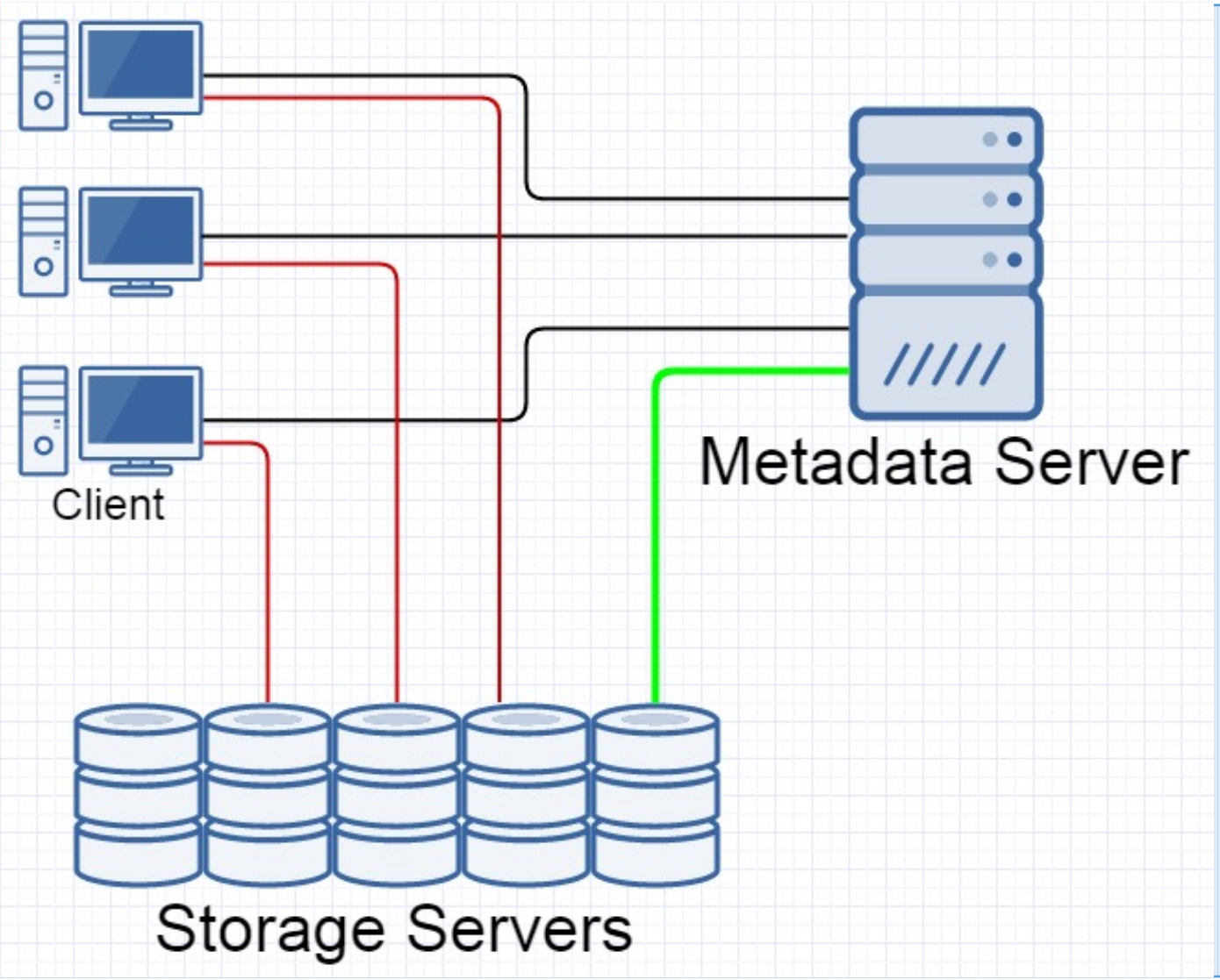
Основы построения файловых систем

Консенсус в распределённой системе

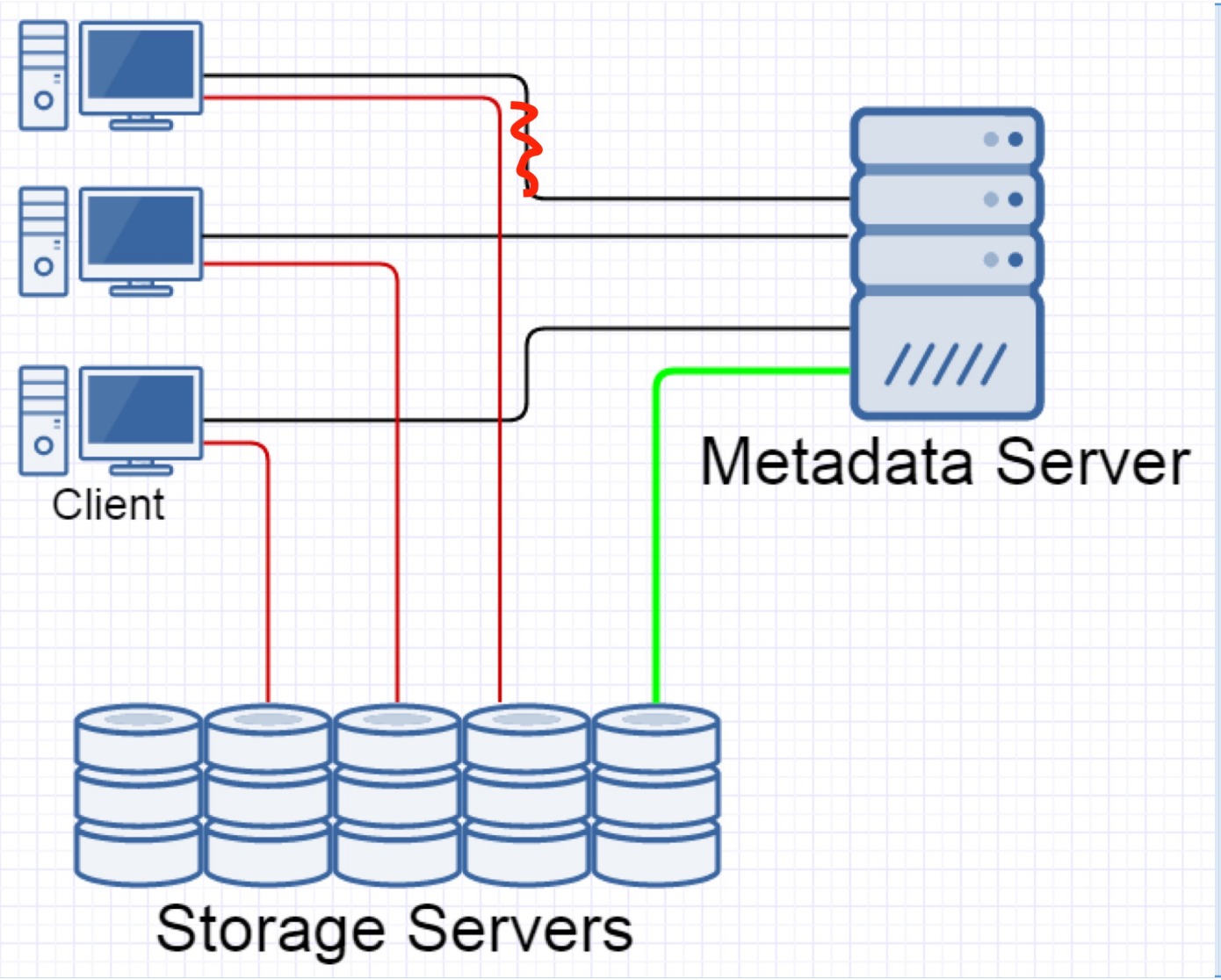
Сегодня мы обсудим, как сделать надёжный распределённый конечный автомат.

Напоминание о Parallel NFS



При открытии файла metadata-сервер сообщает клиенту “file layout”, т.е. где находятся данные файла и по какому протоколу они доступны.

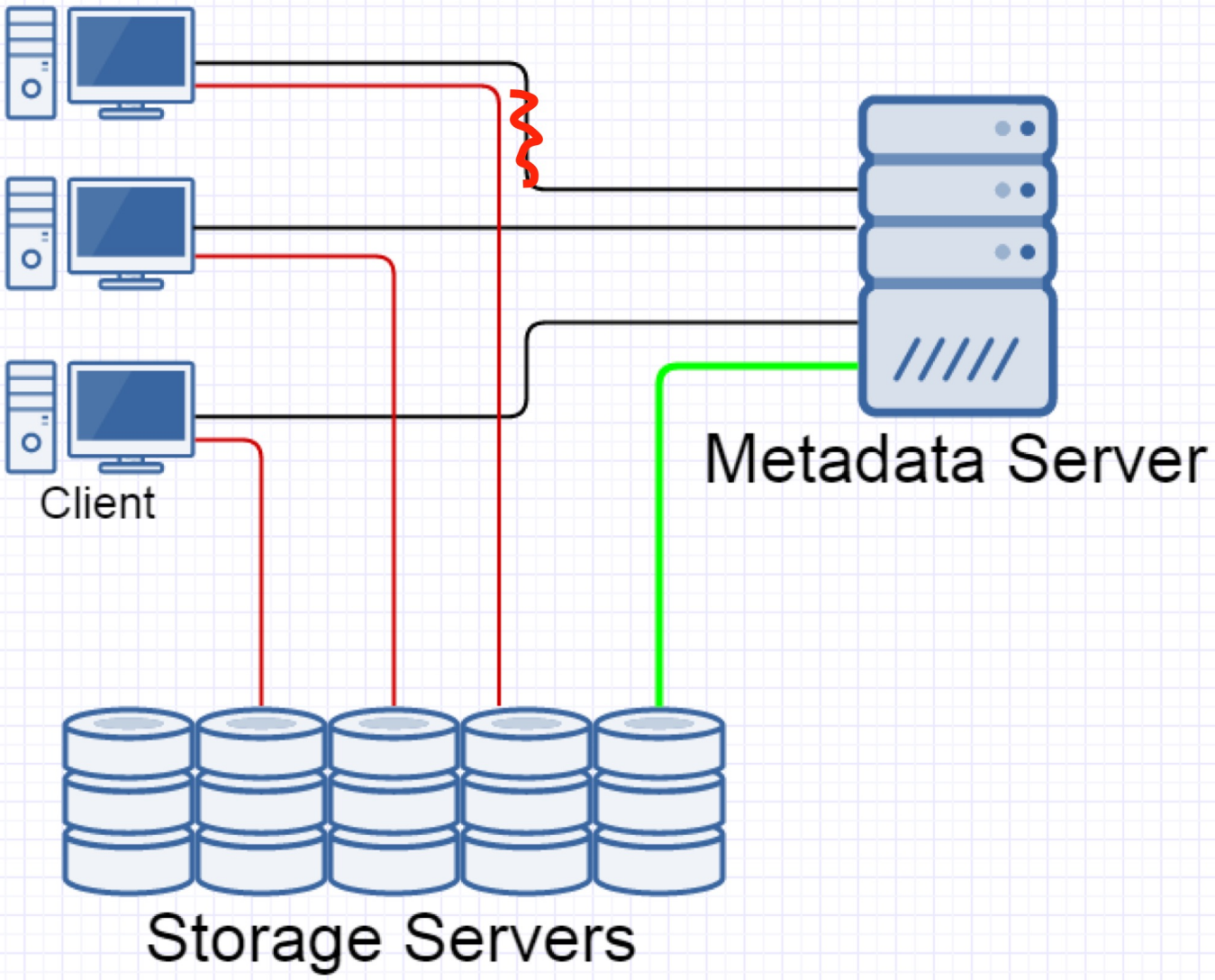
Напоминание о Parallel NFS



При открытии файла metadata-сервер сообщает клиенту “file layout”, т.е. где находятся данные файла и по какому протоколу они доступны.

Вопрос: что делать с клиентом, который потерял связь с metadata-сервером, но, потенциально, может ещё общаться со storage-сервером? Нельзя допускать ситуацию, когда два клиента модифицируют файл, не согласовываясь друг с другом.

