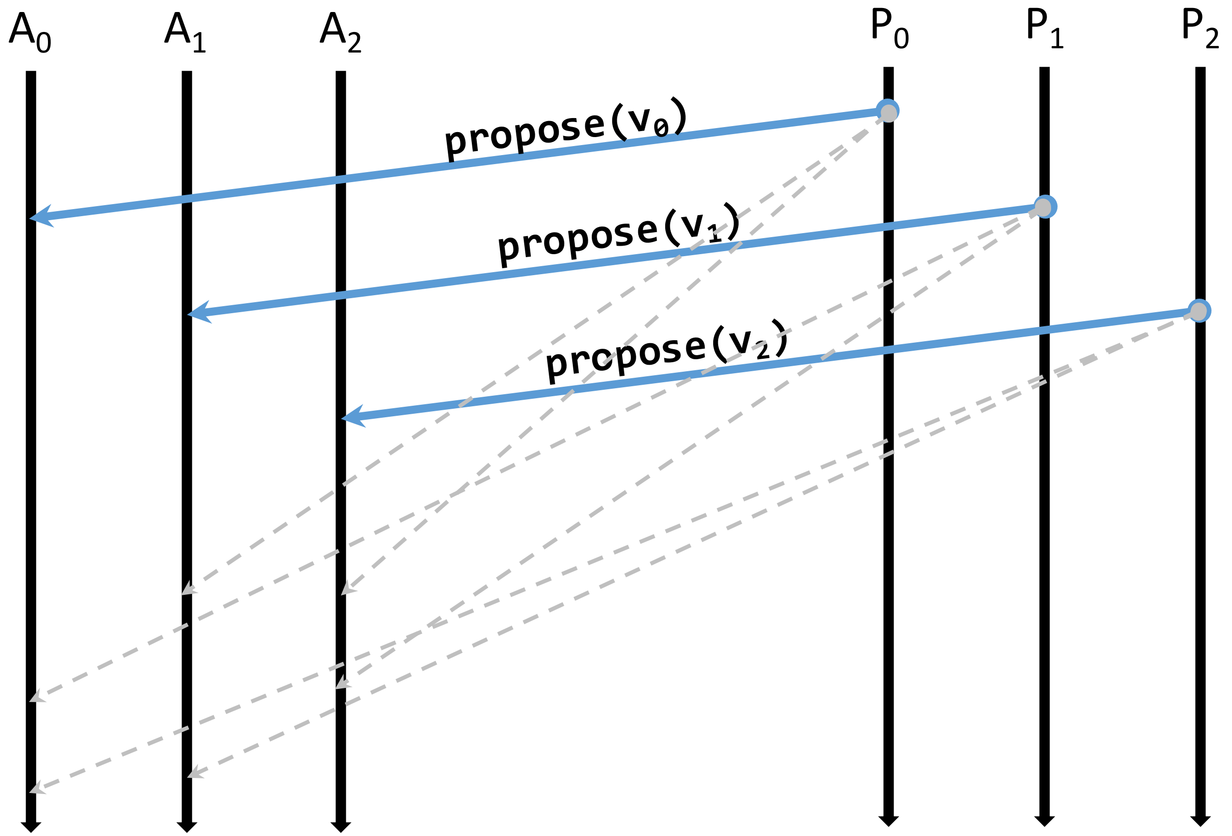
In this scenario all acceptor have accepted different values. There can be no majority that will accept the same value.

Motivation for definitions and rules

If only one proposer proposed a value, then this value must be chosen by the system.

In this situation every acceptor receives only one `propose()` request.

Requirement 1: an acceptor must accept the first proposed value.



In this scenario all acceptor have accepted different values. There can be no majority that will accept the same value.

Idea: one must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

Motivation for definitions and rules

One must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

Motivation for definitions and rules

One must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

Requirement 2a: every proposer issues requests `propose(n, v)` with epoch numbers that are unique and ascending.

Motivation for definitions and rules

One must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

Requirement 2a: every proposer issues requests `propose(n, v)` with epoch numbers that are unique and ascending.
Requirement 2b: different proposers choose their epoch numbers from disjoint sets to ensure that epoch numbers are globally unique.

** In a system with N proposers their epochs from sets* $\{0, N, 2N, \ldots\}$, $\{1, N+1, 2N+1, \ldots\}$, …

Motivation for definitions and rules

One must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

Reminder (requirement 0): an acceptor accepts at most one value.

Motivation for definitions and rules

One must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

Reminder (requirement 0): an acceptor accepts at most one value.

Requirement 0': an acceptor accepts pairs that have the same value and ascending epoch numbers:
$(n_0, v), (n_1, v), \ldots\ (n_k, v)$ with $n_0 < n_1 < \ldots < n_k$.

Motivation for definitions and rules

One must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

Reminder: We will consider a value chosen if it was *accepted* by a majority of acceptors.

Motivation for definitions and rules

One must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

Reminder: We will consider a value chosen if it was *accepted* by a majority of acceptors.

Now that we allow acceptors to accept multiple proposals (n, v) and (n', v), we must allow situations where a majority of acceptors have accepted pairs $\{(n_i, v)\}$ with different epochs. We only need to require that all accepted pairs have the same value v.

Motivation for definitions and rules

One must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

The algorithm of a proposer:

1. Choose an epoch number n that was not used earlier.
2. Send $\mathrm{prepare}(n)$ to all acceptors. It is a request to ignore requests with the epoch number $< n$.

The algorithm of an acceptor:

Motivation for definitions and rules

One must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

The algorithm of a proposer:

1. Choose an epoch number n that was not used earlier.
2. Send $\mathrm{prepare}(n)$ to all acceptors. It is a request to ignore requests with the epoch number $< n$.

The algorithm of an acceptor:

1. Upon receiving $\mathrm{prepare}(n)$, remember the epoch n and ignore all requests with lower epoch numbers. Reply with $\mathrm{promise}(n, m, v)$ where (m, v) is the last accepted value or (nil, nil) if no value has been accepted yet.

Motivation for definitions and rules

One must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

The algorithm of a proposer:

1. Choose an epoch number n that was not used earlier.
2. Send $\mathrm{prepare}(n)$ to all acceptors. It is a request to ignore requests with the epoch number $< n$.

The algorithm of an acceptor:

1. Upon receiving $\mathrm{prepare}(n)$, remember the epoch n and ignore all requests with lower epoch numbers. Reply with $\mathrm{promise}(n, m, v)$ where (m, v) is the last accepted value or (nil, nil) if no value has been accepted yet.
   Note: $\mathrm{promise}(n, m, v)$ has $n > m$.

Motivation for definitions and rules

One must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

The algorithm of a proposer:

1. Choose an epoch number n that was not used earlier.
2. Send $\text{prepare}(n)$ to all acceptors. It is a request to ignore requests with the epoch number $< n$.
3. Wait for $\text{promise}(n, m_i, v_i)$ from a majority of acceptors and **remember a value v with the highest epoch number**. If all replies have $v_i = nil$, the proposer may propose its value.
4. Send $\text{propose}(n, v)$ to all acceptors **that have replied with a promise**.

The algorithm of an acceptor:

1. Upon receiving $\text{prepare}(n)$, remember the epoch n and ignore all requests with lower epoch numbers. Reply with $\text{promise}(n, m, v)$ where (m, v) is the last accepted value or (nil, nil) if no value has been accepted yet.
   Note: $\text{promise}(n, m, v)$ has $n > m$.

Motivation for definitions and rules

One must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

The algorithm of a proposer:

1. Choose an epoch number n that was not used earlier.
2. Send $\mathrm{prepare}(n)$ to all acceptors. It is a request to ignore requests with the epoch number $< n$.
3. Wait for $\mathrm{promise}(n, m_i, v_i)$ from a majority of acceptors and **remember a value v with the highest epoch number**. If all replies have $v_i = nil$, the proposer may propose its value.
4. Send $\mathrm{propose}(n, v)$ to all acceptors **that have replied with a promise**.

The algorithm of an acceptor:

1. Upon receiving $\mathrm{prepare}(n)$, remember the epoch n and ignore all requests with lower epoch numbers. Reply with $\mathrm{promise}(n, m, v)$ where (m, v) is the last accepted value or (nil, nil) if no value has been accepted yet.
   Note: $\mathrm{promise}(n, m, v)$ has $n > m$.
2. Upon receiving $\mathrm{propose}(k, w)$ with k that is greater than the last remembered epoch, accept the value (k, w) and reply $\mathrm{accept}(k, w)$.

Motivation for definitions and rules

One must not propose plain values v, but must propose pairs (n, v) where n is a natural number, the epoch number of a proposer. Acceptors must accept pairs (n, v) and (n', v) with the same v.

The epoch number enables us to distinguish newer requests from stale ones. Proposers can use this numbering to coordinate among themselves and avoid proposing **known-stale values**.

The algorithm of a proposer:

1. Choose an epoch number n that was not used earlier.
2. Send $\text{prepare}(n)$ to all acceptors. It is a request to ignore requests with the epoch number $< n$.
3. Wait for $\text{promise}(n, m_i, v_i)$ from a majority of acceptors and **remember a value v with the highest epoch number**. If all replies have $v_i = nil$, the proposer may propose its value.
4. Send $\text{propose}(n, v)$ to all acceptors **that have replied with a promise**.
5. Wait for $\text{accept}()$ from a majority of acceptors.
6. If timed out, go to #1.

The algorithm of an acceptor:

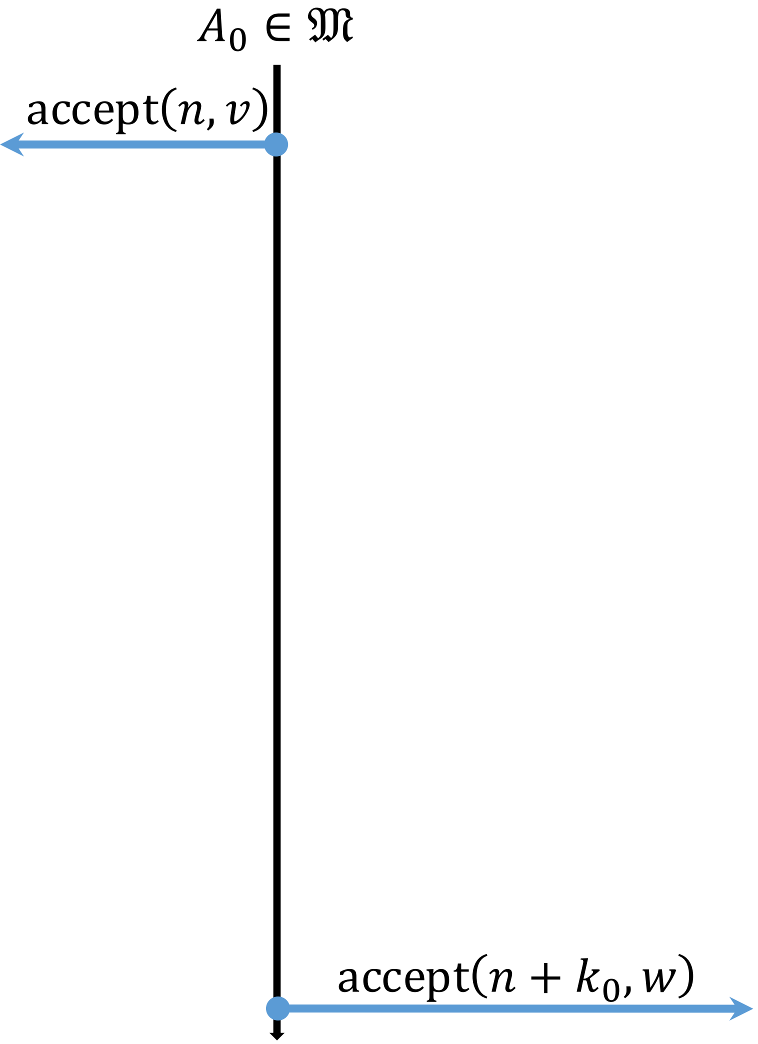
1. Upon receiving $\text{prepare}(n)$, remember the epoch n and ignore all requests with lower epoch numbers. Reply with $\text{promise}(n, m, v)$ where (m, v) is the last accepted value or (nil, nil) if no value has been accepted yet.
   Note: $\text{promise}(n, m, v)$ has $n > m$.
2. Upon receiving $\text{propose}(k, w)$ with k that is greater than the last remembered epoch, accept the value (k, w) and reply $\text{accept}(k, w)$.

Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.

Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n+k, w)$ with $k > 0$ and $w \neq v$.

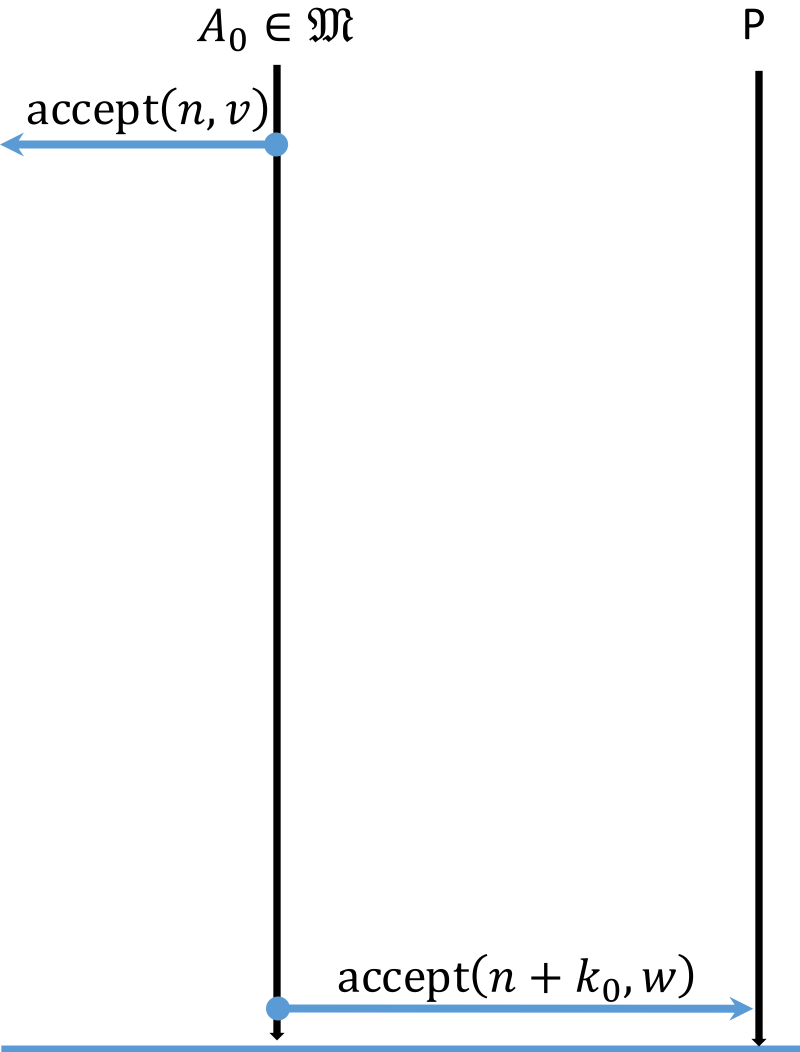


Remark: let us consider **the minimal epoch number** $n + k_0$ when some acceptor from $\mathfrak{M}$ sent $\mathrm{accept}(n+k_0, w)$ after having accepted (n, v).