Напоминание о Parallel NFS

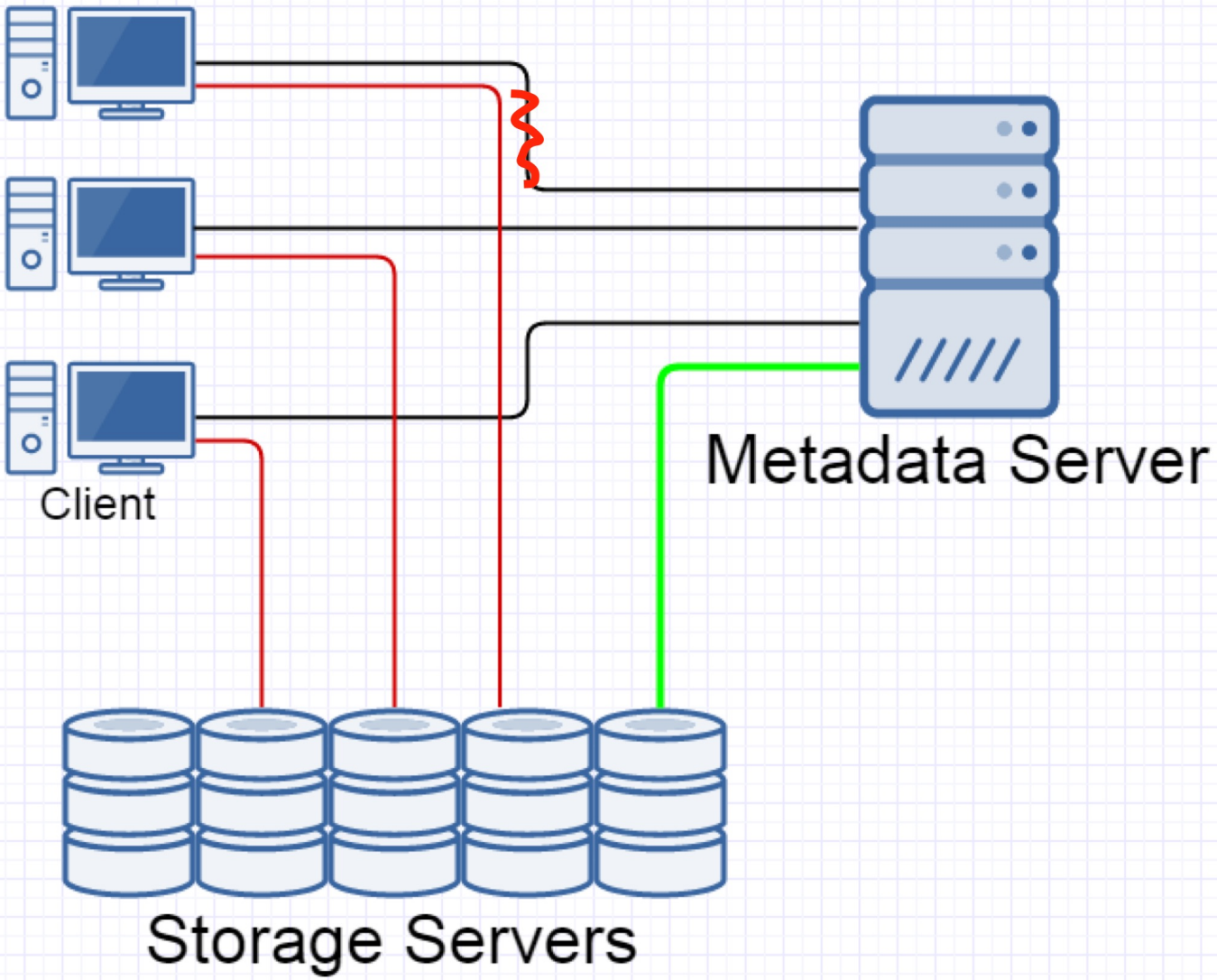


При открытии файла metadata-сервер сообщает клиенту “file layout”, т.е. где находятся данные файла и по какому протоколу они доступны.

Вопрос: что делать с клиентом, который потерял связь с metadata-сервером, но, потенциально, может ещё общаться со storage-сервером? Нельзя допускать ситуацию, когда два клиента модифицируют файл, не согласовываясь друг с другом.

Решение в NFSv4.1: каждому file layout-у назначается “layout lease” – период времени, в течение которого клиенту можно пользоваться layout-ом. Если клиент не продляет свой layout lease, то storage-сервер перестанет принимать от него запросы.

Напоминание о Parallel NFS



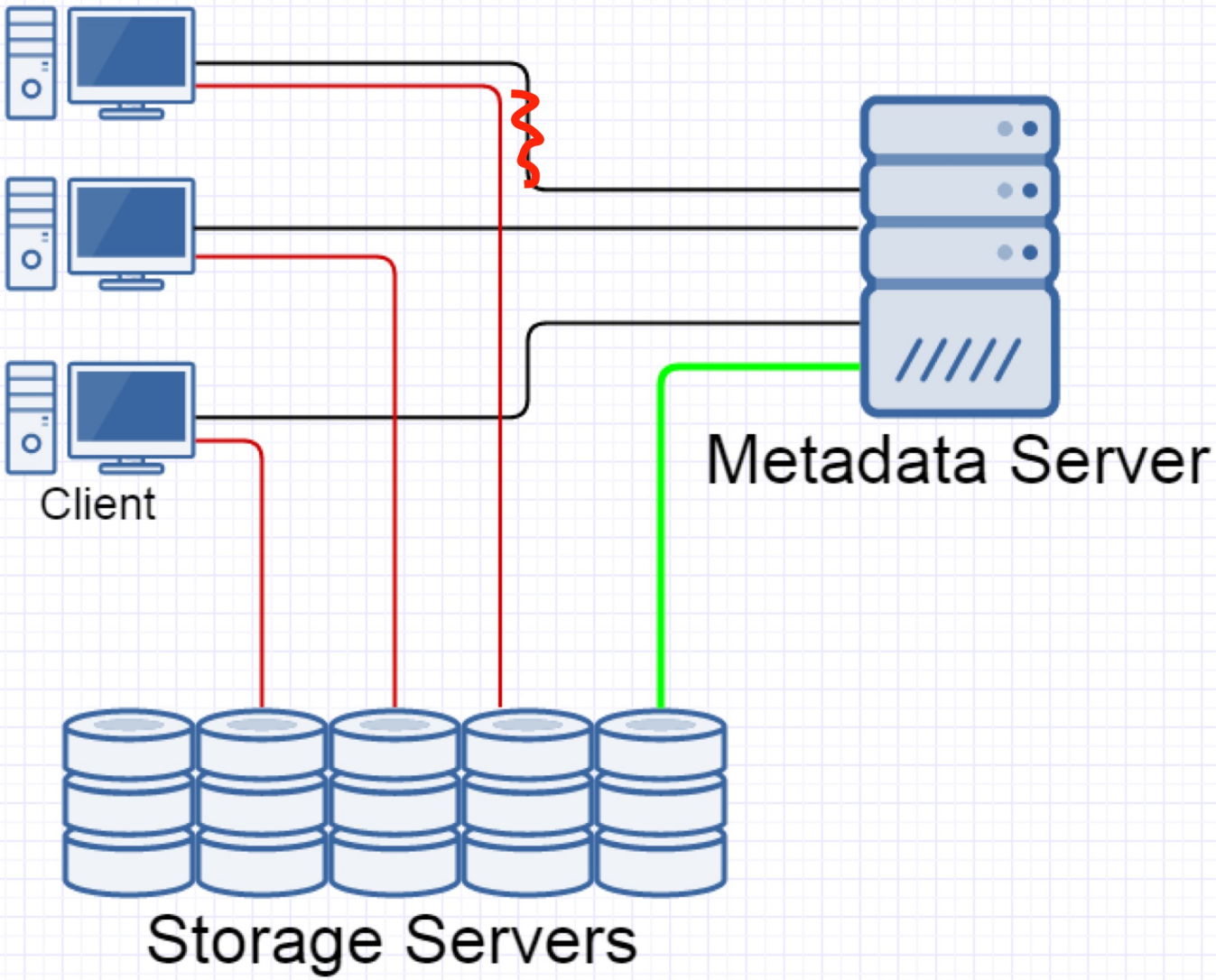
При открытии файла metadata-сервер сообщает клиенту “file layout”, т.е. где находятся данные файла и по какому протоколу они доступны.

Вопрос: что делать с клиентом, который потерял связь с metadata-сервером, но, потенциально, может ещё общаться со storage-сервером? Нельзя допускать ситуацию, когда два клиента модифицируют файл, не согласовываясь друг с другом.

Решение в NFSv4.1: каждому file layout-у назначается “layout lease” – период времени, в течение которого клиенту можно пользоваться layout-ом. Если клиент не продляет свой layout lease, то storage-сервер перестанет принимать от него запросы.

Вопрос: layout lease не должны полагаться на синхронность часов на клиентах, metadata-сервере и storage-серверах. Как этого достичь?

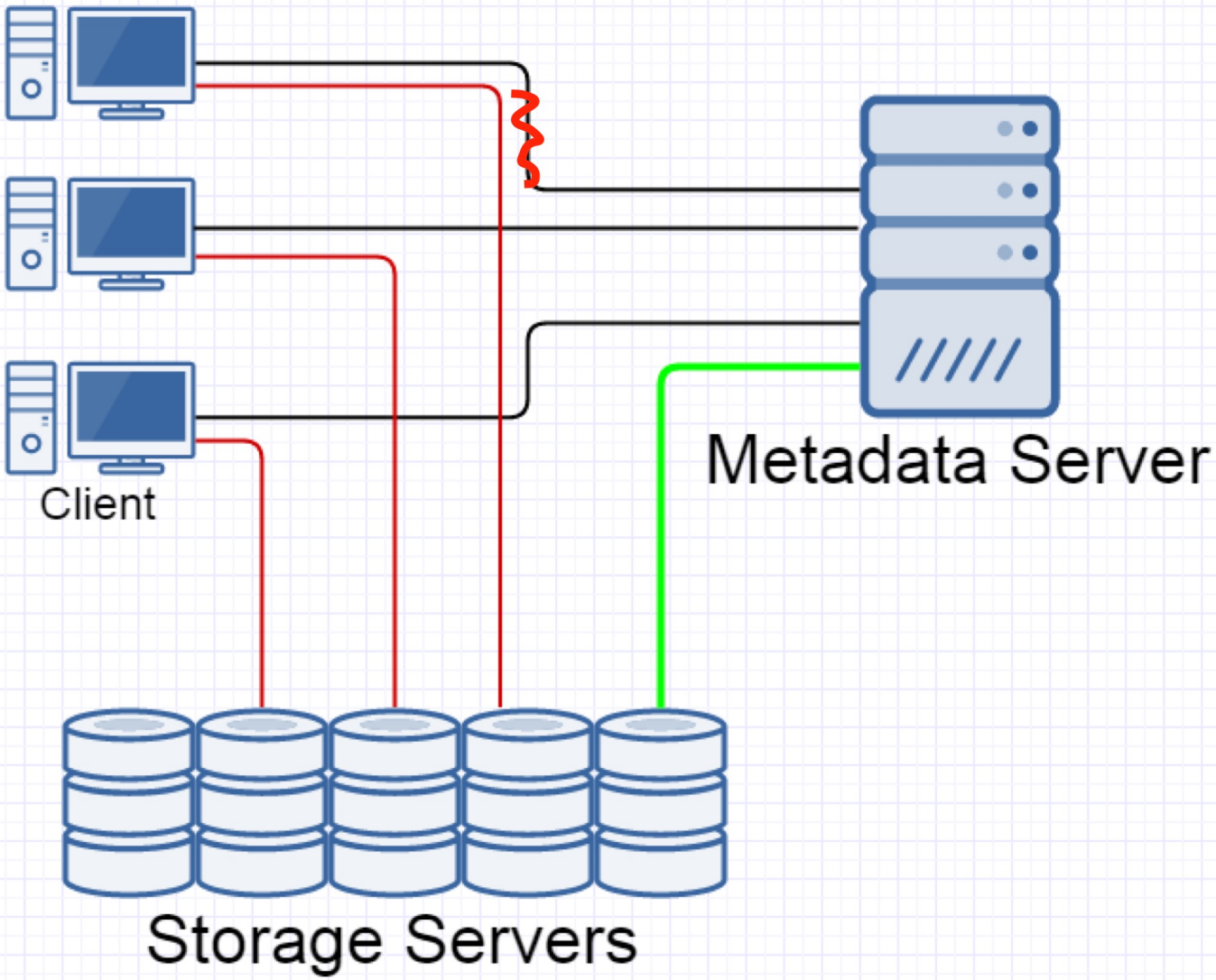
Напоминание о Parallel NFS



При открытии файла metadata-сервер сообщает клиенту “file layout”, т.е. где находятся данные файла и по какому протоколу они доступны.

Вопрос: проверка layout lease на стороне storage-серверов выглядит излишней сложностью. NFS-клиент по истечении lease мог бы сам прекращать делать IO. Верно?

Напоминание о Parallel NFS

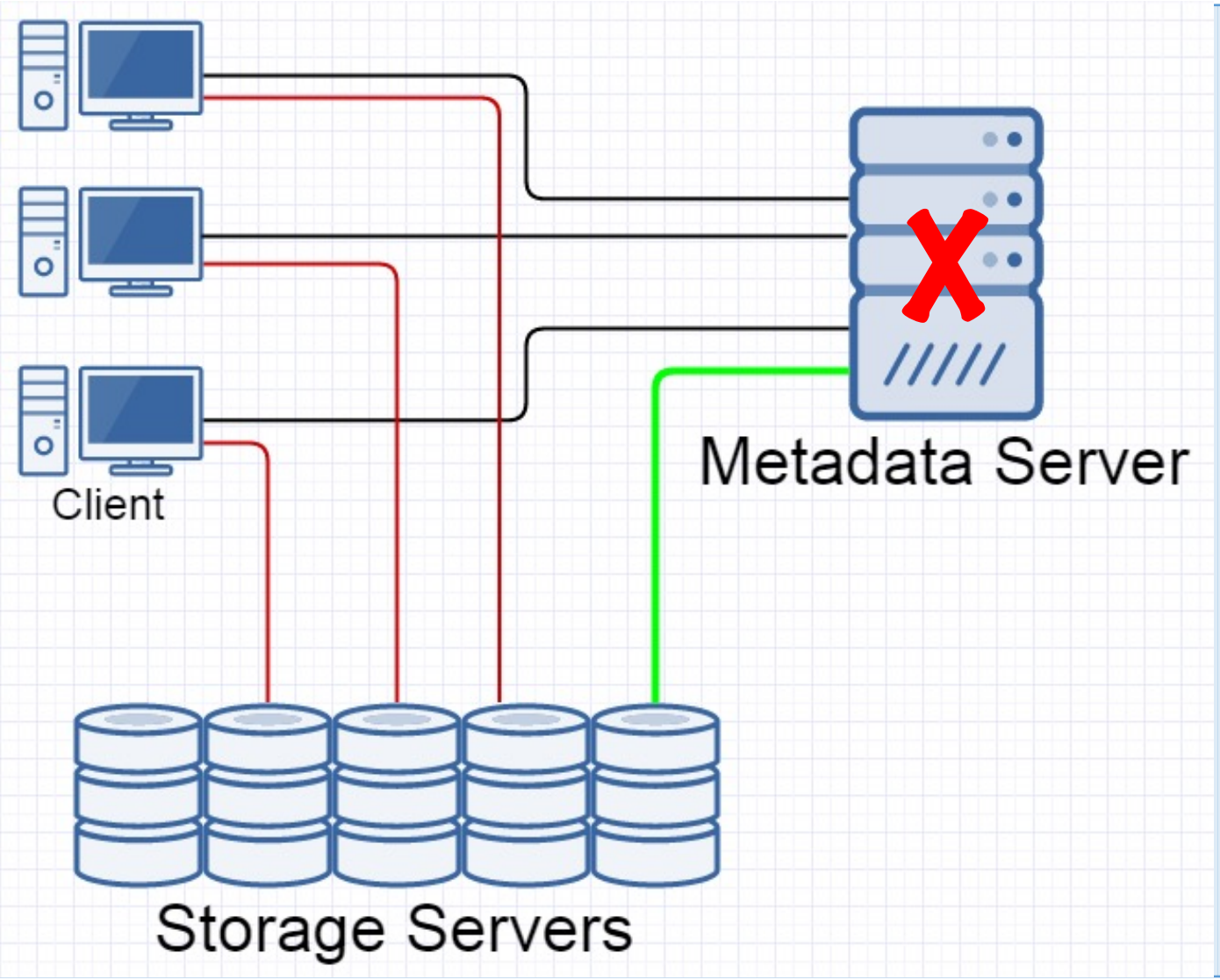


При открытии файла metadata-сервер сообщает клиенту “file layout”, т.е. где находятся данные файла и по какому протоколу они доступны.

Вопрос: проверка layout lease на стороне storage-серверов выглядит излишней сложностью. NFS-клиент по истечении lease мог бы сам прекращать делать IO. Верно?

См. также: client fencing.

Напоминание о Parallel NFS



Metadata-сервер является ”single point of failure”.

Вопрос: как реплицировать состояние metadata-сервера на несколько машин так, чтобы у всех участниках кластера было одно и то же состояние?

Консенсус в распределённой системе

Исходная задача: как реплицировать состояние metadata-сервера на несколько машин так, чтобы у всех участниках кластера было одно и то же состояние?

Консенсус в распределённой системе

Исходная задача: как реплицировать состояние metadata-сервера на несколько машин так, чтобы у всех участниках кластера было одно и то же состояние?

- На metadata-сервер можно смотреть как на детерминированный конечный автомат, состояниями которого являются образы ФС, а переходами между состояниями – операции вроде «создать файл», «удалить файл» и прочие.

Консенсус в распределённой системе

Исходная задача: как реплицировать состояние metadata-сервера на несколько машин так, чтобы у всех участниках кластера было одно и то же состояние?

- На metadata-сервер можно смотреть как на детерминированный конечный автомат, состояниями которого являются образы ФС, а переходами между состояниями – операции вроде «создать файл», «удалить файл» и прочие.
- На каждом переходе между состояниями реплики metadata-сервера должны договариваться о том, какой переход они исполняют. На каждом шаге все реплики должны выполнять один и тот же переход между состояниями.

Консенсус в распределённой системе

Исходная задача: как реплицировать состояние metadata-сервера на несколько машин так, чтобы у всех участниках кластера было одно и то же состояние?

- На metadata-сервер можно смотреть как на детерминированный конечный автомат, состояниями которого являются образы ФС, а переходами между состояниями – операции вроде «создать файл», «удалить файл» и прочие.
- На каждом переходе между состояниями реплики metadata-сервера должны договариваться о том, какой переход они исполняют. На каждом шаге все реплики должны выполнять один и тот же переход между состояниями.
- Состояние детерминированного конечного автомата определяется его историей перехода между состояниями. Чтобы реплицировать конечный автомат, достаточно научиться реплицировать журнал.

Консенсус в распределённой системе

Исходная задача: как реплицировать состояние metadata-сервера на несколько машин так, чтобы у всех участниках кластера было одно и то же состояние?

- На metadata-сервер можно смотреть как на детерминированный конечный автомат, состояниями которого являются образы ФС, а переходами между состояниями – операции вроде «создать файл», «удалить файл» и прочие.
- На каждом переходе между состояниями реплики metadata-сервера должны договариваться о том, какой переход они исполняют. На каждом шаге все реплики должны выполнять один и тот же переход между состояниями.
- Состояние детерминированного конечного автомата определяется его историей перехода между состояниями. Чтобы реплицировать конечный автомат, достаточно научиться реплицировать журнал.
- Чтобы вести реплицированный журнал, участникам кластера достаточно на каждом шаге выбирать значение, которое на этом шаге дописать в конец журнала.

Консенсус в распределённой системе

Исходная задача: как реплицировать состояние metadata-сервера на несколько машин так, чтобы у всех участниках кластера было одно и то же состояние?

- На metadata-сервер можно смотреть как на детерминированный конечный автомат, состояниями которого являются образы ФС, а переходами между состояниями – операции вроде «создать файл», «удалить файл» и прочие.
- На каждом переходе между состояниями реплики metadata-сервера должны договариваться о том, какой переход они исполняют. На каждом шаге все реплики должны выполнять один и тот же переход между состояниями.
- Состояние детерминированного конечного автомата определяется его историей перехода между состояниями. Чтобы реплицировать конечный автомат, достаточно научиться реплицировать журнал.
- Чтобы вести реплицированный журнал, участникам кластера достаточно на каждом шаге выбирать значение, которое на этом шаге дописать в конец журнала.
- Итак, достаточно решить задачу: узлы кластера должны договориться, какое значение на N-м шаге дописать в N-ю позицию в журнале.

Постановка задачи

Исходная задача: узлы кластера должны договориться, какое значение на N-м шаге дописать в N-ю позицию в журнале.

Постановка задачи

Исходная задача: узлы кластера должны договориться, какое значение на N-м шаге дописать в N-ю позицию в журнале.

Модельная задача: из множества *предложенных** значений *выбрать** одно.

** Значения слов «предложенных» и «выбрать» мы определим чуть позже.*

Постановка задачи

Исходная задача: узлы кластера должны договориться, какое значение на N-м шаге дописать в N-ю позицию в журнале.

Модельная задача: из множества *предложенных** значений *выбрать** одно.

Процессы-участники:

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

** Значения слов «предложенных» и «выбрать» мы определим чуть позже.*

Постановка задачи

Исходная задача: узлы кластера должны договориться, какое значение на N-м шаге дописать в N-ю позицию в журнале.

Модельная задача: из множества *предложенных** значений *выбрать** одно.

Процессы-участники:

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

Предложенное значение – значение, испущенное каким-либо proposer'ом.

Acceptor – процесс, принимающий от proposer'а предложения о значениях-кандидатах на выбор.

Выбранное значение – это значение, сообщённое процессу-learner'у.

В реальных кластерных системах каждый участник выполняет все три роли.

Постановка задачи

Исходная задача: узлы кластера должны договориться, какое значение на N-м шаге дописать в N-ю позицию в журнале.

Модельная задача: из множества *предложенных** значений *выбрать** одно.

Процессы-участники:

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

Proposer можно себе мыслить как источник запросов «на шаге N надо создать файл такой-то» или «на шаге N надо выделить блокировку на файле такому-то клиенту».

Постановка задачи

Исходная задача: узлы кластера должны договориться, какое значение на N-м шаге дописать в N-ю позицию в журнале.

Модельная задача: из множества *предложенных** значений *выбрать** одно.

Процессы-участники:
1. proposers,
2. acceptors,
3. learners.

Модель сети и ошибок:
1. процессы-участники могут работать с произвольной скоростью,
2. процессы могут умирать и перезапускаться в произвольное время,
3. сообщения могут быть произвольно задержаны, потеряны или дублированы.

Наводящие соображения

Будем считать значение выбранным, если его *приняло* большинство процессов-acceptor’ов.

Наводящие соображения

Будем считать значение выбранным, если его *приняло* большинство процессов-acceptor'ов.

Это определение работает, если выполнено

Требование 0: процесс-acceptor может принять не более одного предложенного значения.

Действительно, во множестве из N элементов всякие два подмножества из $\lfloor N/2 \rfloor + 1$ элементов пересекаются. Поэтому, если два большинства acceptor'ов M_0 и M_1 приняли значения v_0 и v_1, соответственно, то acceptor, лежащий в пересечении $M_0 \cap M_1$, принял оба значения, т.е. $v_0 = v_1$ и это значение принято всеми acceptor'ами из $M_0 \cup M_1$.

Наводящие соображения

1. Если из всех процессов-proposer’ов только один процесс предложил значение, то это значение и должно быть выбрано.
2. В такой ситуации процессы-acceptor’ы получат от proposer’а только одно предложение о выбираемом значении.

Наводящие соображения

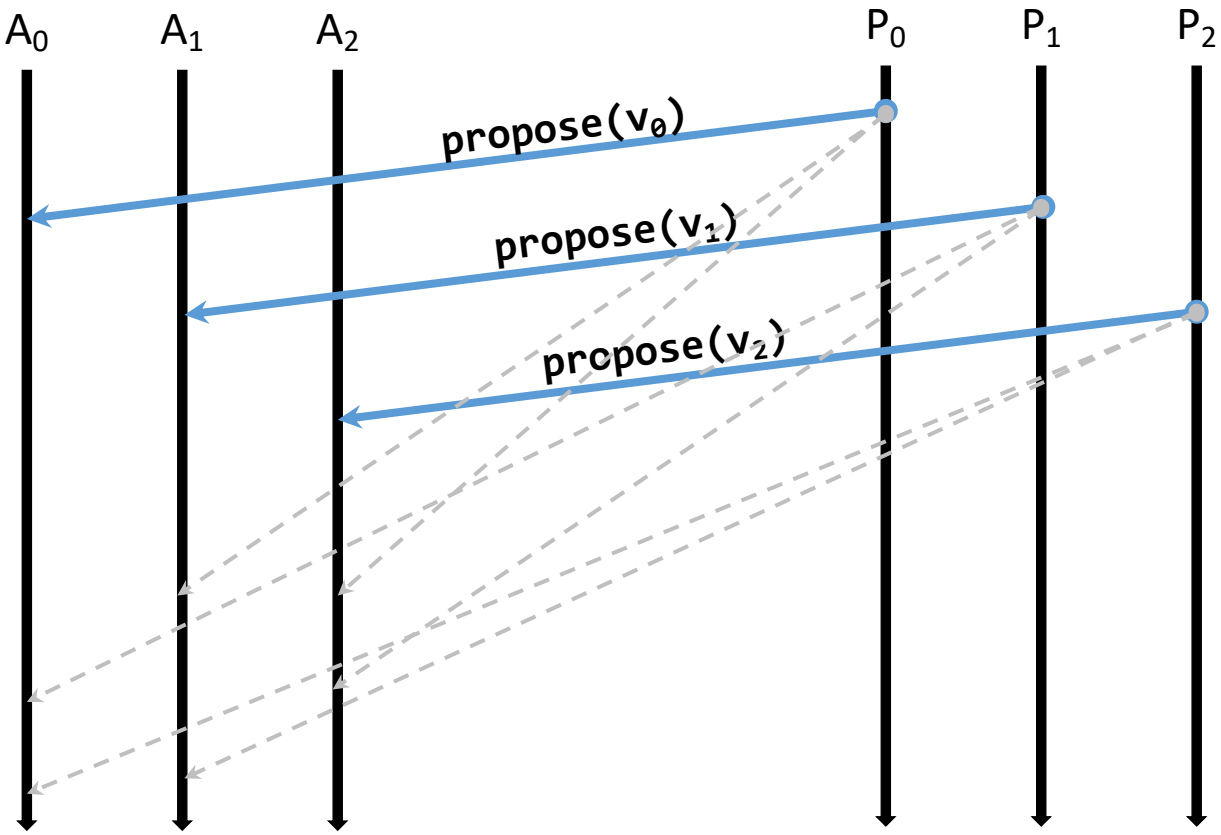
1. Если из всех процессов-proposer’ов только один процесс предложил значение, то это значение и должно быть выбрано.
2. В такой ситуации процессы-acceptor’ы получат от proposer’а только одно предложение о выбираемом значении.