In this epoch the other acceptors from $\mathfrak{M}$ have not yet sent any $\mathrm{accept}(\cdot, w)$ after $\mathrm{accept}(n, v)$.

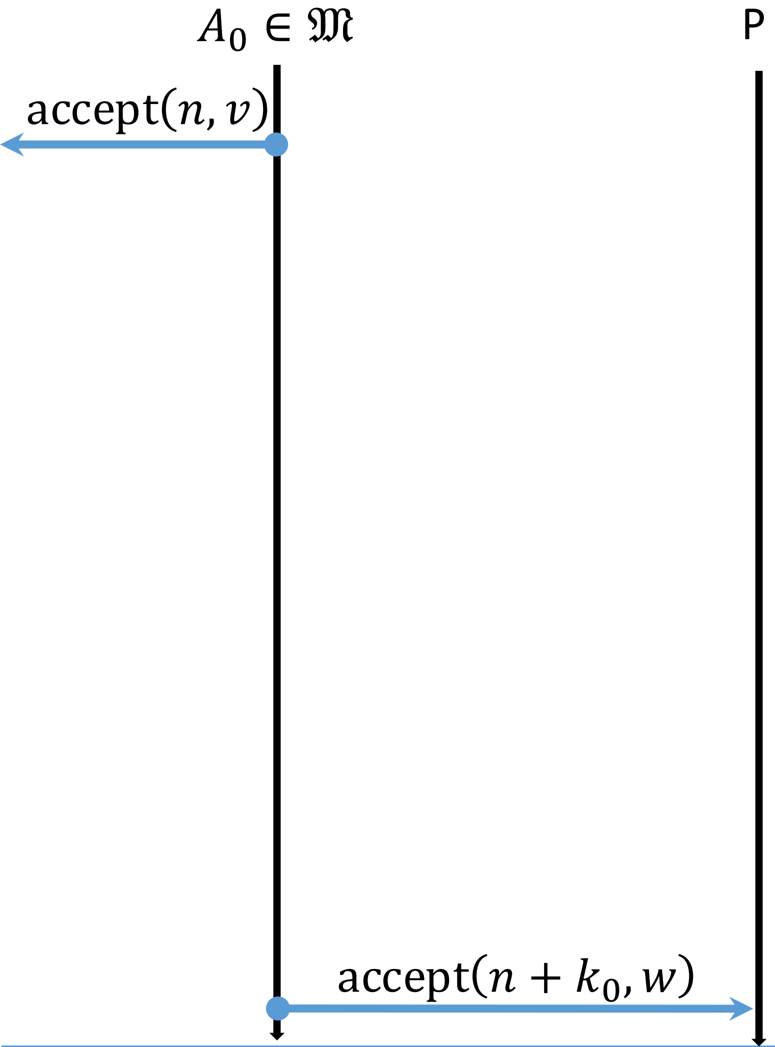
Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.



Safety (only one value can be chosen)

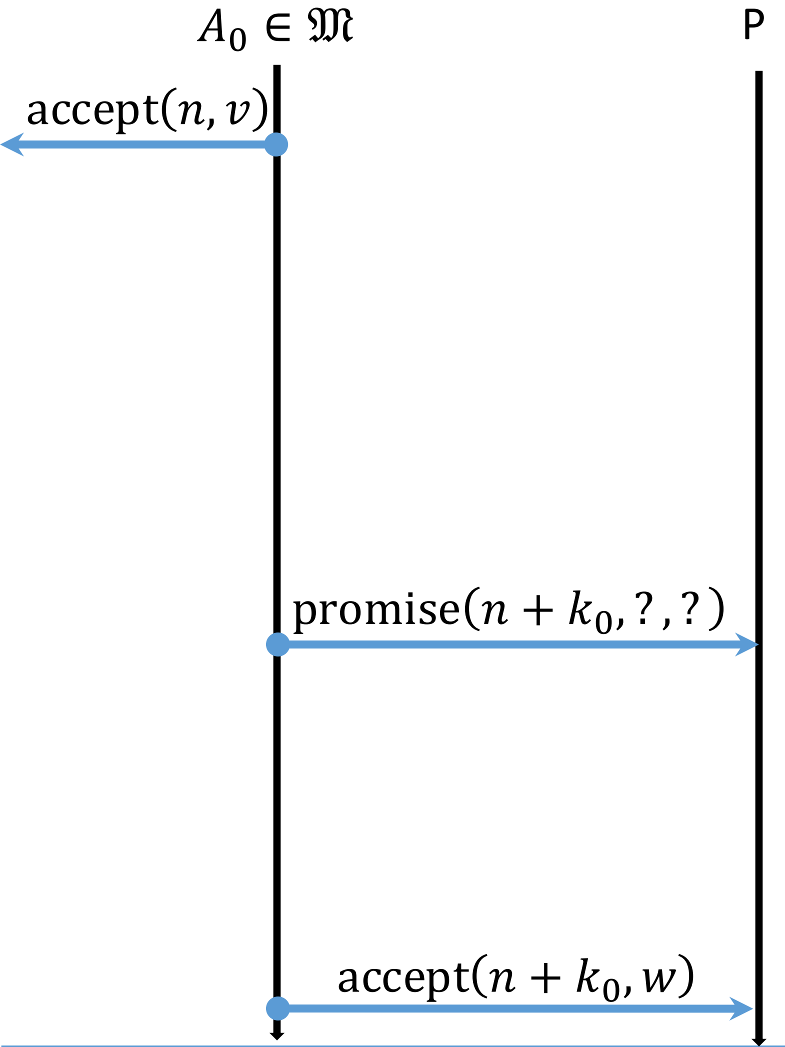
It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n+k, w)$ with $k > 0$ and $w \neq v$.



1. A message $\text{accept}(n + k_0, w)$ is a reply to $\text{propose}(n + k_0, w)$ from a proposer P.
2. P sends $\text{propose}()$ only to acceptors that have earlier replied with a promise.

Safety (only one value can be chosen)

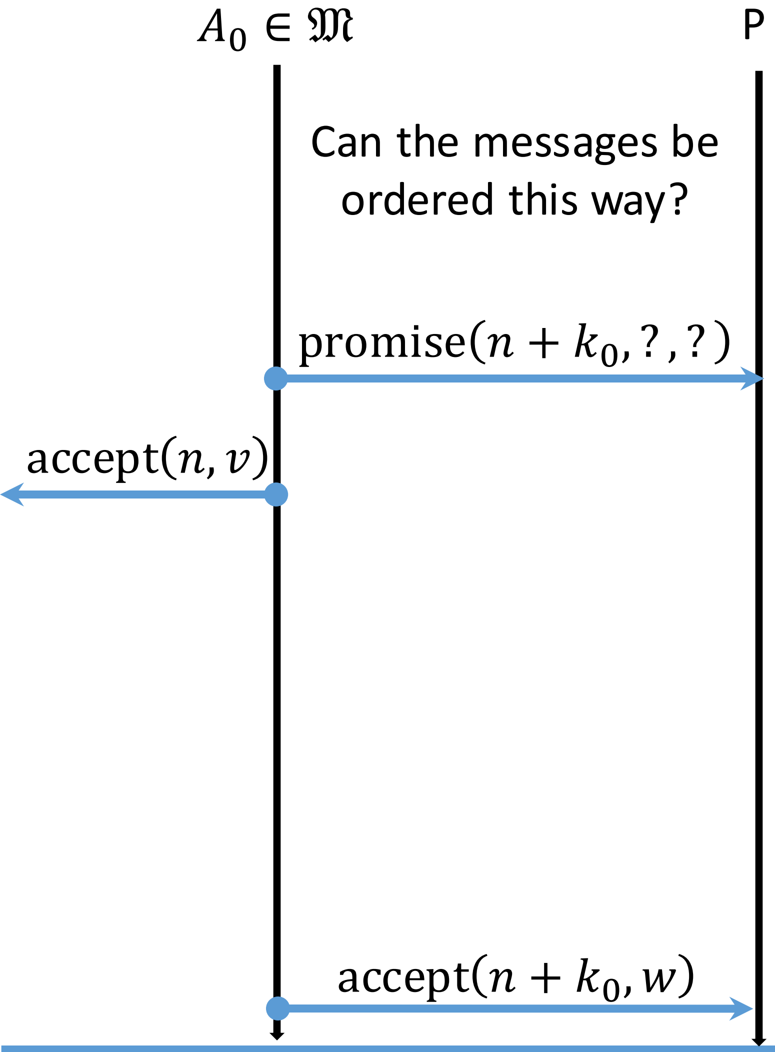
It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.



1. A message $\mathrm{accept}(n + k_0, w)$ is a reply to $\mathrm{propose}(n + k_0, w)$ from a proposer P.
2. P sends $\mathrm{propose}()$ only to acceptors that have earlier replied with a promise.

Safety (only one value can be chosen)

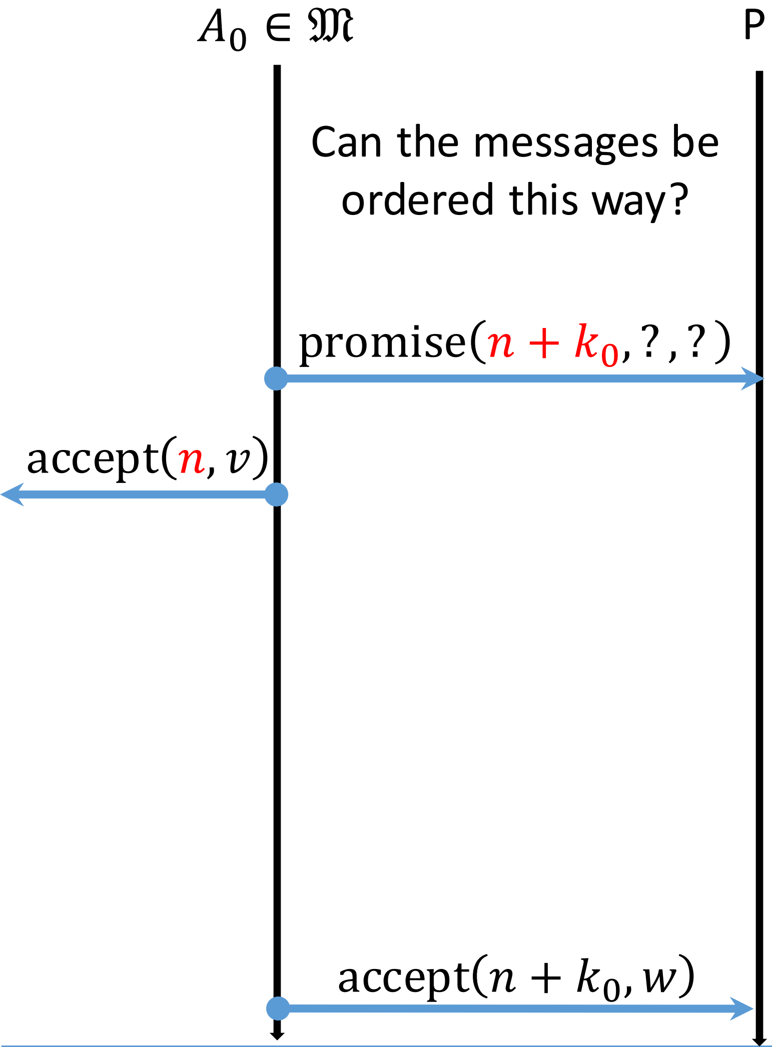
It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.



1. A message $\mathrm{accept}(n + k_0, w)$ is a reply to $\mathrm{propose}(n + k_0, w)$ from a proposer P.
2. P sends $\mathrm{propose}()$ only to acceptors that have earlier replied with a promise.

Safety (only one value can be chosen)

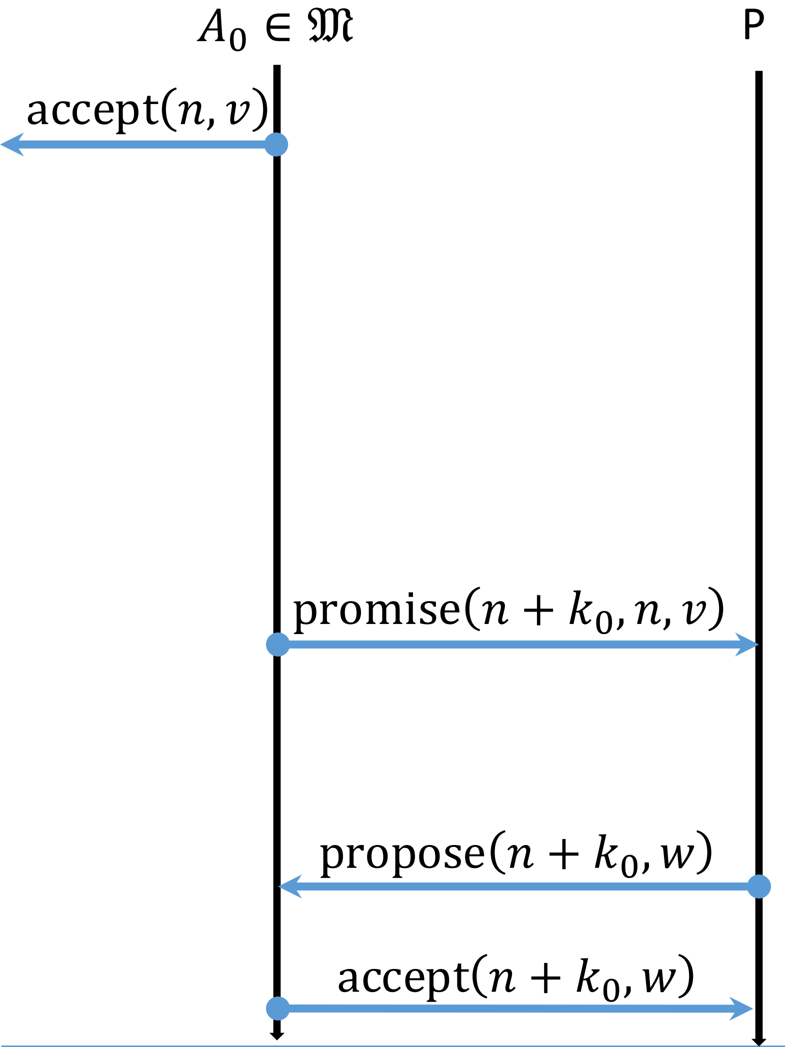
It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n+k, w)$ with $k > 0$ and $w \neq v$.