Требование 1: процесс-acceptor обязан принять первое предложенное ему значение.

Наводящие соображения

1. Если из всех процессов-proposer’ов только один процесс предложил значение, то это значение и должно быть выбрано.
2. В такой ситуации процессы-acceptor’ы получат от proposer’а только одно предложение о выбираемом значении.

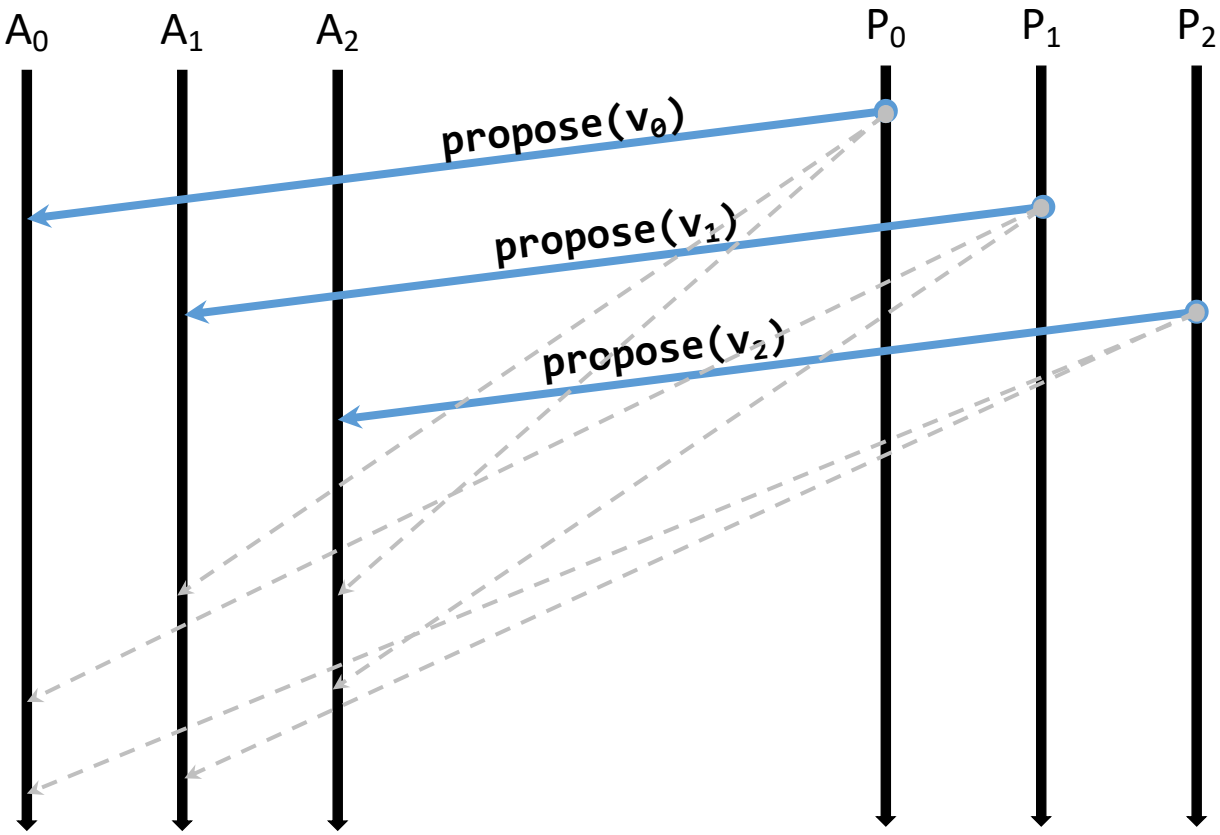
Требование 1: процесс-acceptor обязан принять первое предложенное ему значение.



Наводящие соображения

1. Если из всех процессов-proposer’ов только один процесс предложил значение, то это значение и должно быть выбрано.
2. В такой ситуации процессы-acceptor’ы получат от proposer’а только одно предложение о выбираемом значении.

Требование 1: процесс-acceptor обязан принять первое предложенное ему значение.

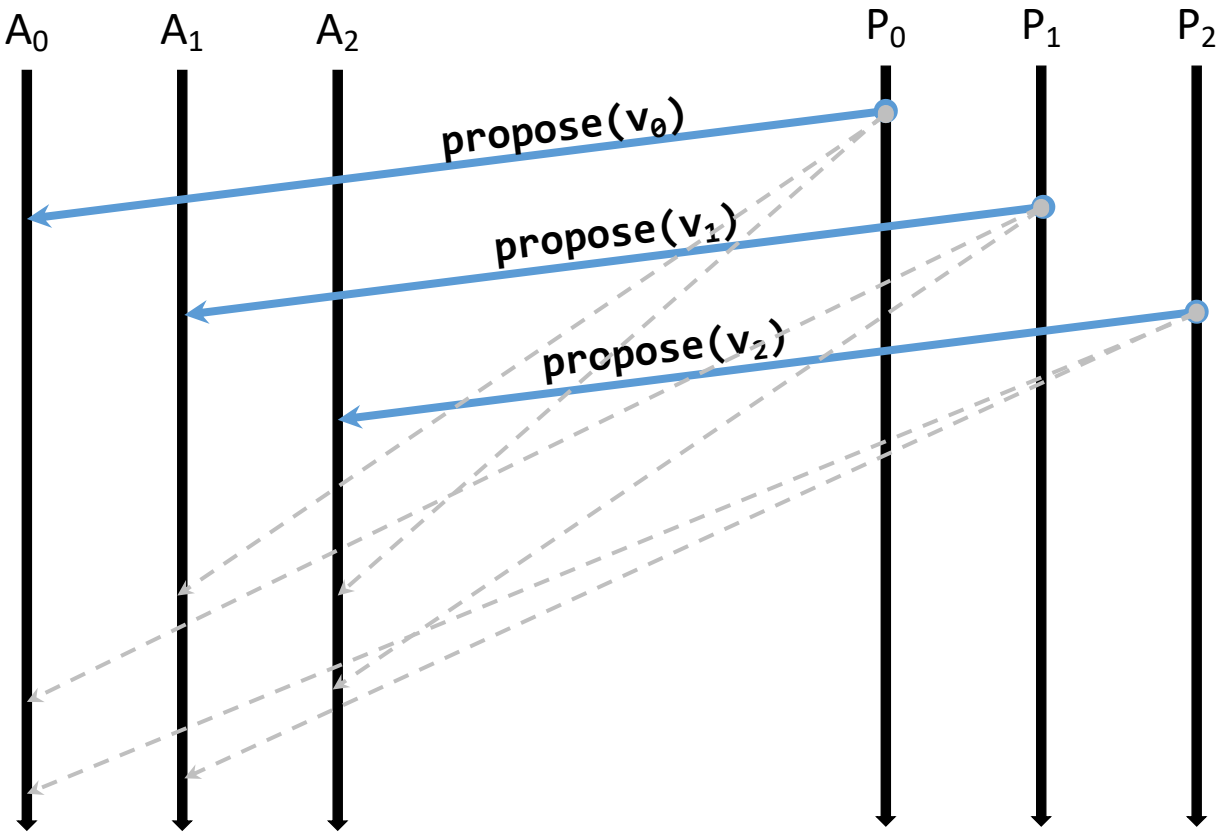


Каждый acceptor принял своё значение.
Большинства никакое значение уже не наберёт.

Наводящие соображения

1. Если из всех процессов-proposer’ов только один процесс предложил значение, то это значение и должно быть выбрано.
2. В такой ситуации процессы-acceptor’ы получат от proposer’а только одно предложение о выбираемом значении.

Требование 1: процесс-acceptor обязан принять первое предложенное ему значение.



Каждый acceptor принял своё значение. Большинства никакое значение уже не наберёт.

Идея: правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer’а. Acceptor’ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer’ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Наводящие соображения

Правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer'а. Acceptor'ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer'ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Наводящие соображения

Правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer'а. Acceptor'ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer'ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Требование 2а: в пределах одного процесса-proposer'а каждый запрос `propose(n, v)` использует номер эпохи, строго больший, чем номера, использованные предыдущими запросами.

Наводящие соображения

Правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer'а. Acceptor'ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer'ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Требование 2a: в пределах одного процесса-proposer'а каждый запрос `propose(n, v)` использует номер эпохи, строго больший, чем номера, использованные предыдущими запросами.
Требование 2b: разные процессы-proposer'ы выбирают номера эпох из непересекающихся множеств*, чтобы обеспечить глобальную уникальность «временных меток».

** В системе с N proposer'ами можно взять множества эпох вида $\{0, N, 2N, \dots\}$, $\{1, N+1, 2N+1, \dots\}$, …*

Наводящие соображения

Правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer'а. Acceptor'ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer'ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Напоминание (требование 0): процесс-acceptor может принять не более одного предложенного значения.

Наводящие соображения

Правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer'а. Acceptor'ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer'ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Напоминание (требование 0): процесс-acceptor может принять не более одного предложенного значения.

Требование 0': процесс-acceptor может принимать несколько значений, если они отличаются только эпохой: $(n_0, v), (n_1, v), \ldots (n_k, v)$ для $n_0 < n_1 < \ldots n_k$.

Наводящие соображения

Правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer'а. Acceptor'ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer'ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Напоминание: Будем считать значение выбранным, если его *приняло* большинство процессов-acceptor'ов.

Наводящие соображения

Правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer'а. Acceptor'ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer'ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Напоминание: Будем считать значение выбранным, если его *приняло* большинство процессов-acceptor'ов.

Поскольку мы разрешили acceptor'ам принимать несколько предложений вида (n, v) и (n', v), мы должны разрешить ситуацию, когда большинство accepter'ов выбрало пары $\{(n_i, v)\}$ с разными номерами эпох. Надо только потребовать, чтобы у всех выбранных пар значение v было одно и то же.

Наводящие соображения

Правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer'а. Acceptor'ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer'ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Алгоритм proposer'а:

1. Выбрать номер эпохи n, не использованный ранее.
2. Разослать acceptor'ам сообщение $\mathrm{prepare}(n)$, т.е. просьбу не принимать значения с эпохой $< n$.

Алгоритм acceptor'а:

Наводящие соображения

Правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer'а. Acceptor'ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer'ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Алгоритм proposer'а:

1. Выбрать номер эпохи n, не использованный ранее.
2. Разослать acceptor'ам сообщение $\mathrm{prepare}(n)$, т.е. просьбу не принимать значения с эпохой $< n$.

Алгоритм acceptor'а:

1. При получении запроса $\mathrm{prepare}(n)$ запомнить номер эпохи n, чтобы не принимать запросы с меньшими номерами. В ответ отослать сообщение $\mathrm{promise}(n, m, v)$, где (m, v) – принятое ранее значение, или (nil, nil), если принятого значения ещё нет.

Наводящие соображения

Правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer'а. Acceptor'ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer'ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Алгоритм proposer'а:

1. Выбрать номер эпохи n, не использованный ранее.
2. Разослать acceptor'ам сообщение $\mathrm{prepare}(n)$, т.е. просьбу не принимать значения с эпохой $< n$.

Алгоритм acceptor'а:

1. При получении запроса $\mathrm{prepare}(n)$ запомнить номер эпохи n, чтобы не принимать запросы с меньшими номерами. В ответ отослать сообщение $\mathrm{promise}(n, m, v)$, где (m, v) – принятое ранее значение, или (nil, nil), если принятого значения ещё нет.
   Замечание: в ответе $\mathrm{promise}(n, m, v)$ обязательно $n > m$.

Наводящие соображения

Правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer'а. Acceptor'ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer'ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Алгоритм proposer'а:

1. Выбрать номер эпохи n, не использованный ранее.
2. Разослать acceptor'ам сообщение $\mathrm{prepare}(n)$, т.е. просьбу не принимать значения с эпохой $< n$.
3. Дождаться ответов $\mathrm{promise}(n, m_i, v_i)$ от большинства acceptor'ов и **выбрать значение v, соответствующее наибольшему номеру эпохи**. Если все $v_i = nil$, можно выбрать своё значение.
4. Разослать **ответившим** acceptor'ам запрос $\mathrm{propose}(n, v)$.

Алгоритм acceptor'а:

1. При получении запроса $\mathrm{prepare}(n)$ запомнить номер эпохи n, чтобы не принимать запросы с меньшими номерами. В ответ отослать сообщение $\mathrm{promise}(n, m, v)$, где (m, v) – принятое ранее значение, или (nil, nil), если принятого значения ещё нет.
   Замечание: в ответе $\mathrm{promise}(n, m, v)$ обязательно $n > m$.

Наводящие соображения

Правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer'а. Acceptor'ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer'ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Алгоритм proposer'а:

1. Выбрать номер эпохи n, не использованный ранее.
2. Разослать acceptor'ам сообщение $\mathrm{prepare}(n)$, т.е. просьбу не принимать значения с эпохой $< n$.
3. Дождаться ответов $\mathrm{promise}(n, m_i, v_i)$ от большинства acceptor'ов и **выбрать значение v, соответствующее наибольшему номеру эпохи**. Если все $v_i = nil$, можно выбрать своё значение.
4. Разослать **ответившим** acceptor'ам запрос $\mathrm{propose}(n, v)$.

Алгоритм acceptor'а:

1. При получении запроса $\mathrm{prepare}(n)$ запомнить номер эпохи n, чтобы не принимать запросы с меньшими номерами. В ответ отослать сообщение $\mathrm{promise}(n, m, v)$, где (m, v) – принятое ранее значение, или (nil, nil), если принятого значения ещё нет.
   Замечание: в ответе $\mathrm{promise}(n, m, v)$ обязательно $n > m$.
2. При получении запроса $\mathrm{propose}(k, w)$, где k больше запомненного номера эпохи, принять значение (k, w) и ответить $\mathrm{accept}(k, w)$.

Наводящие соображения

Правильно предлагать не сами значения v, а пары (n, v), где n – натуральное число, номер эпохи proposer'а. Acceptor'ы должны принимать пары (n, v) и (n', v), где предлагаемое значение v одно и то же.

Дополнительная нумерация позволяет ранжировать предложения по времени их появления. Процессы-proposer'ы могут использовать эту нумерацию, чтобы договориться **не предлагать заведомо устаревшие значения**.

Алгоритм proposer'а:

1. Выбрать номер эпохи n, не использованный ранее.
2. Разослать acceptor'ам сообщение $\mathrm{prepare}(n)$, т.е. просьбу не принимать значения с эпохой $< n$.
3. Дождаться ответов $\mathrm{promise}(n, m_i, v_i)$ от большинства acceptor'ов и **выбрать значение v, соответствующее наибольшему номеру эпохи**. Если все $v_i = nil$, можно выбрать своё значение.
4. Разослать **ответившим** acceptor'ам запрос $\mathrm{propose}(n, v)$.
5. Дождаться ответов accept от большинства acceptor'ов.
6. В случае таймаута вернуться к #1.

Алгоритм acceptor'а:

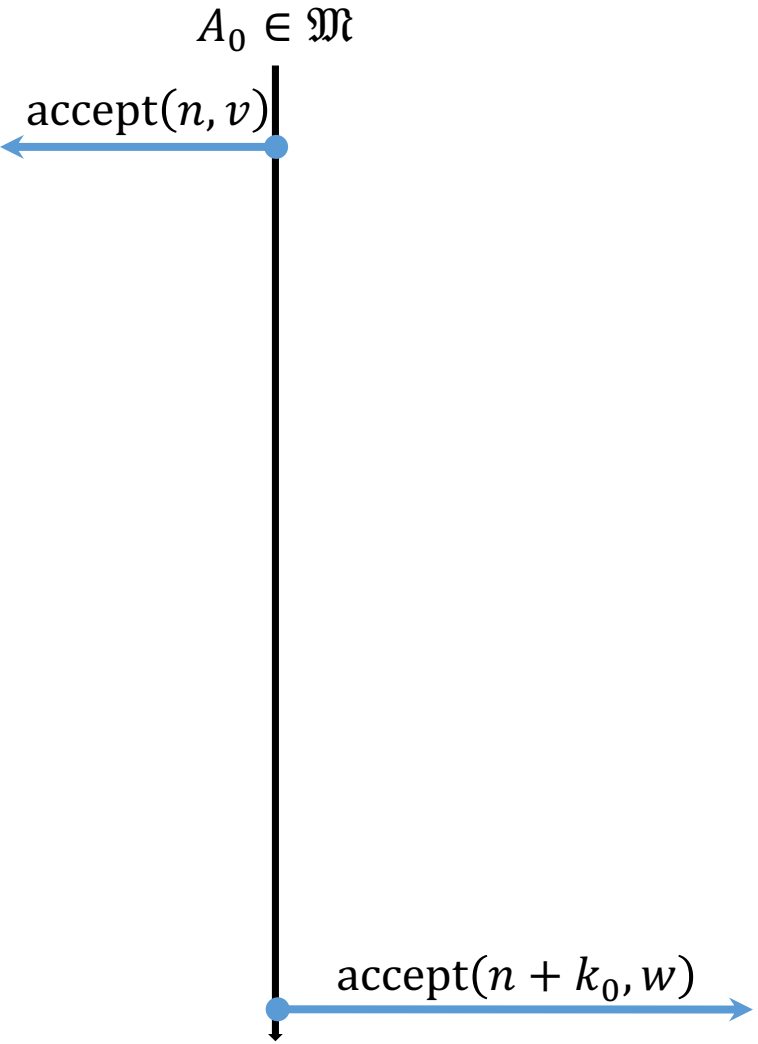
1. При получении запроса $\mathrm{prepare}(n)$ запомнить номер эпохи n, чтобы не принимать запросы с меньшими номерами. В ответ отослать сообщение $\mathrm{promise}(n, m, v)$, где (m, v) – принятое ранее значение, или (nil, nil), если принятого значения ещё нет.
   Замечание: в ответе $\mathrm{promise}(n, m, v)$ обязательно $n > m$.
2. При получении запроса $\mathrm{propose}(k, w)$, где k больше запомненного номера эпохи, принять значение (k, w) и ответить $\mathrm{accept}(k, w)$.

Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.

Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.

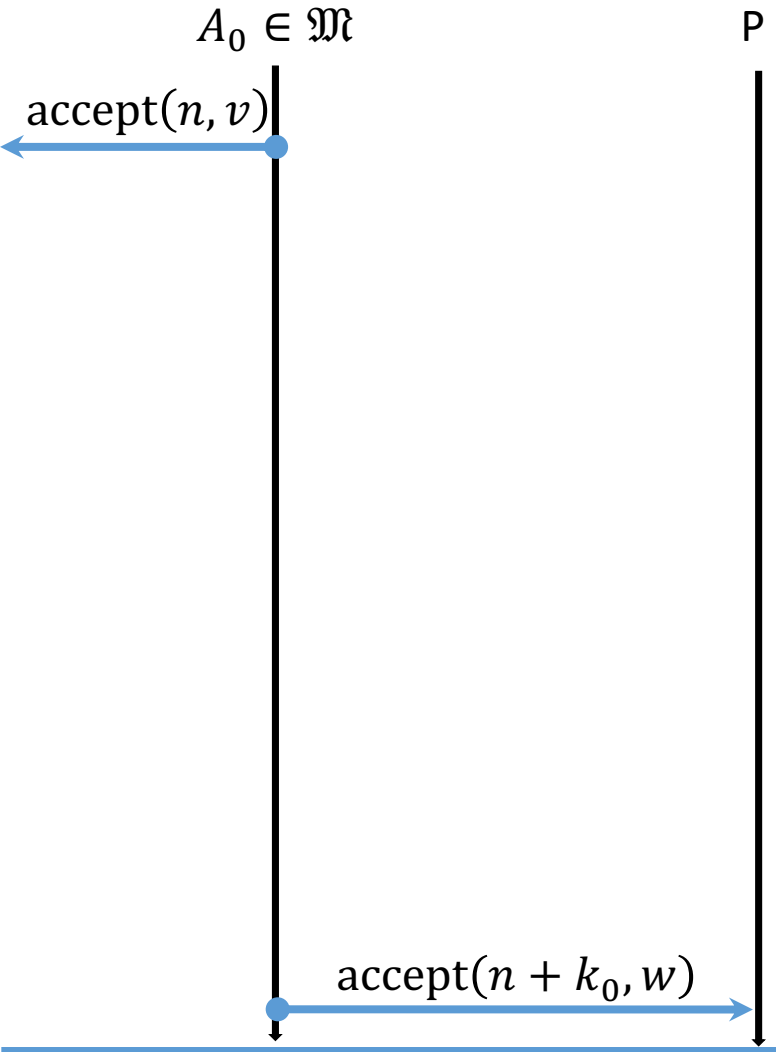


Замечание: мы рассматриваем **минимальную** эпоху $n + k_0$, в которую испущен $\text{accept}(n + k_0, w)$.

В эту эпоху все остальные acceptor'ы из $\mathfrak{M}$ ещё не отправляли accept'ы после $\text{accept}(n, v)$.

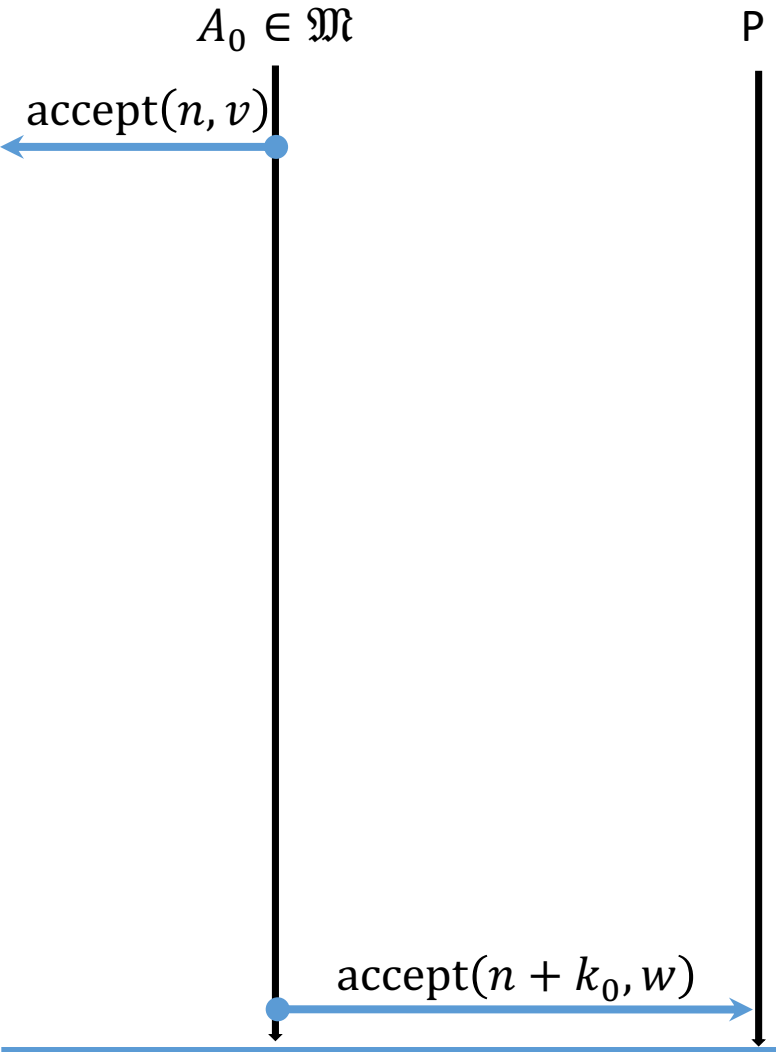
Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.



Корректность алгоритма

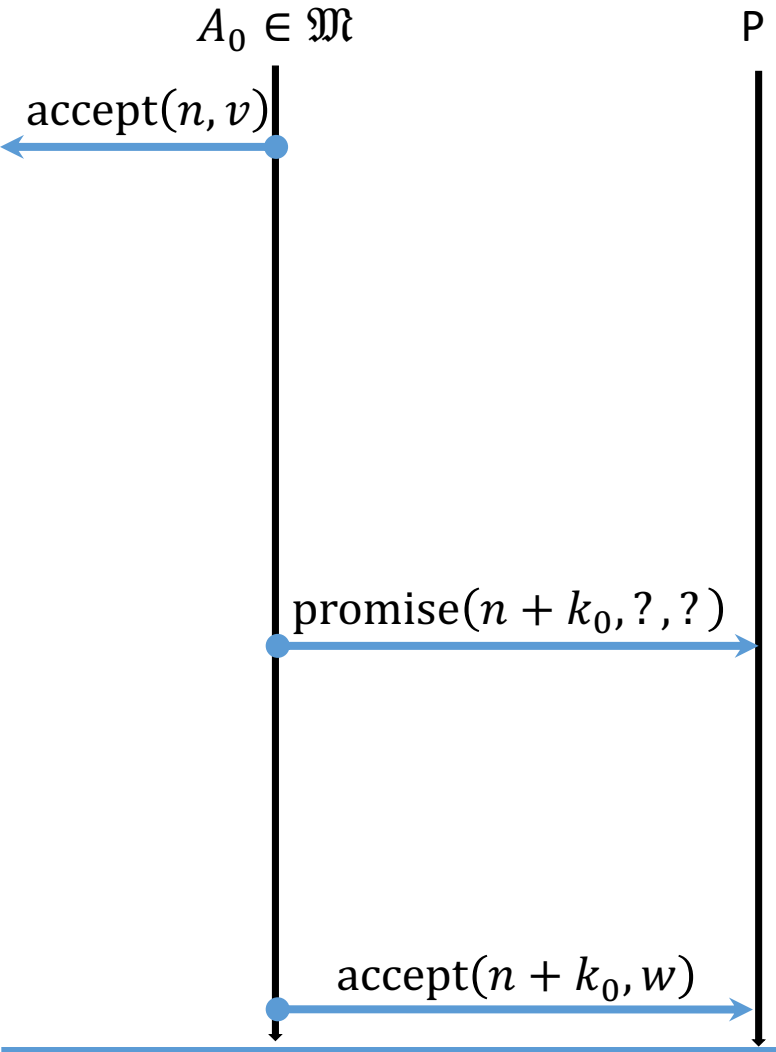
Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.