1. A message $\mathrm{accept}(n+k_0, w)$ is a reply to $\mathrm{propose}(n+k_0, w)$ from a proposer P.
2. P sends $\mathrm{propose}()$ only to acceptors that have earlier replied with a promise.

Safety (only one value can be chosen)

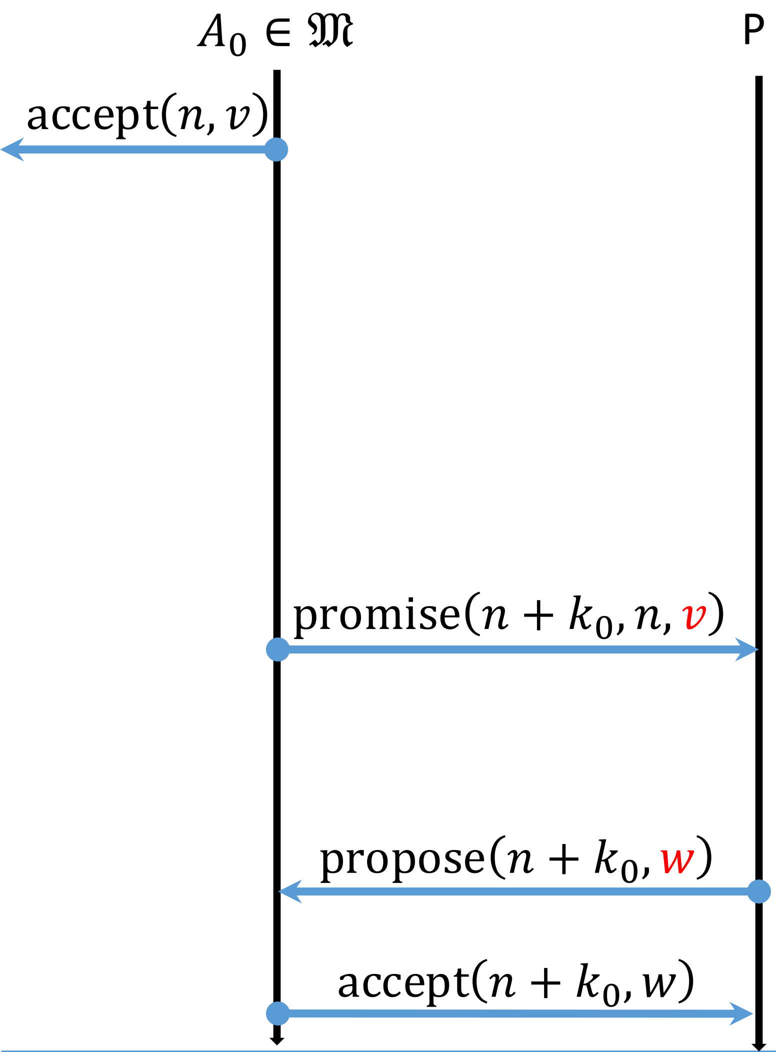
It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.



1. A message $\mathrm{accept}(n + k_0, w)$ is a reply to $\mathrm{propose}(n + k_0, w)$ from a proposer P.
2. P sends $\mathrm{propose}()$ only to acceptors that have earlier replied with a promise.

Safety (only one value can be chosen)

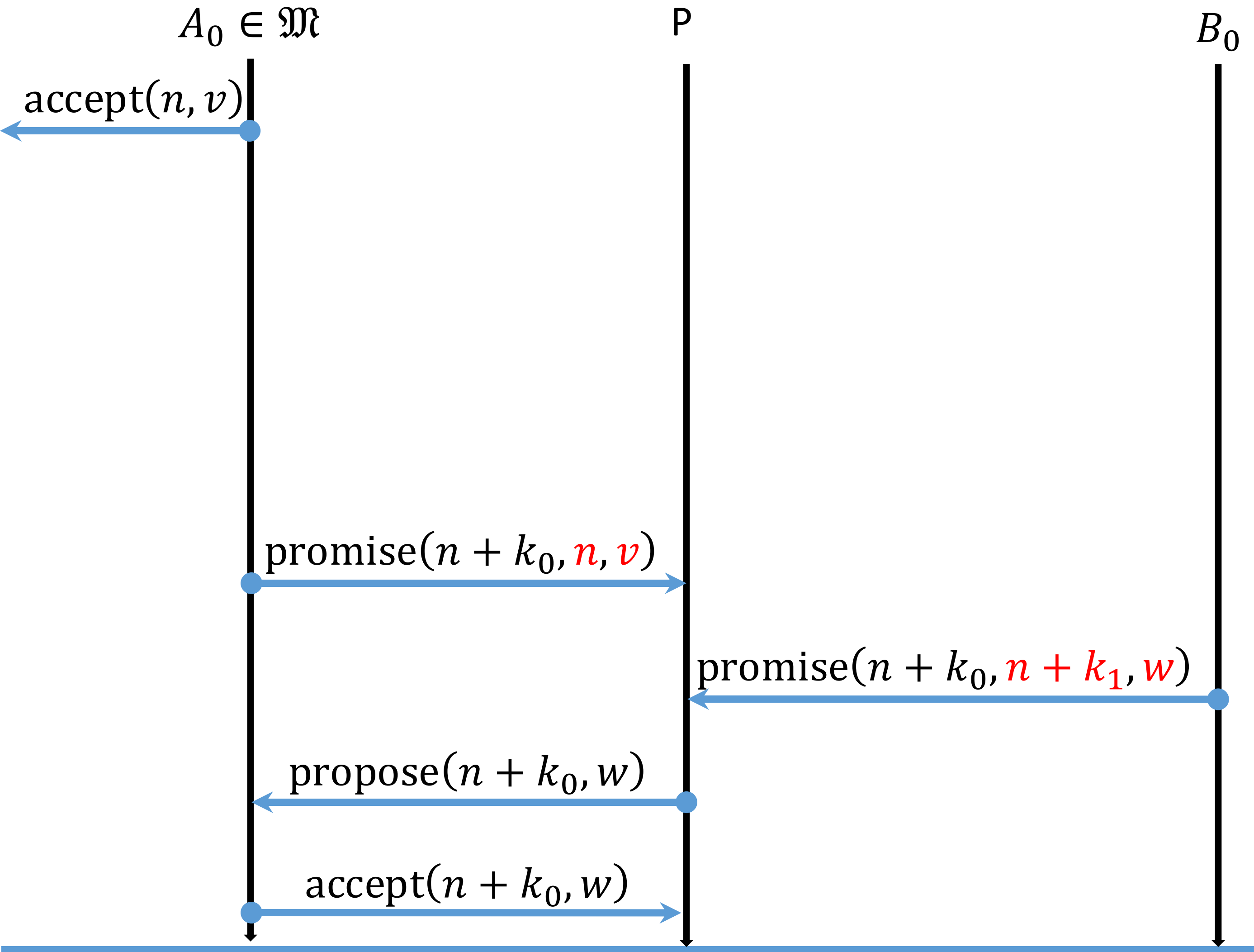
It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.



1. A message accept$(n + k_0, w)$ is a reply to propose$(n + k_0, w)$ from a proposer P.
2. P sends propose() only to acceptors that have earlier replied with a promise.
3. P has received promise$(n + k_0, n, v)$. How can it be that P sent propose$(n + k_0, w)$?

Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.

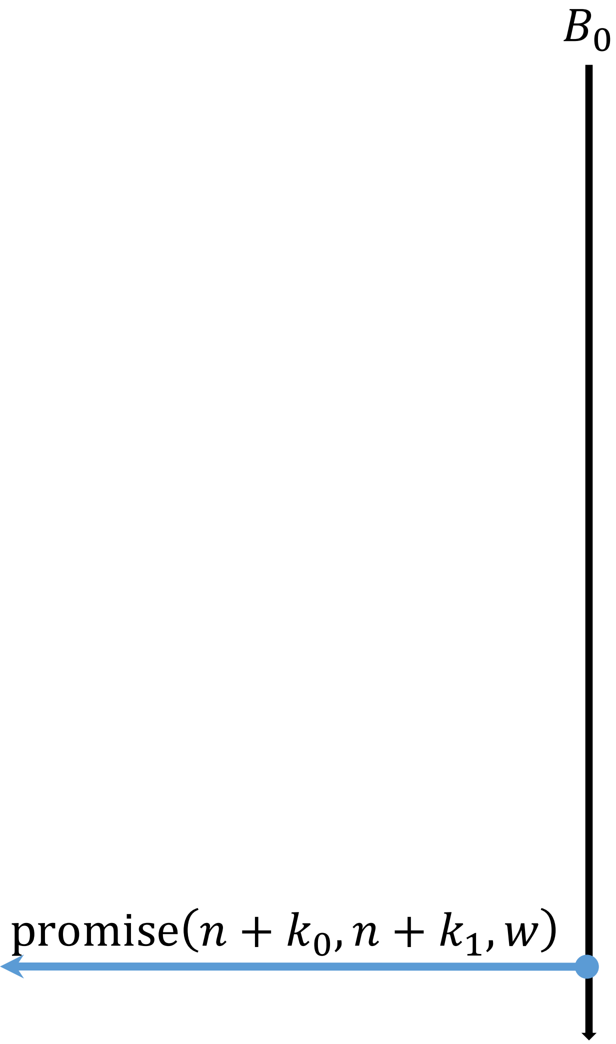


1. A message $\mathrm{accept}(n + k_0, w)$ is a reply to $\mathrm{propose}(n + k_0, w)$ from a proposer P.
2. P sends $\mathrm{propose}()$ only to acceptors that have earlier replied with a promise.
3. P has received $\mathrm{promise}(n + k_0, n, v)$. How can it be that P sent $\mathrm{propose}(n + k_0, w)$?

   This is only possible if P received $\mathrm{promise}(n + k_0, n + k_1, w)$ with $n < n + k_1 < n + k_0$.

Safety (only one value can be chosen)

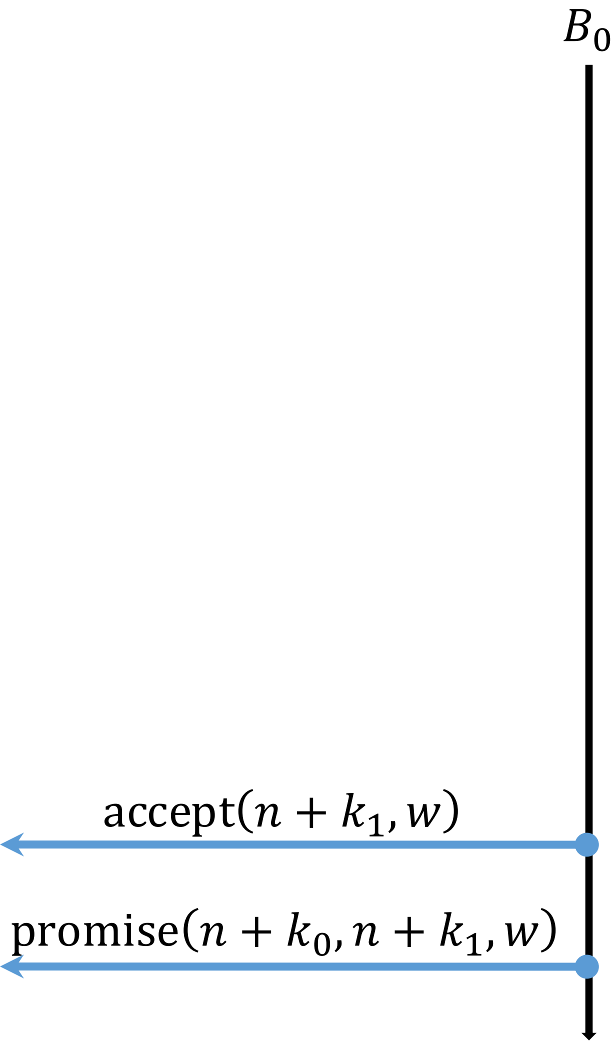
It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.



Let us examine the acceptor B_0 more closely.

Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n+k, w)$ with $k > 0$ and $w \neq v$.

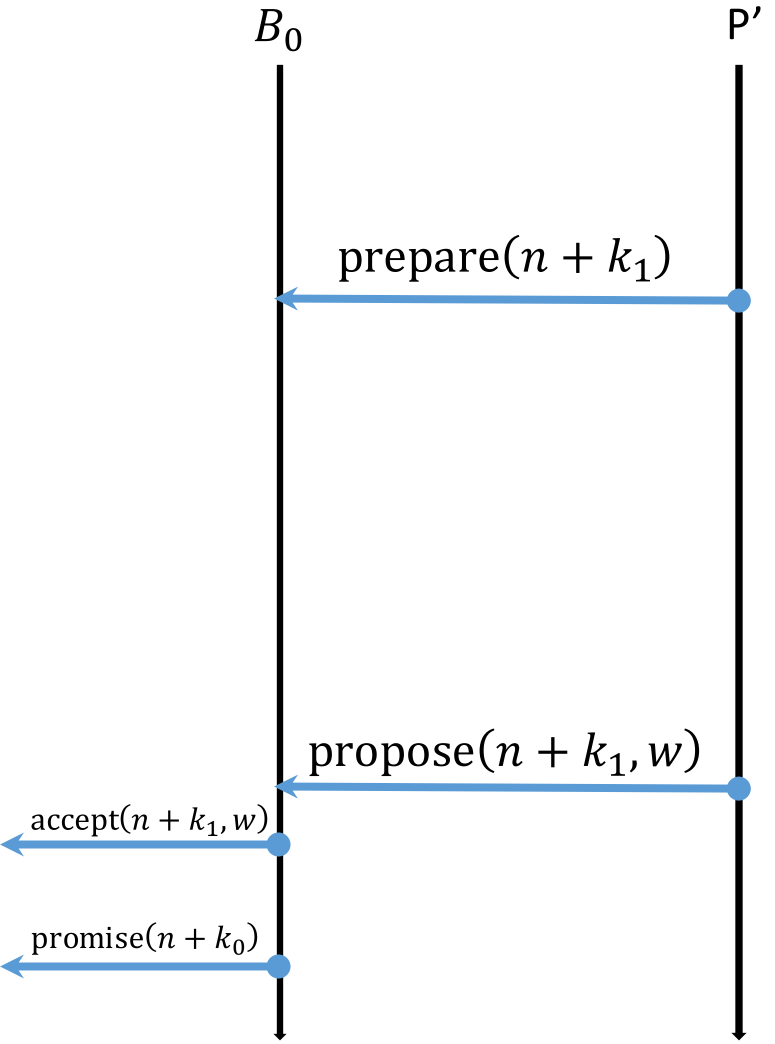


Let us examine the acceptor B_0 more closely.