1. Сообщение $\text{accept}(n + k_0, w)$ посылается только в ответ на $\text{propose}(n + k_0, w)$ от P.
2. P отправляет propose только тем acceptor'ам, которые ответили promise.

Корректность алгоритма

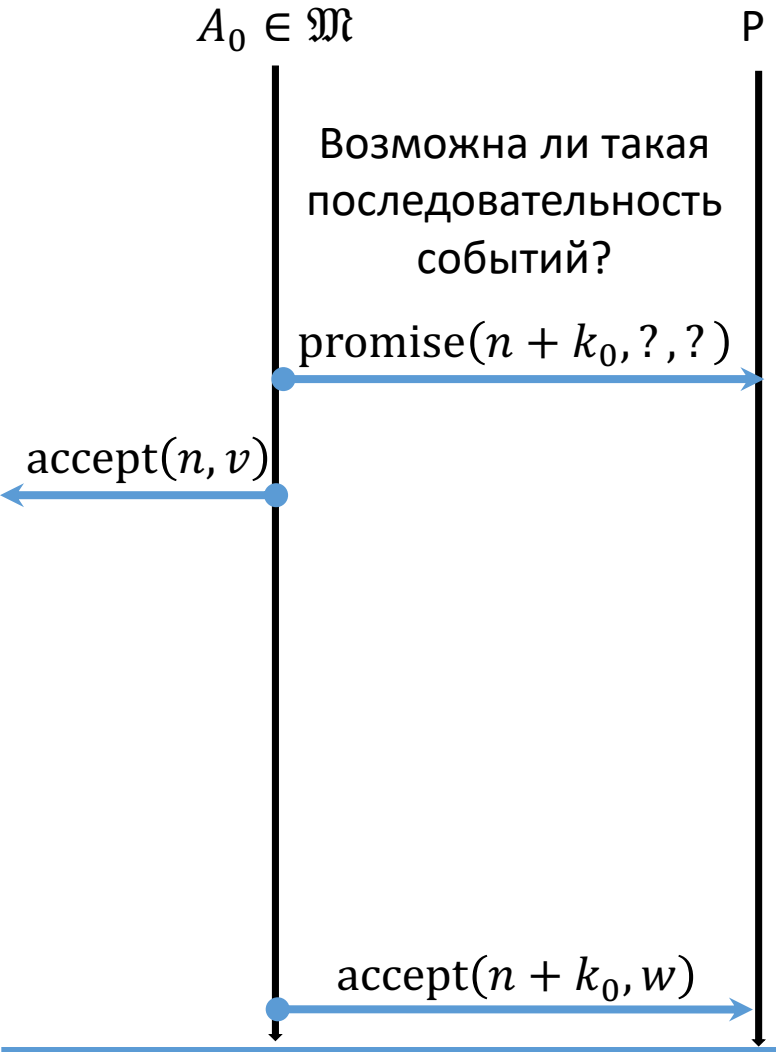
Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.



1. Сообщение accept$(n + k_0, w)$ посылается только в ответ на propose$(n + k_0, w)$ от P.
2. P отправляет propose только тем acceptor'ам, которые ответили promise.

Корректность алгоритма

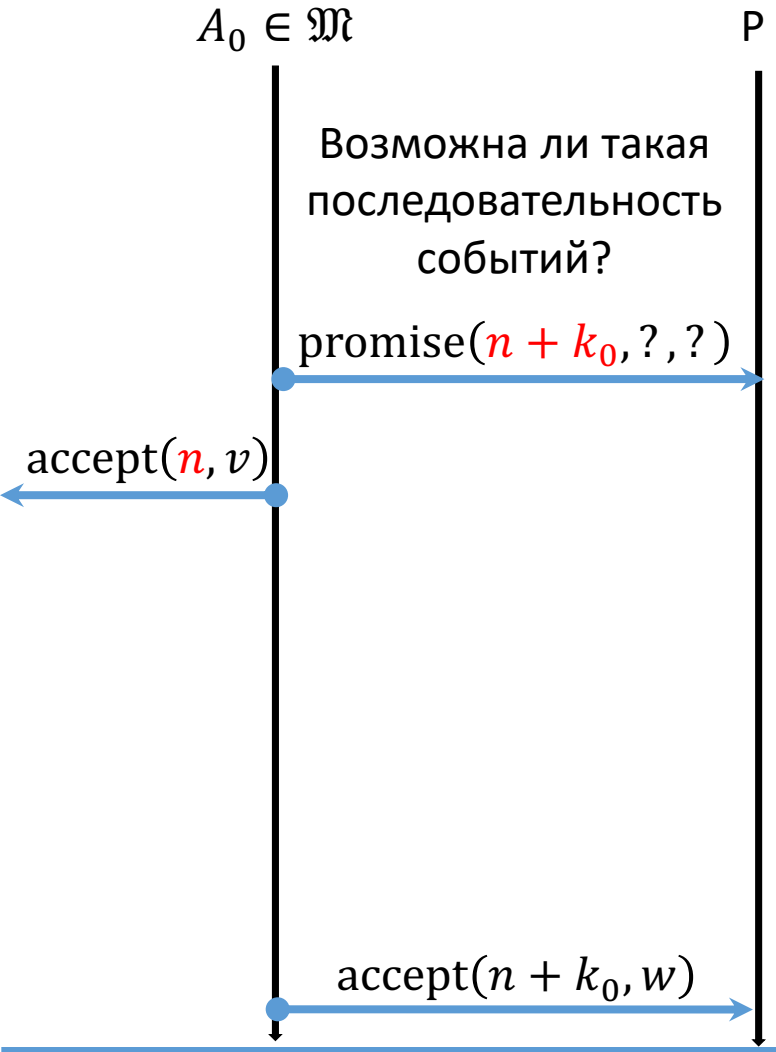
Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.



1. Сообщение $\text{accept}(n + k_0, w)$ посылается только в ответ на $\text{propose}(n + k_0, w)$ от P.
2. P отправляет propose только тем acceptor'ам, которые ответили promise.

Корректность алгоритма

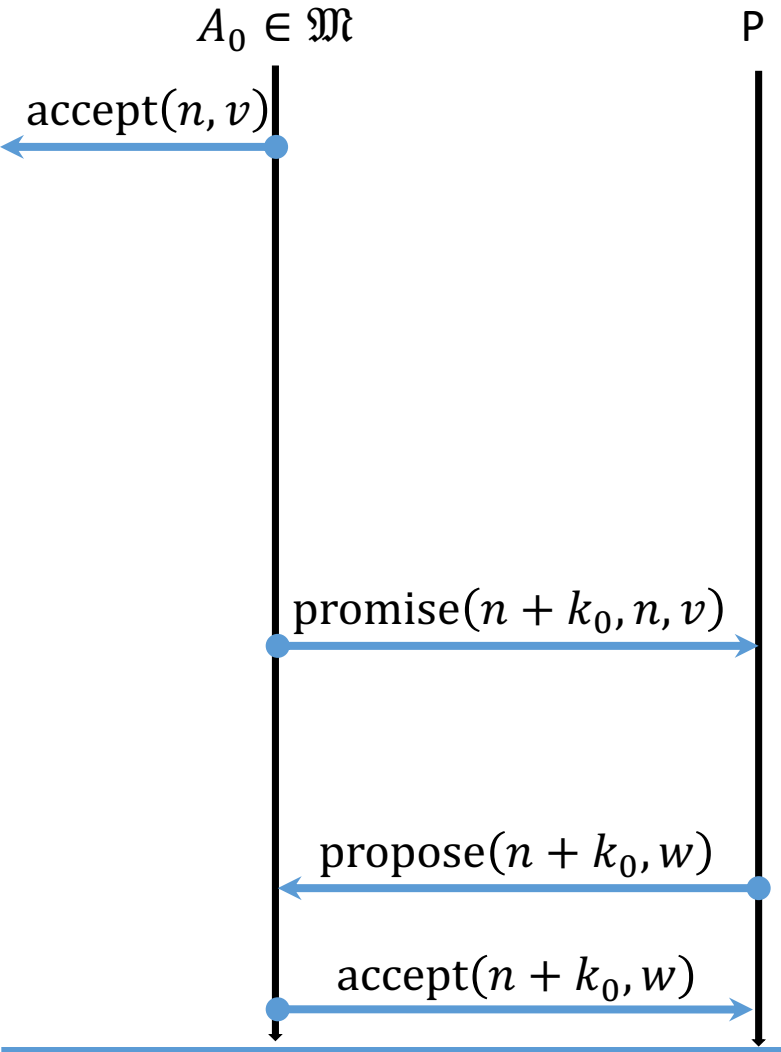
Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.



1. Сообщение accept$(n + k_0, w)$ посылается только в ответ на propose$(n + k_0, w)$ от P.
2. P отправляет propose только тем acceptor'ам, которые ответили promise.

Корректность алгоритма

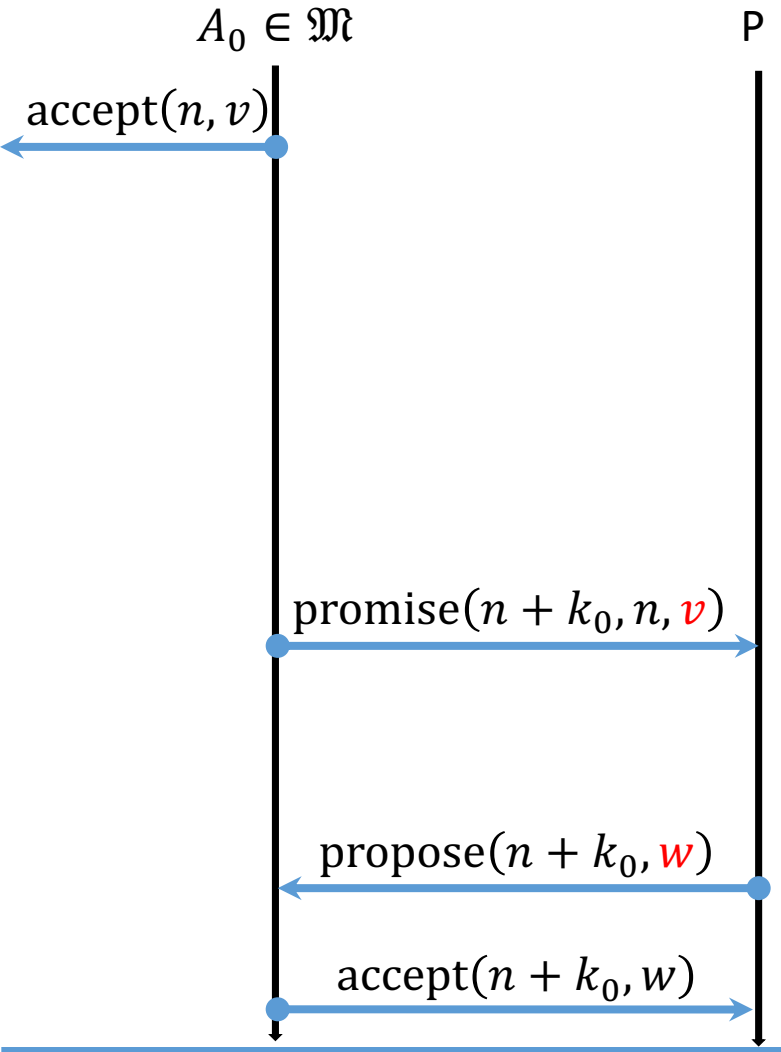
Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.



1. Сообщение accept$(n + k_0, w)$ посылается только в ответ на propose$(n + k_0, w)$ от P.
2. P отправляет propose только тем acceptor'ам, которые ответили promise.

Корректность алгоритма

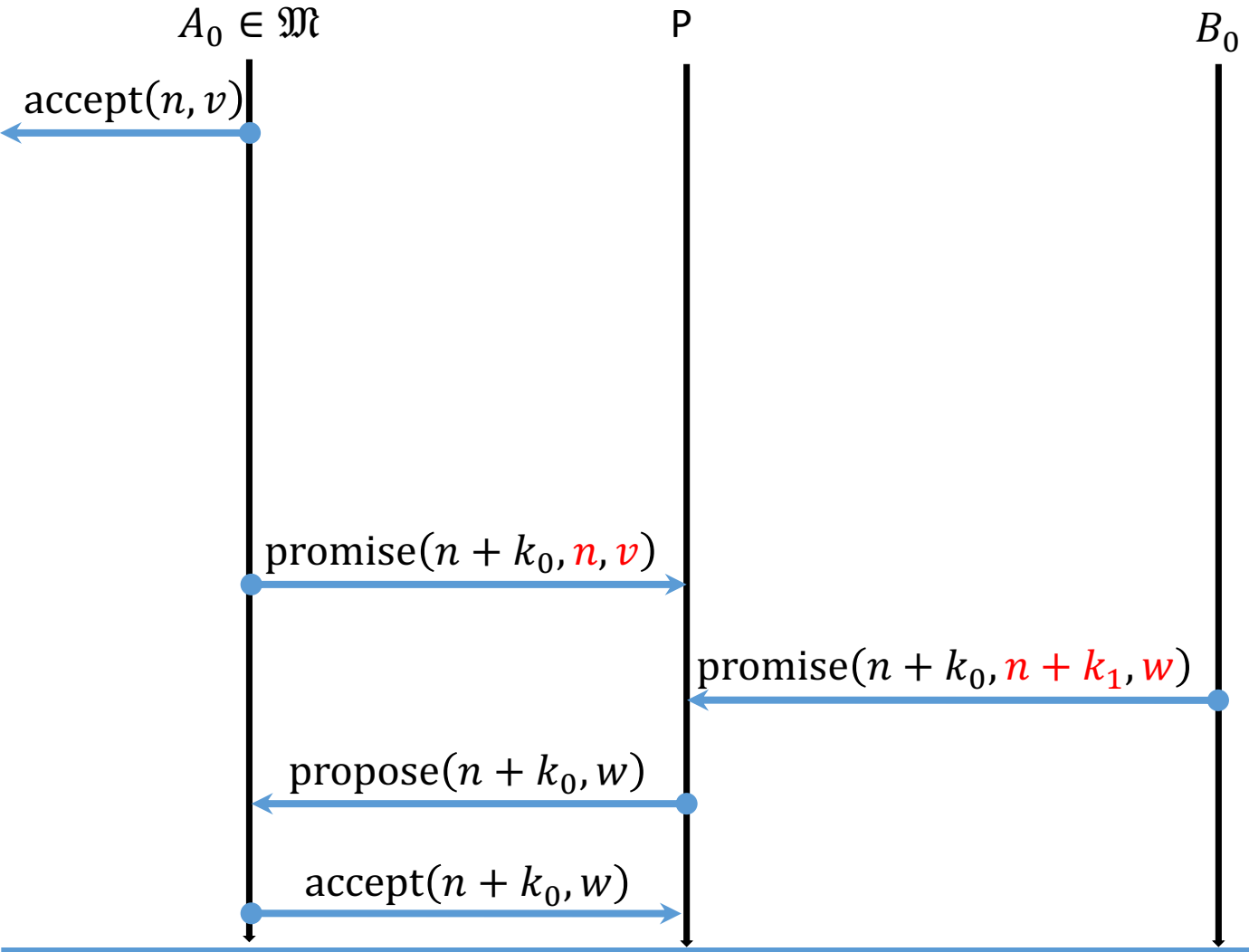
Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.



1. Сообщение accept$(n + k_0, w)$ посылается только в ответ на propose$(n + k_0, w)$ от P.
2. P отправляет propose только тем acceptor'ам, которые ответили promise.
3. При наличии ответа promise$(n + k_0, n, v)$, как P мог отправить propose$(n + k_0, w)$?

Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.

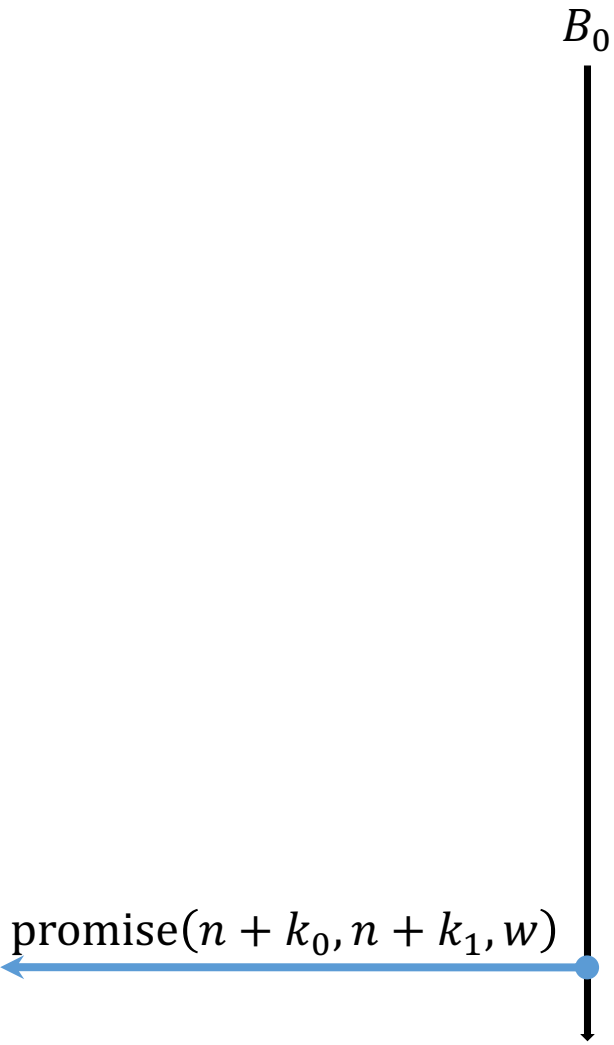


1. Сообщение accept$(n + k_0, w)$ посылается только в ответ на propose$(n + k_0, w)$ от P.
2. P отправляет propose только тем acceptor'ам, которые ответили promise.
3. При наличии ответа promise$(n + k_0, n, v)$, как P мог отправить propose$(n + k_0, w)$?

   Только при условии, что он ещё получил promise$(n + k_0, n + k_1, w)$, где $n < n + k_1 < n + k_0$.

Корректность алгоритма

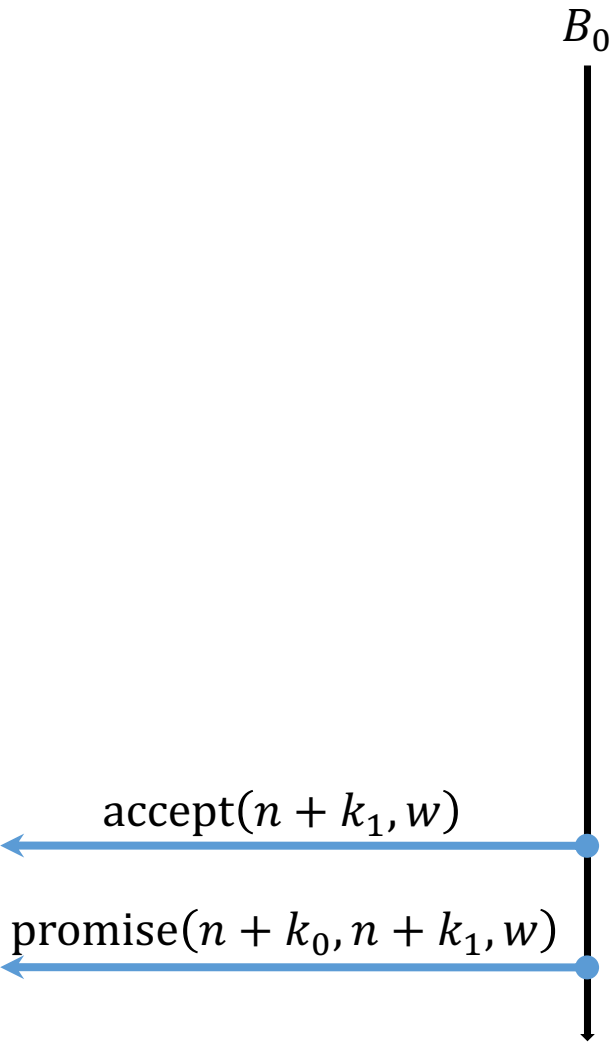
Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.



Посмотрим на B_0 внимательнее.

Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.

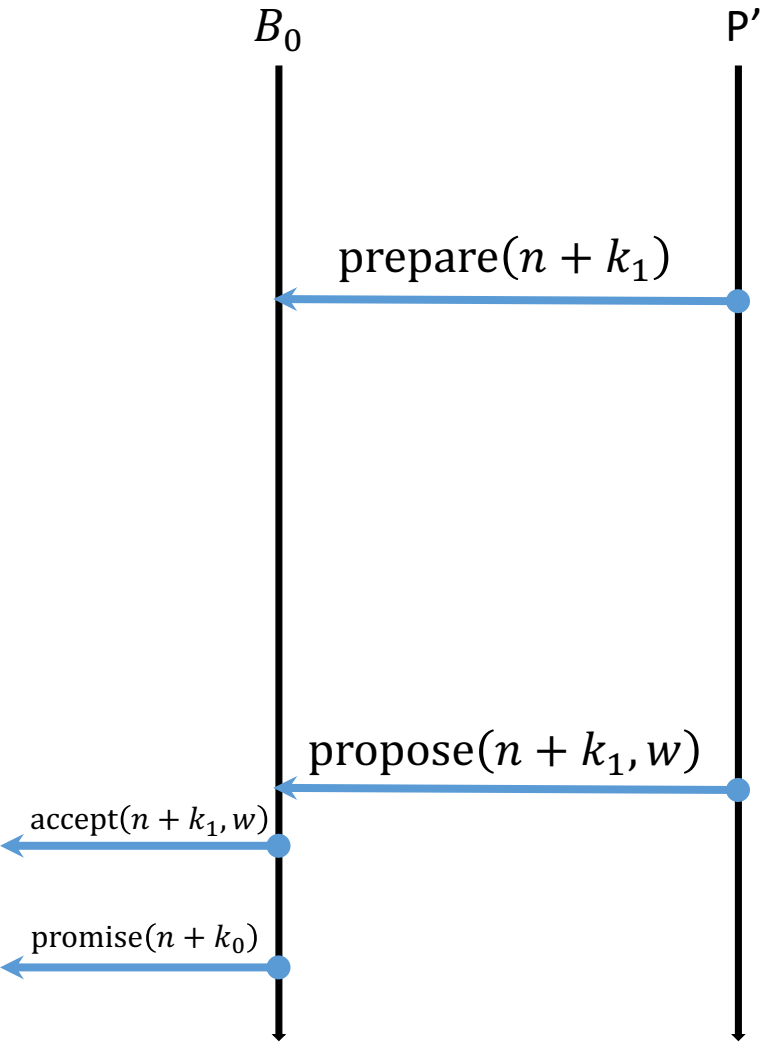


Посмотрим на B_0 внимательнее.

1. B_0 отправит $\text{promise}(n + k_0, n + k_1, w)$ только после $\text{accept}(n + k_1, w)$.

Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.

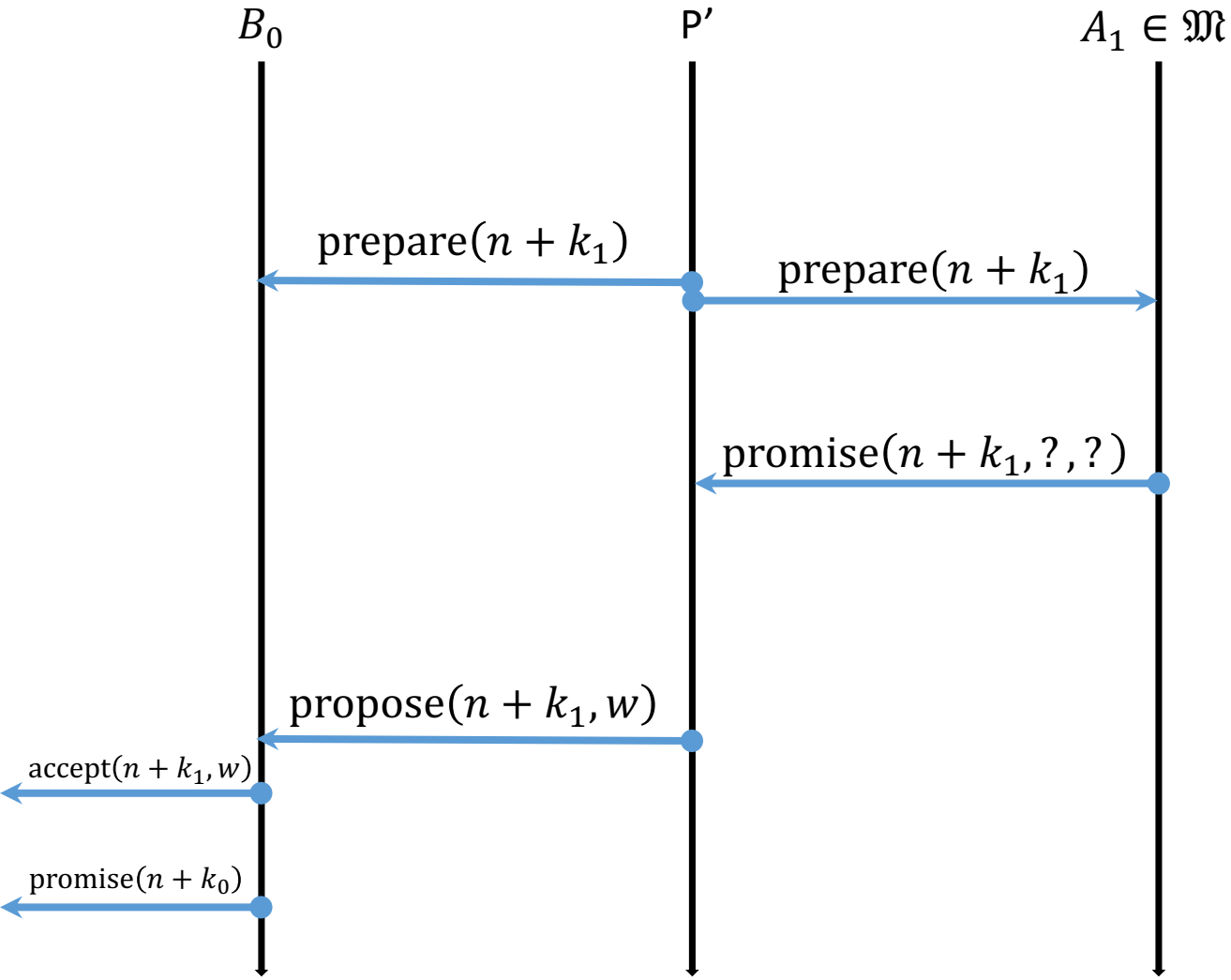


Посмотрим на B_0 внимательнее.