1. B_0 sends $\text{promise}(n+k_0, n+k_1, w)$ only after $\text{accept}(n+k_1, w)$.

Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.

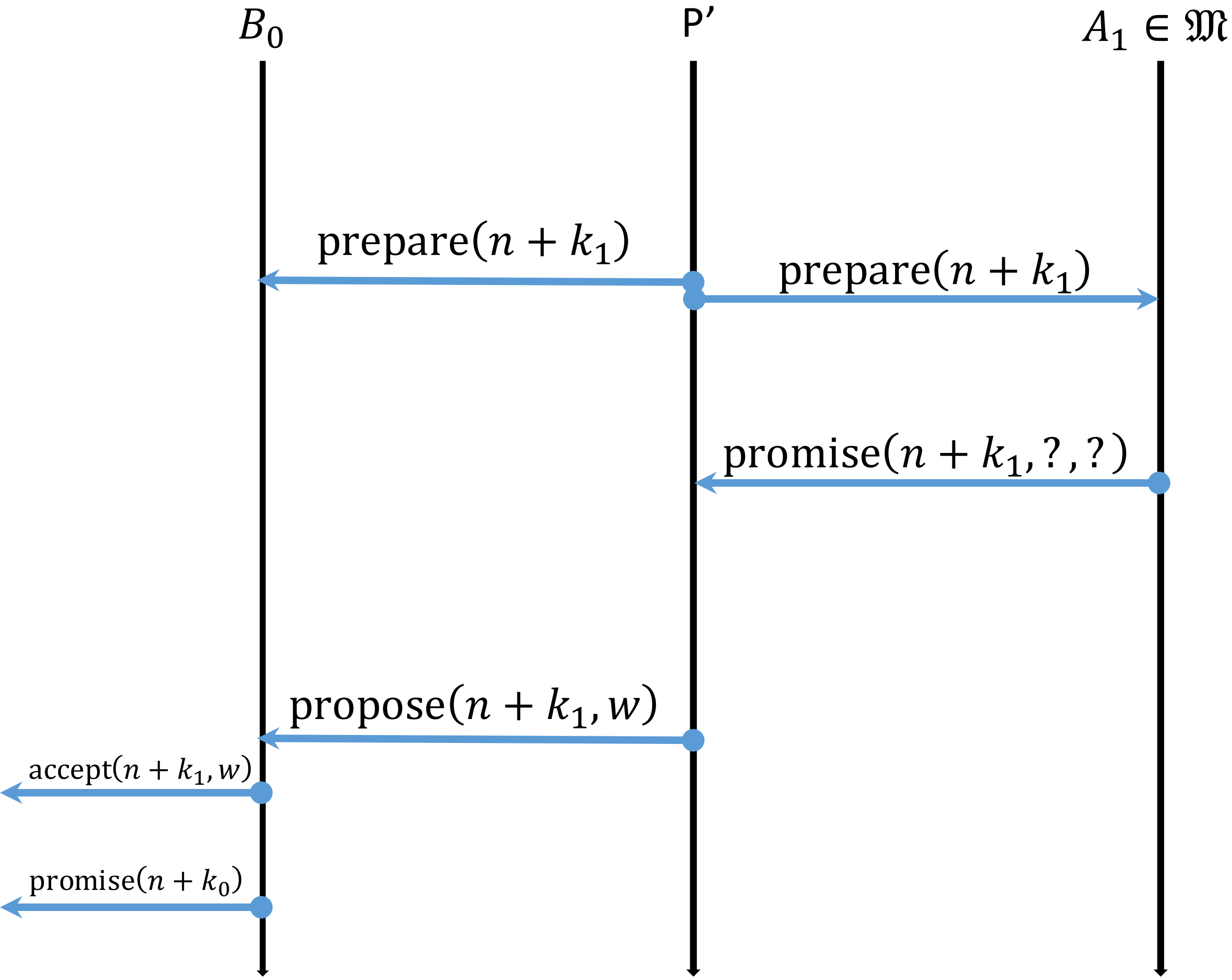


Let us examine the acceptor B_0 more closely.

1. B_0 sends promise$(n + k_0, n + k_1, w)$ only after accept$(n + k_1, w)$.
2. B_0 has accepted $(n + k_1, w)$. Hence, there is a proposer P' that has proposed it.

Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.

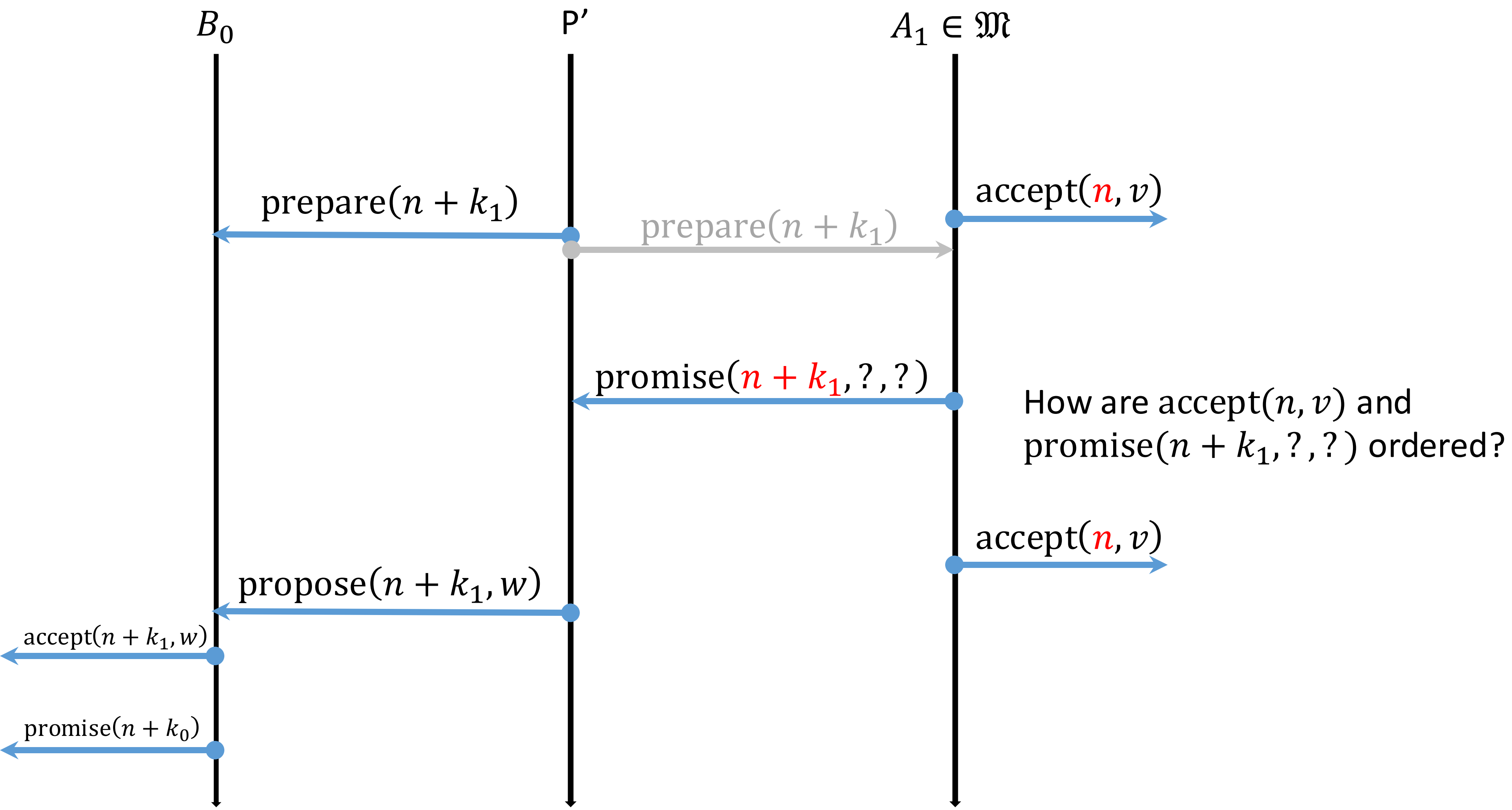


Let us examine the acceptor B_0 more closely.

1. B_0 sends $\text{promise}(n + k_0, n + k_1, w)$ only after $\text{accept}(n + k_1, w)$.
2. B_0 has accepted $(n + k_1, w)$. Hence, there is a proposer P' that has proposed it.
3. P' proposes $(n + k_1, w)$ only after receiving $\text{promise}(n + k_1)$ from a majority of acceptors. In particular, it must have received $\text{promise}(n + k_1)$ from an acceptor $A_1 \in \mathfrak{M}$.

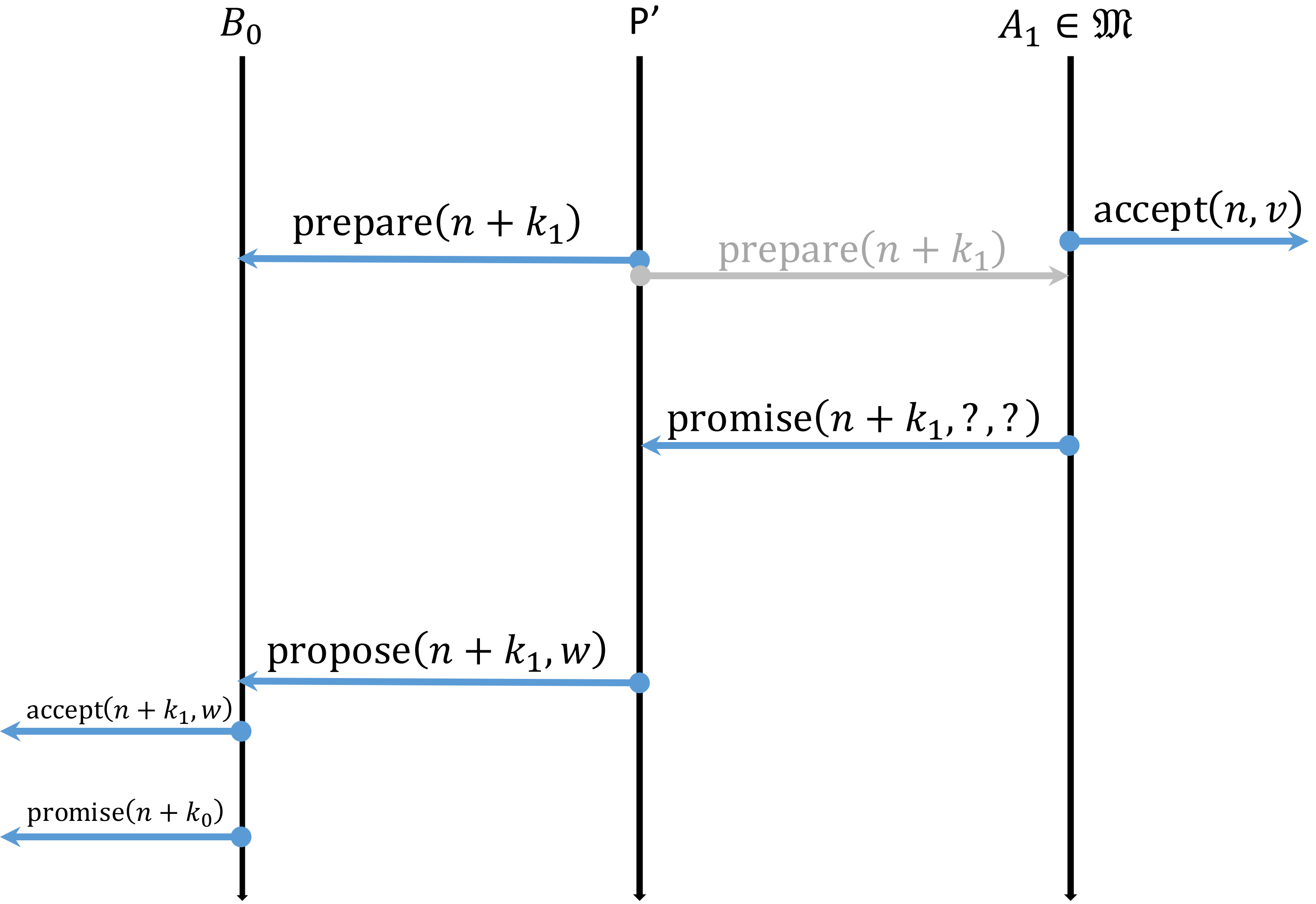
Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.



Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.

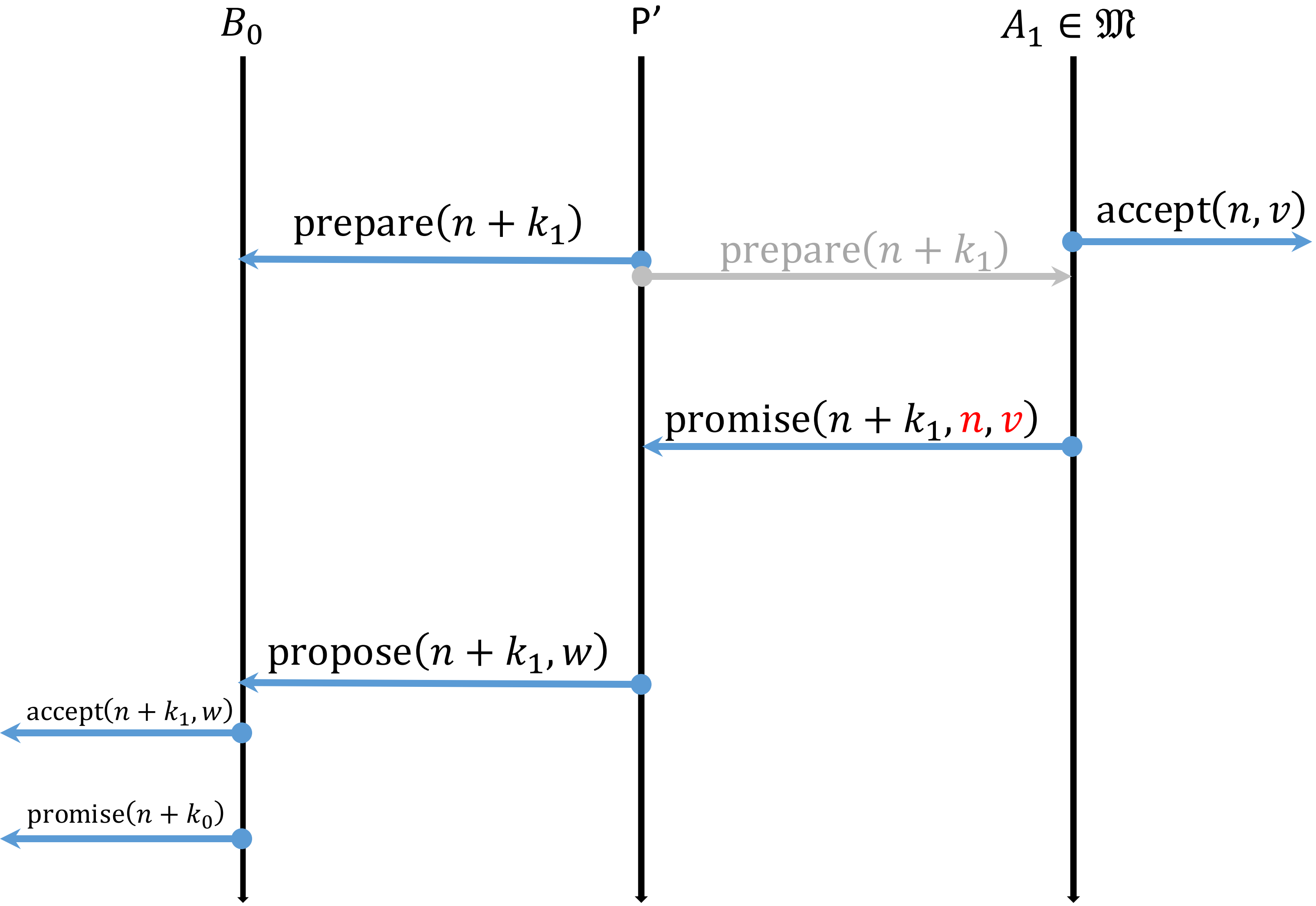


Let us examine the acceptor B_0 more closely.

1. B_0 sends $\mathrm{promise}(n + k_0, n + k_1, w)$ only after $\mathrm{accept}(n + k_1, w)$.
2. B_0 has accepted $(n + k_1, w)$. Hence, there is a proposer P' that has proposed it.
3. P' proposes $(n + k_1, w)$ only after receiving $\mathrm{promise}(n + k_1)$ from a majority of acceptors. In particular, it must have received $\mathrm{promise}(n + k_1)$ from an acceptor $A_1 \in \mathfrak{M}$.

Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.

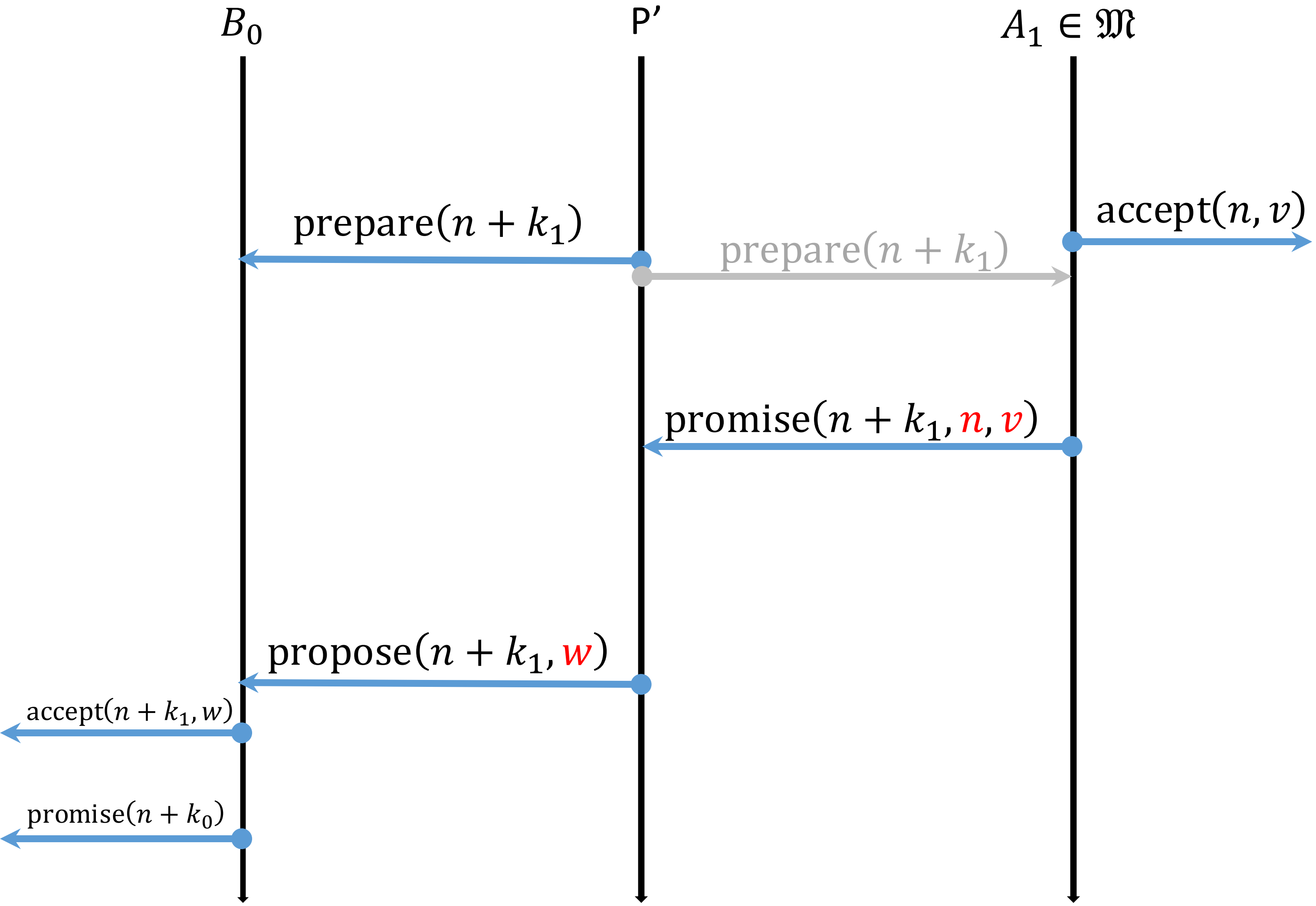


Let us examine the acceptor B_0 more closely.

1. B_0 sends $\text{promise}(n + k_0, n + k_1, w)$ only after $\text{accept}(n + k_1, w)$.
2. B_0 has accepted $(n + k_1, w)$. Hence, there is a proposer P' that has proposed it.
3. P' proposes $(n + k_1, w)$ only after receiving $\text{promise}(n + k_1)$ from a majority of acceptors. In particular, it must have received $\text{promise}(n + k_1)$ from an acceptor $A_1 \in \mathfrak{M}$.
4. Recall that $n + k_0$ is the minimal epoch when an acceptor in $\mathfrak{M}$ replies with a value $w \neq v$. Thus, A_1 has replied $\text{promise}(n + k_1, n, v)$.

Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.

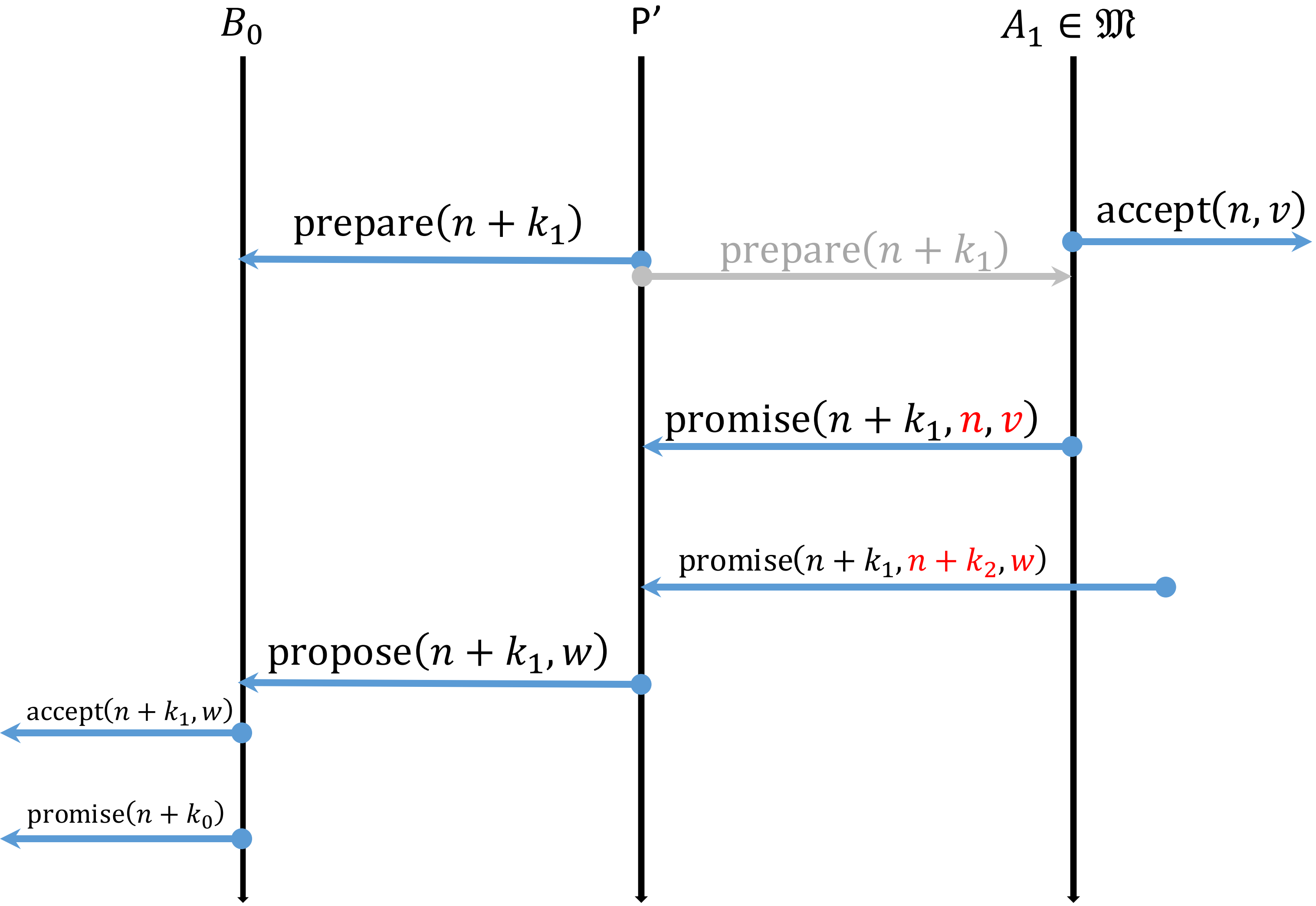


Let us examine the acceptor B_0 more closely.

1. B_0 sends promise$(n + k_0, n + k_1, w)$ only after accept$(n + k_1, w)$.
2. B_0 has accepted $(n + k_1, w)$. Hence, there is a proposer P' that has proposed it.
3. P' proposes $(n + k_1, w)$ only after receiving promise$(n + k_1)$ from a majority of acceptors. In particular, it must have received promise$(n + k_1)$ from an acceptor $A_1 \in \mathfrak{M}$.
4. A_1 has replied promise$(n + k_1, n, v)$.
5. P' has received promise$(n + k_0, n, v)$. What made it send propose$(n + k_1, w)$?

Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n + k, w)$ with $k > 0$ and $w \neq v$.



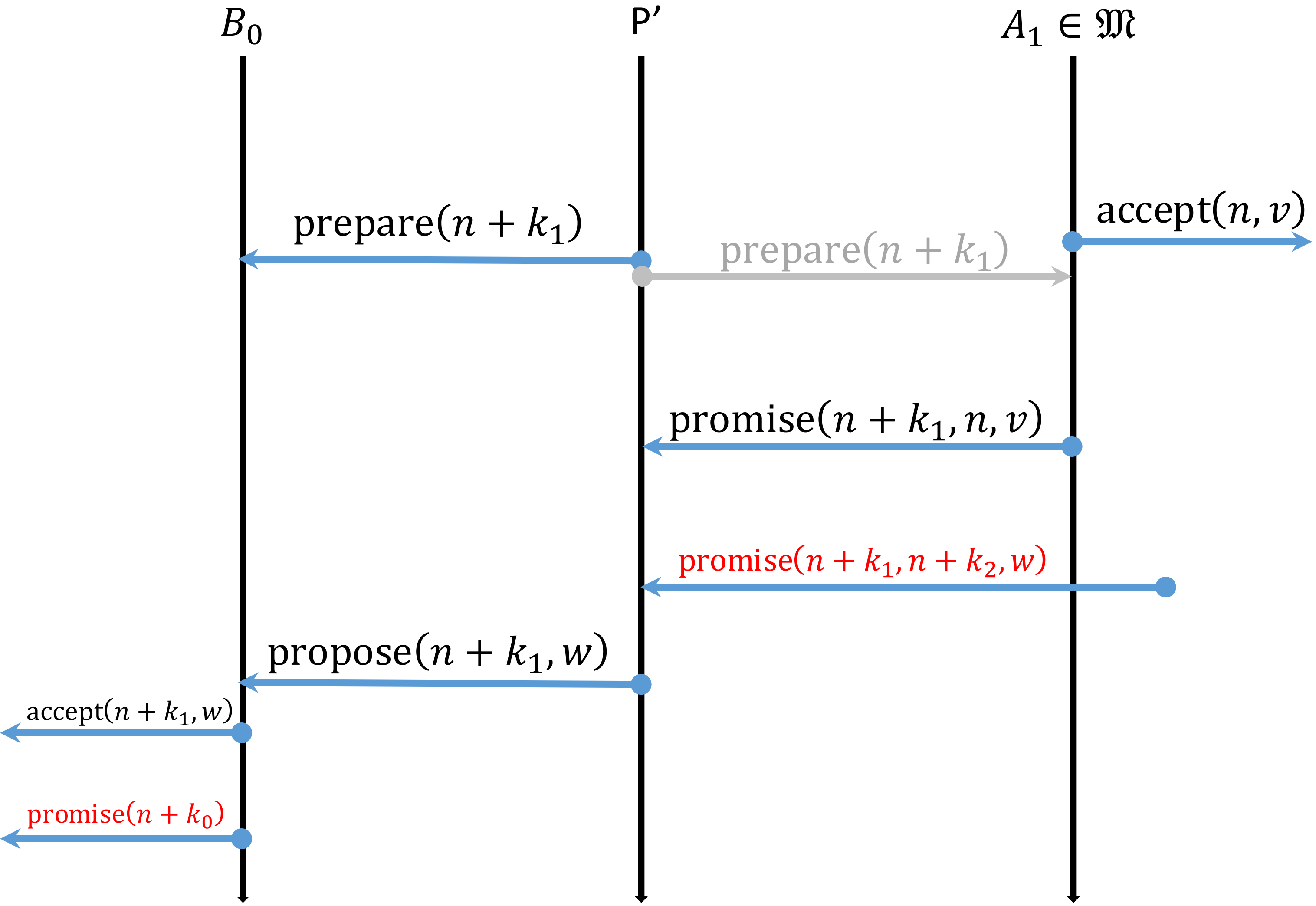
Let us examine the acceptor B_0 more closely.

1. B_0 sends promise$(n + k_0, n + k_1, w)$ only after accept$(n + k_1, w)$.
2. B_0 has accepted $(n + k_1, w)$. Hence, there is a proposer P' that has proposed it.
3. P' proposes $(n + k_1, w)$ only after receiving promise$(n + k_1)$ from a majority of acceptors. In particular, it must have received promise$(n + k_1)$ from an acceptor $A_1 \in \mathfrak{M}$.
4. A_1 has replied promise$(n + k_1, n, v)$.
5. P' has received promise$(n + k_0, n, v)$. What made it send propose$(n + k_1, w)$?

This is only possible if P' received promise$(n + k_1, n + k_2, w)$ with $n < n + k_2 < n + k_1$.

Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n+k, w)$ with $k > 0$ and $w \neq v$.



There exists the acceptor that sent replies and accept(n, v) and accept$(n+k_0, w)$.

$\Downarrow$

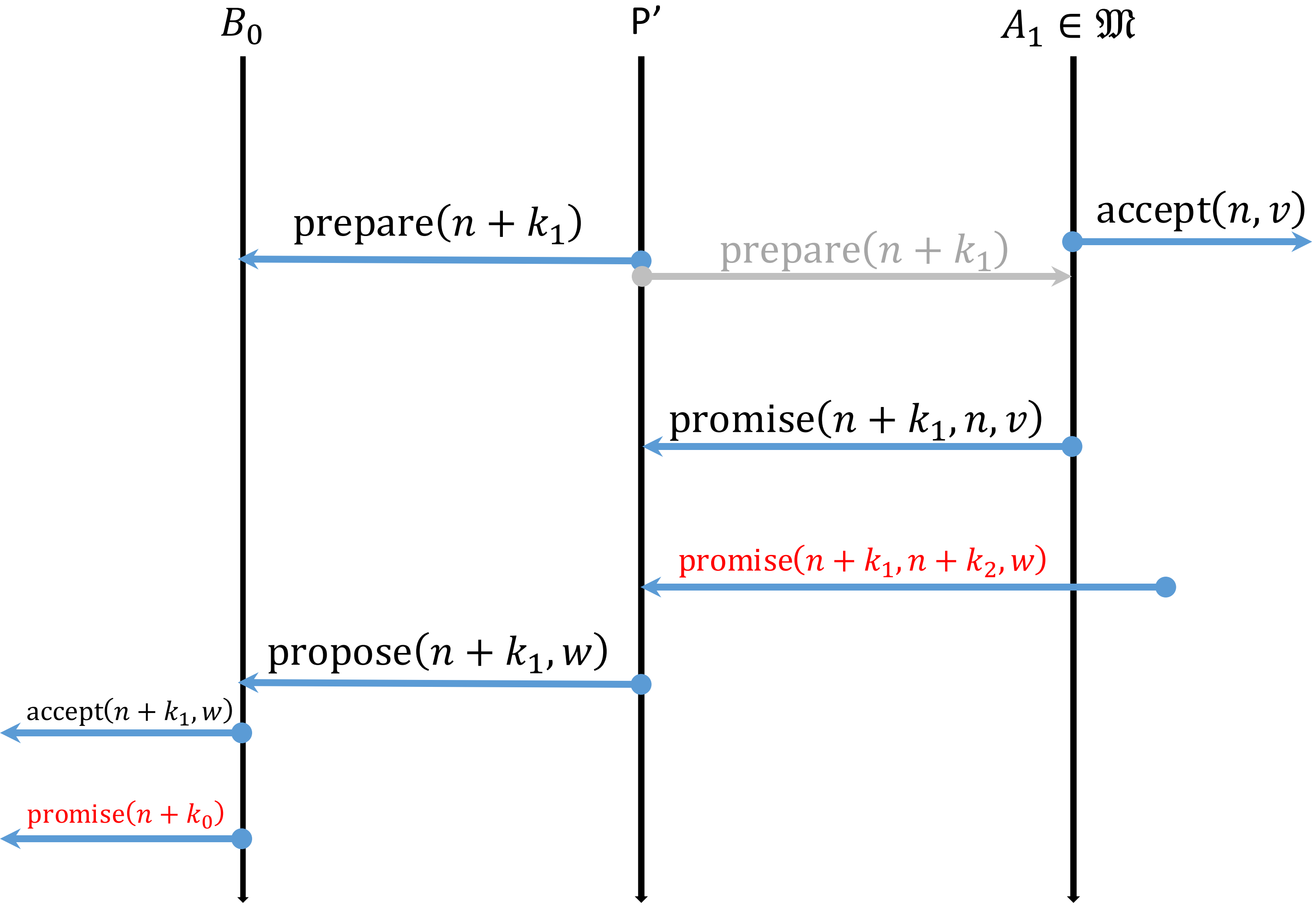
There exists an acceptor that sent promise$(n+k_0, n+k_1, w)$ with $n < n+k_1 < n+k_0$.

$\Downarrow$

There exists an acceptor that sent promise$(n+k_1, n+k_2, w)$ with $n < n+k_2 < n+k_1 < n+k_0$.

Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n+k, w)$ with $k > 0$ and $w \neq v$.



There exists the acceptor that sent replies and accept(n, v) and accept$(n+k_0, w)$.

$\Downarrow$

There exists an acceptor that sent promise$(n+k_0, n+k_1, w)$ with $n < n+k_1 < n+k_0$.

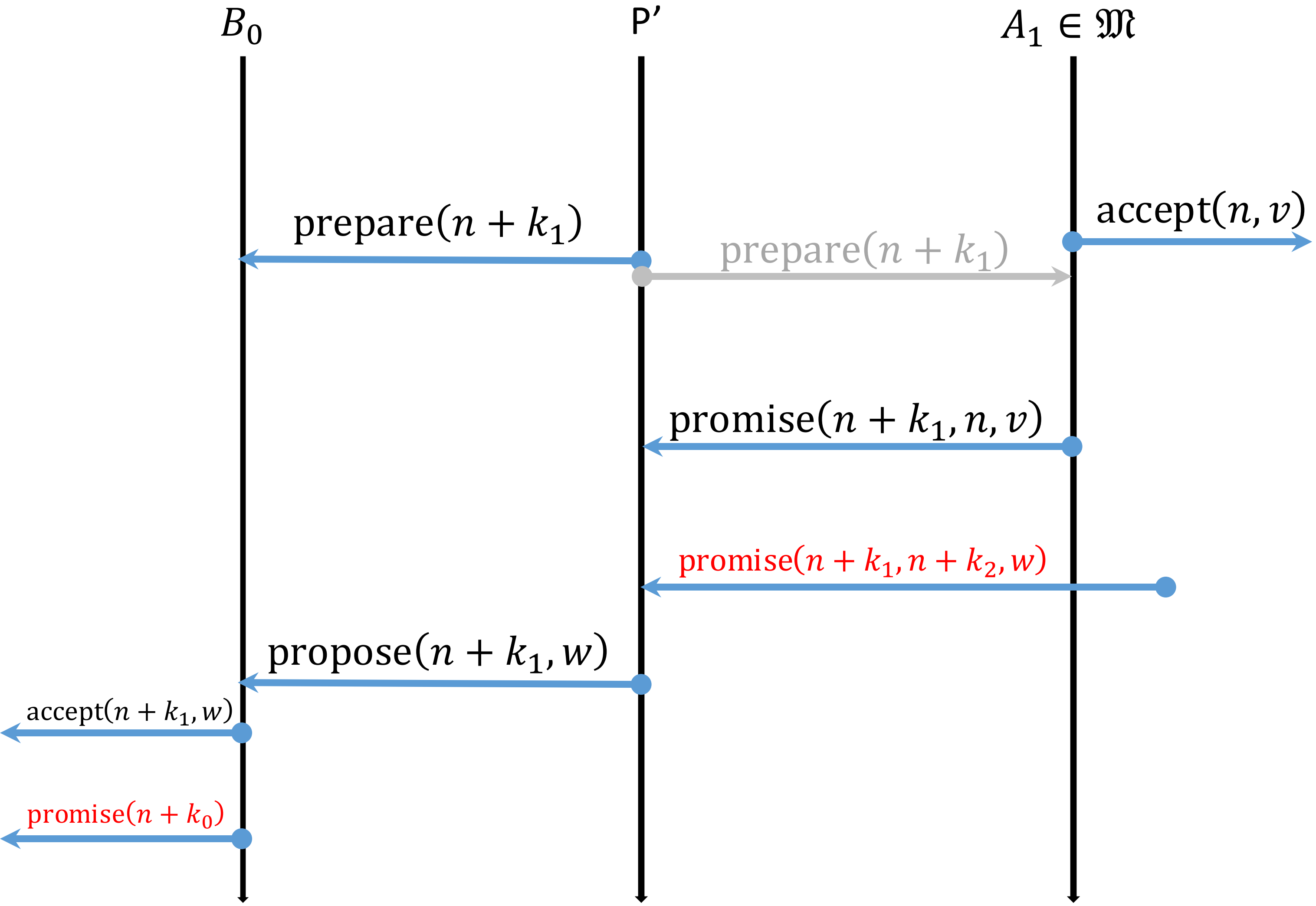
$\Downarrow$

There exists an acceptor that sent promise$(n+k_1, n+k_2, w)$ with $n < n+k_2 < n+k_1 < n+k_0$.

We can keep applying this deduction indefinitely which leads to a contradiction because there only is a finite number of epochs between n and $n+k_0$.

Safety (only one value can be chosen)

It seems that we do not forbid the following scenario: a majority $\mathfrak{M}$ of acceptors accepted a value (n, v) and then some of them accepted $(n+k, w)$ with $k > 0$ and $w \neq v$.



There exists the acceptor that sent replies and $\mathrm{accept}(n, v)$ and $\mathrm{accept}(n+k_0, w)$.

$\Downarrow$

There exists an acceptor that sent $\mathrm{promise}(n+k_0, n+k_1, w)$ with $n < n+k_1 < n+k_0$.

$\Downarrow$

There exists an acceptor that sent $\mathrm{promise}(n+k_1, n+k_2, w)$ with $n < n+k_2 < n+k_1 < n+k_0$.

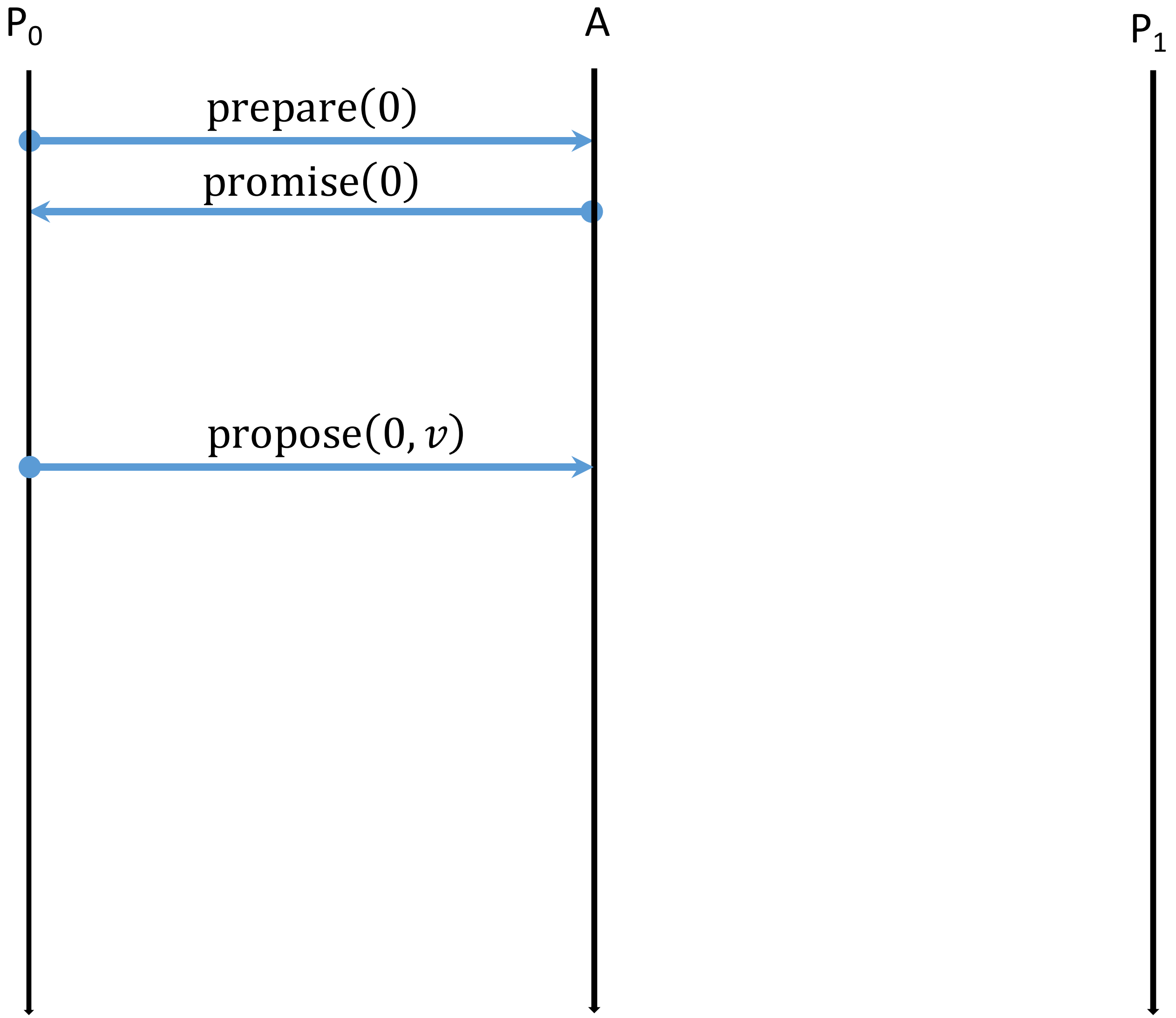
We can keep applying this deduction indefinitely which leads to a contradiction because there only is a finite number of epochs between n and $n+k_0$.

$\Downarrow$

If an acceptor replied $\mathrm{accept}(n, v)$, it can no longer reply $\mathrm{accept}(n+k_0, w)$.

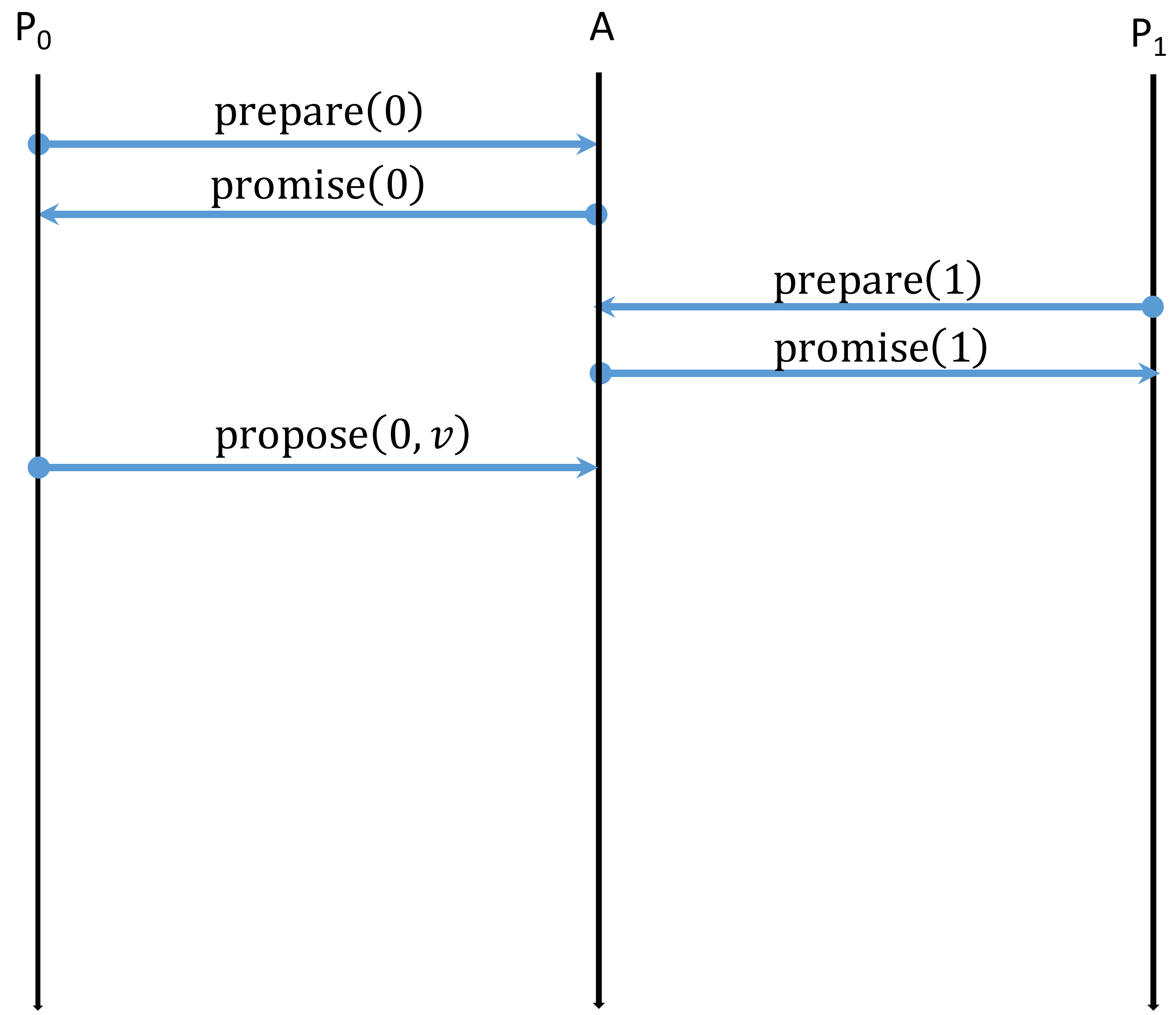
Liveness

PAXOS guarantees safety (at most one value is chosen). Does it guarantee liveness (a value is chosen)?



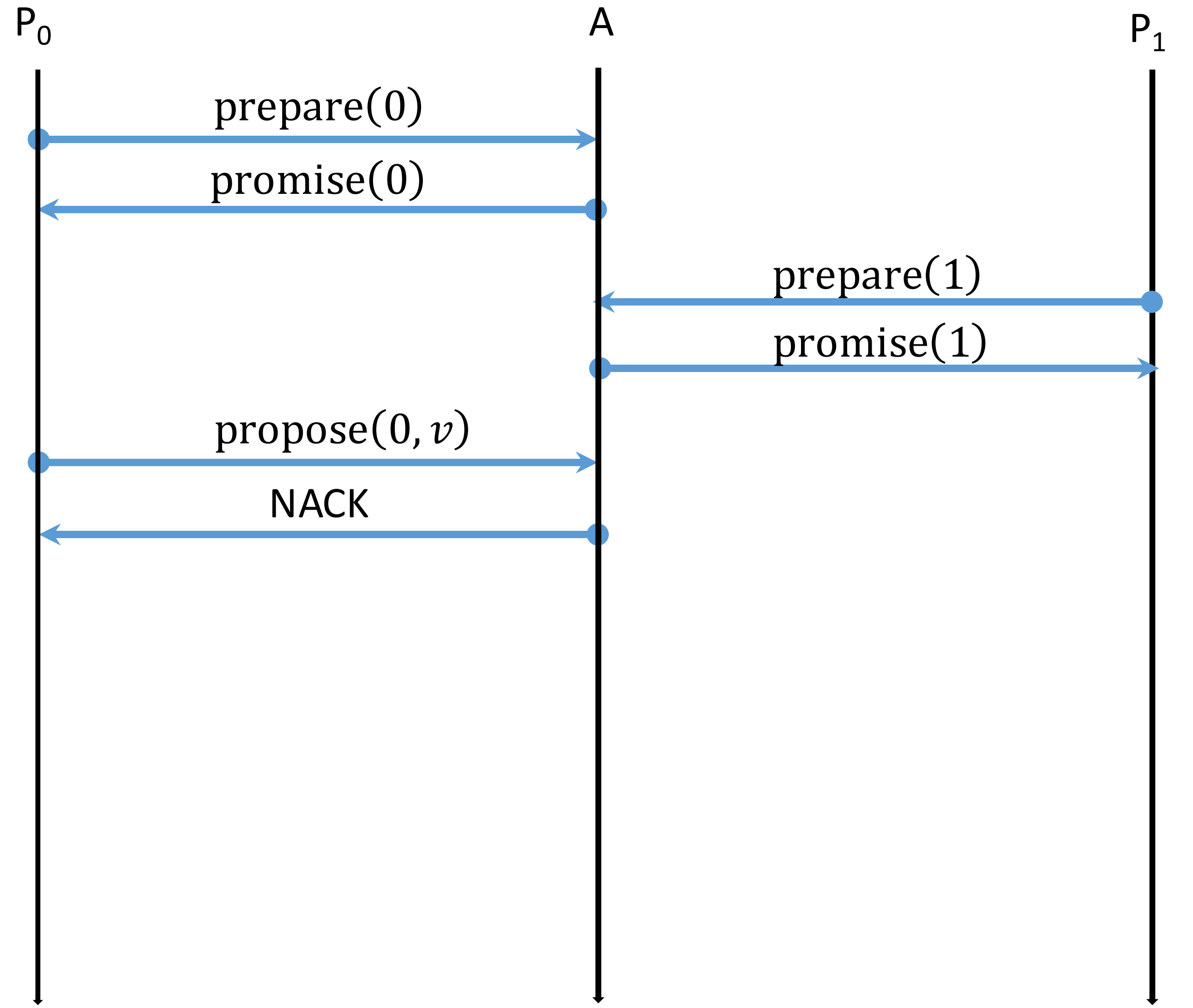
Liveness

PAXOS guarantees safety (at most one value is chosen). Does it guarantee liveness (a value is chosen)?



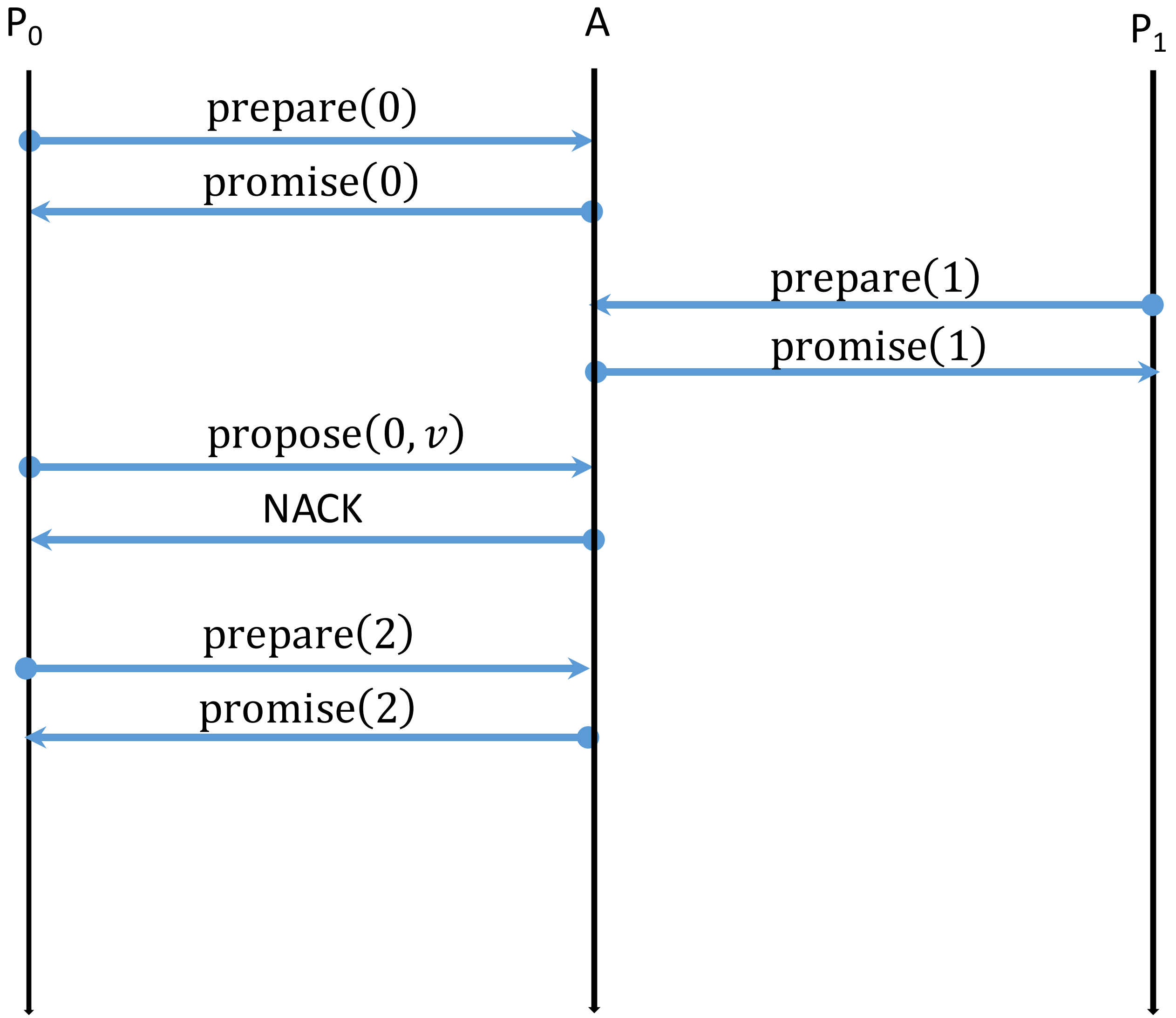
Liveness

PAXOS guarantees safety (at most one value is chosen). Does it guarantee liveness (a value is chosen)?



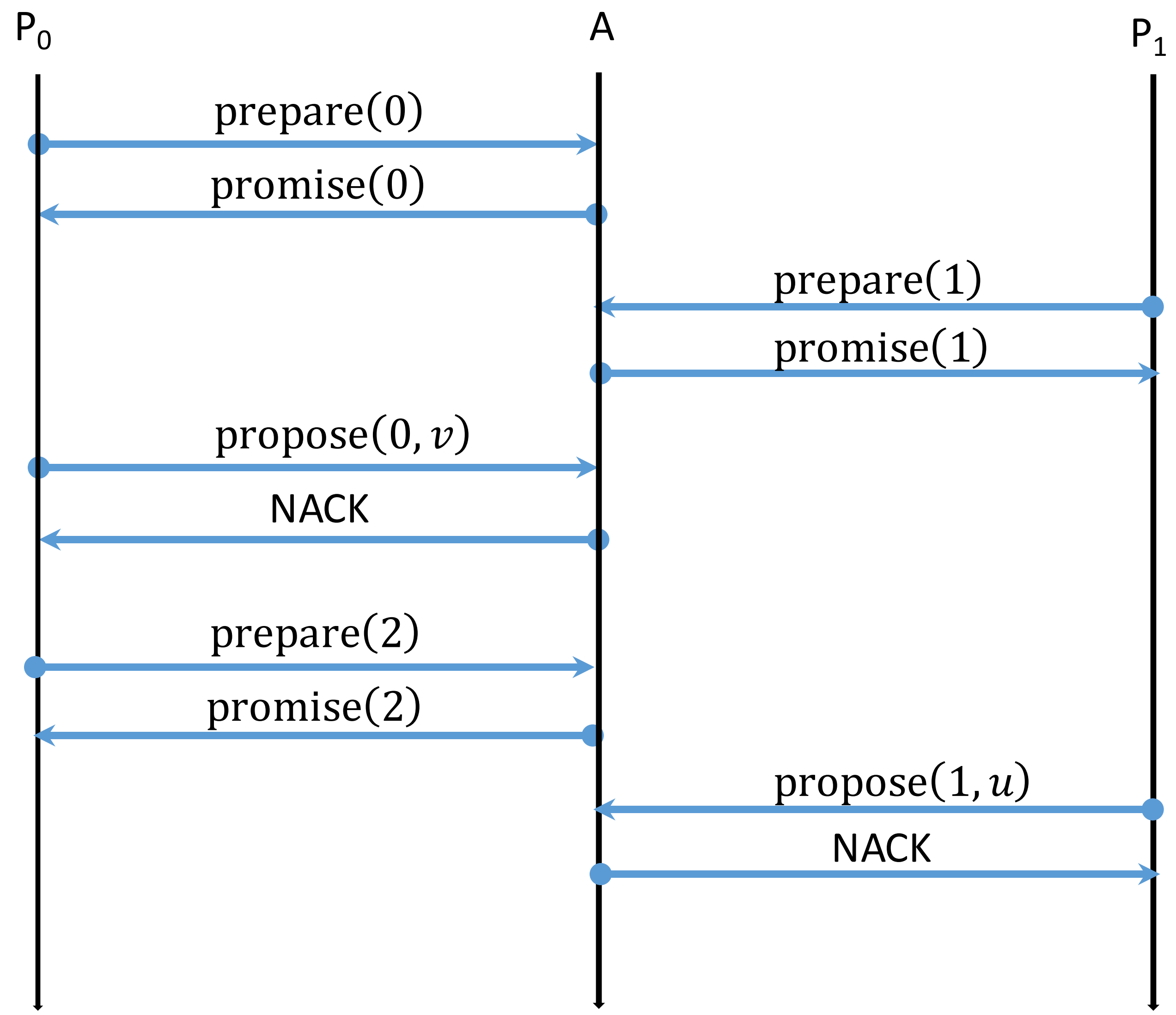
Liveness

PAXOS guarantees safety (at most one value is chosen). Does it guarantee liveness (a value is chosen)?



Liveness

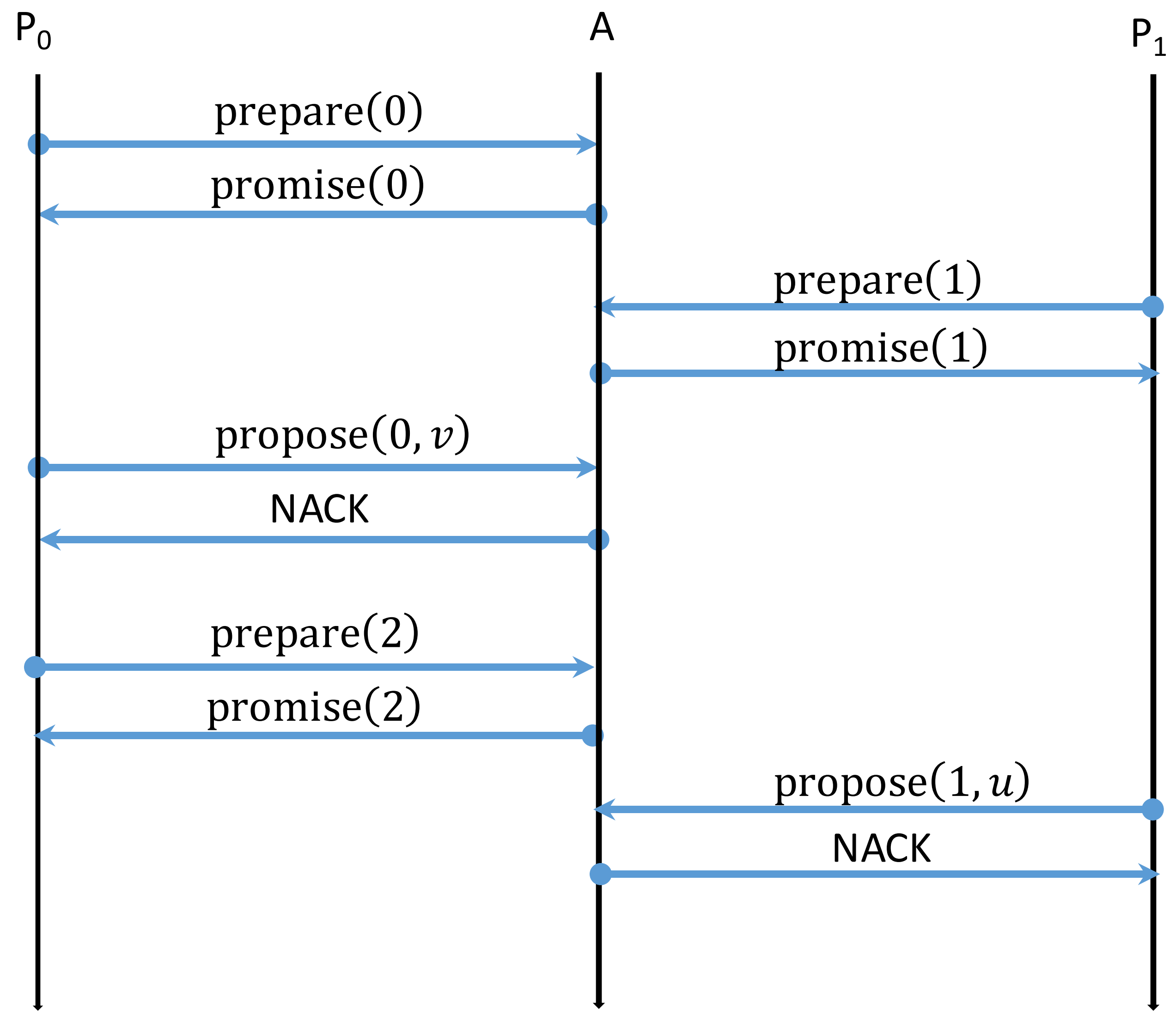
PAXOS guarantees safety (at most one value is chosen). Does it guarantee liveness (a value is chosen)?



Proposers P_0 and P_1 can keep interfering one with the other indefinitely long.

Liveness

PAXOS guarantees safety (at most one value is chosen). Does it guarantee liveness (a value is chosen)?

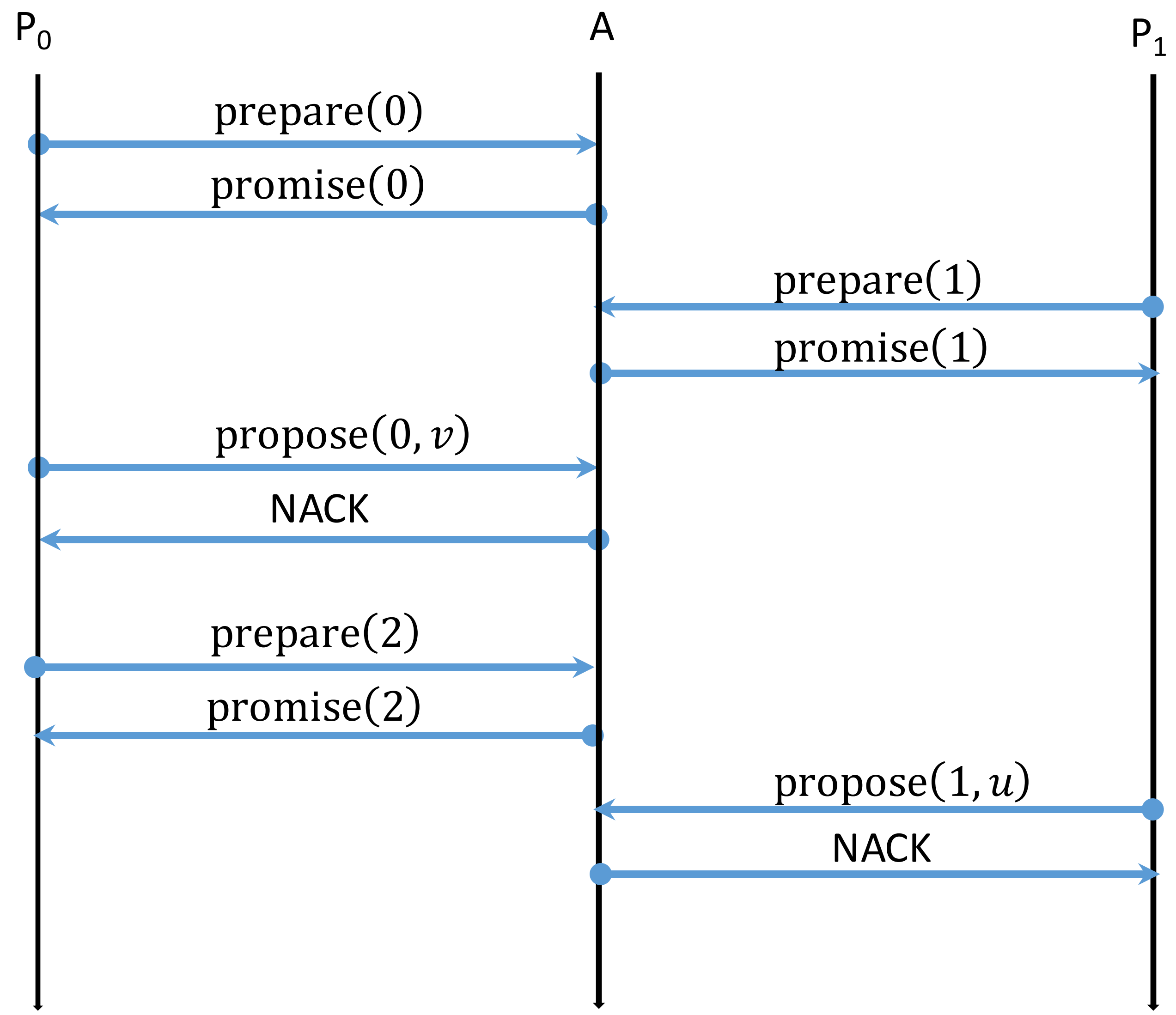


Proposers P_0 and P_1 can keep interfering one with the other indefinitely long.

Fischer, Lynch, Paterson: Impossibility of distributed consensus with one faulty process.

Liveness

PAXOS guarantees safety (at most one value is chosen). Does it guarantee liveness (a value is chosen)?



Proposers P_0 and P_1 can keep interfering one with the other indefinitely long.

Fischer, Lynch, Paterson: Impossibility of distributed consensus with one faulty process.

In practice, this problem can be solved by electing one process to issue all proposals.

The election can be made sufficiently reliable by adding random delays to candidate proposals.

Cf. jitter in exponential backoffs.

Wrong ways to handle FS corruption

The paper [1] ran an experiment where the data of the following applications was located on a file system that would randomly corrupt the content of blocks:

- Redis,
- ZooKeeper,
- Cassandra,
- Kafka,
- RethinkDB,
- LogCabin.

[1] https://www.usenix.org/system/files/conference/fast17/fast17-ganesan.pdf
[2] https://www.usenix.org/system/files/conference/fast18/fast18-alagappan.pdf

Wrong ways to handle FS corruption

The paper [1] ran an experiment where the data of the following applications was located on a file system that would randomly corrupt the content of blocks:

- Redis,
  - Does not validate checksums of user data,
  - Replicates corrupted data across nodes of a cluster,
  - Corruptions of a FS are “handled” with assert()s.
- ZooKeeper,
- Cassandra,
  - Has no checksums for uncompressed data,
  - When the checksum does not match, Cassandra chooses the last write as “the correct one”. This way it can replicate corrupted data to other cluster nodes.
- Kafka,
- RethinkDB,
- LogCabin.

[1] https://www.usenix.org/system/files/conference/fast17/fast17-ganesan.pdf
[2] https://www.usenix.org/system/files/conference/fast18/fast18-alagappan.pdf

Wrong ways to handle FS corruption

The paper [1] ran an experiment where the data of the following applications was located on a file system that would randomly corrupt the content of blocks:

- Redis,
    - Does not validate checksums of user data,
    - Replicates corrupted data across nodes of a cluster,
    - Corruptions of a FS are “handled” with assert()s.
- ZooKeeper,
- Cassandra,
    - Has no checksums for uncompressed data,
    - When the checksum does not match, Cassandra chooses the last write as “the correct one”. This way it can replicate corrupted data to other cluster nodes.
- Kafka,
- RethinkDB,
- LogCabin.

Mistakes in Redis and Cassandra highlight the importance of the model of faults that PAXOS protects from.
PAXOS assumes fail-stop participants and a network that does not corrupt messages.

[1] https://www.usenix.org/system/files/conference/fast17/fast17-ganesan.pdf
[2] https://www.usenix.org/system/files/conference/fast18/fast18-alagappan.pdf

PAXOS made live

1. Participants must adhere to PAXOS even if their RAM or disks have corruption.

PAXOS made live

1. Participants must adhere to PAXOS even if their RAM or disks have corruption.
2. Master leases: Most of the load on distributed key-value storages comes from read requests. We want to serve such requests by plain reads from one node, without reaching consensus ”the result of read $\#i$ is x”.

PAXOS made live

1. Participants must adhere to PAXOS even if their RAM or disks have corruption.
2. Master leases: Most of the load on distributed key-value storages comes from read requests. We want to serve such requests by plain reads from one node, without reaching consensus ”the result of read $\#i$ is x”.
3. There must be a way to add and delete cluster nodes.

PAXOS made live

1. Participants must adhere to PAXOS even if their RAM or disks have corruption.
2. Master leases: Most of the load on distributed key-value storages comes from read requests. We want to serve such requests by plain reads from one node, without reaching consensus ”the result of read $\#i$ is x”.
3. There must be a way to add and delete cluster nodes.
4. The FSM log cannot grow infinitely long. Periodically, one must make snapshots of the FSM and truncate the journal.

To read

1. Distributed consensus revised.
   https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-935.pdf
2. PAXOS made live.
   https://www.cs.utexas.edu/users/lorenzo/corsi/cs380d/papers/paper2-1.pdf