1. B_0 отправит $\text{promise}(n + k_0, n + k_1, w)$ только после $\text{accept}(n + k_1, w)$.
2. Чтобы B_0 принял $(n + k_1, w)$, надо, чтобы его кто-то предложил.

Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.

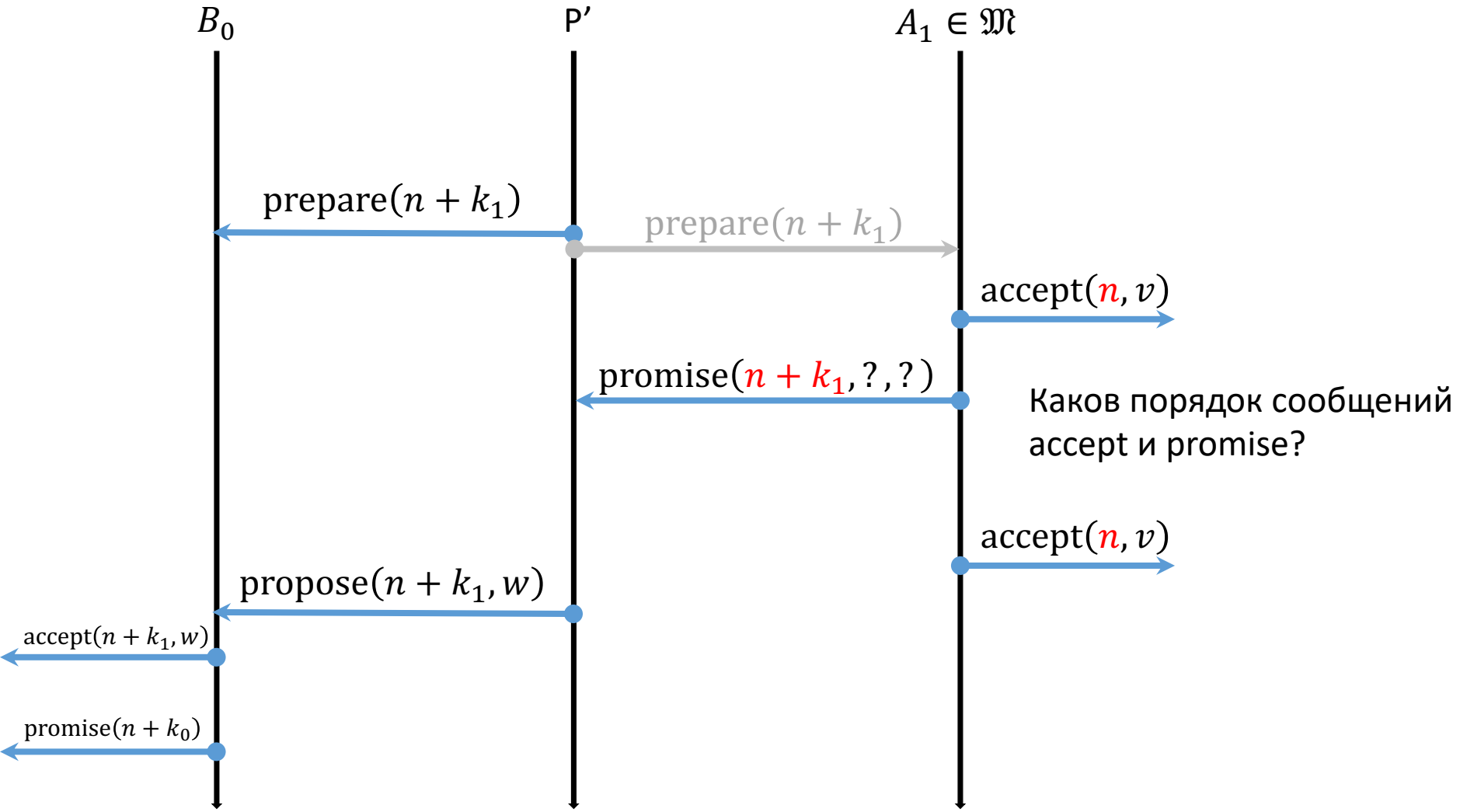


Посмотрим на B_0 внимательнее.

1. B_0 отправит $\text{promise}(n + k_0, n + k_1, w)$ только после $\text{accept}(n + k_1, w)$.
2. Чтобы B_0 принял $(n + k_1, w)$, надо, чтобы его кто-то предложил.
3. P' предложит $(n + k_1, w)$ только после того, как получит $\text{promise}(n + k_1)$ от большинства acceptor'ов, в частности, от одного из acceptor'ов из $\mathfrak{M}$.

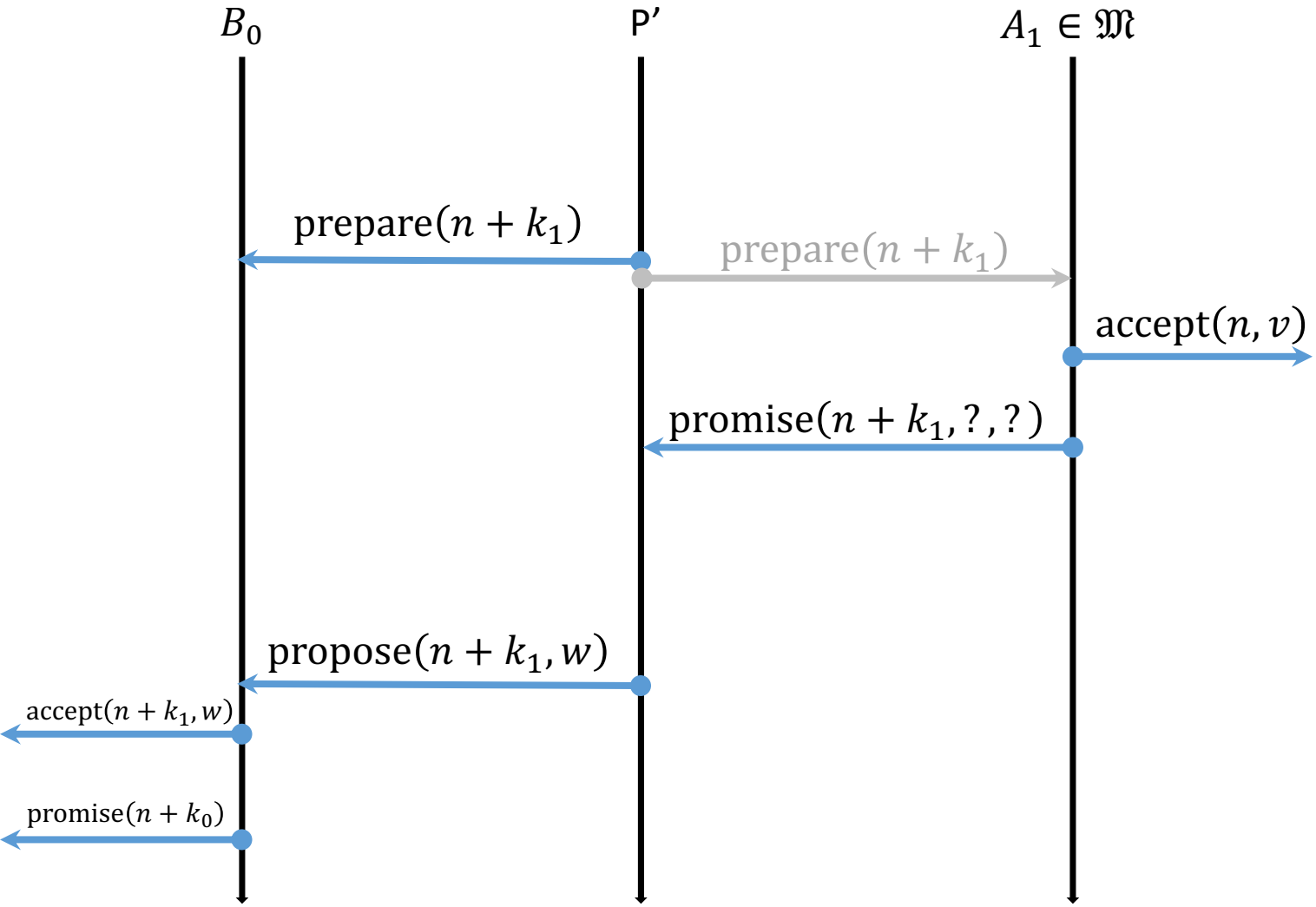
Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.



Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.

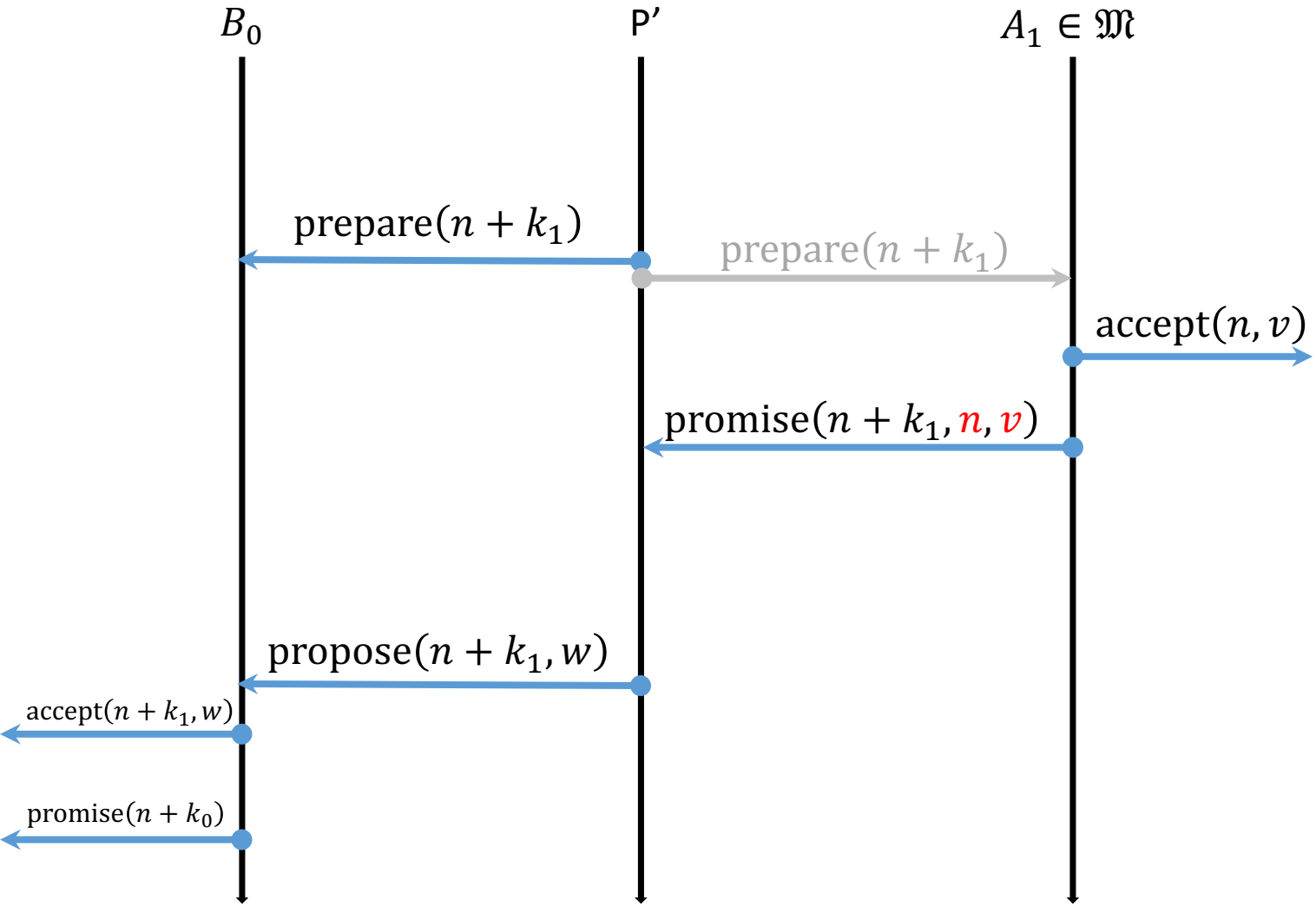


Посмотрим на B_0 внимательнее.

1. B_0 отправит $\text{promise}(n + k_0, n + k_1, w)$ только после $\text{accept}(n + k_1, w)$.
2. Чтобы B_0 принял $(n + k_1, w)$, надо, чтобы его кто-то предложил.
3. P' предложит $(n + k_1, w)$ только после того, как получит $\text{promise}(n + k_1)$ от большинства acceptor'ов, в частности, от одного из acceptor'ов из $\mathfrak{M}$.

Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.

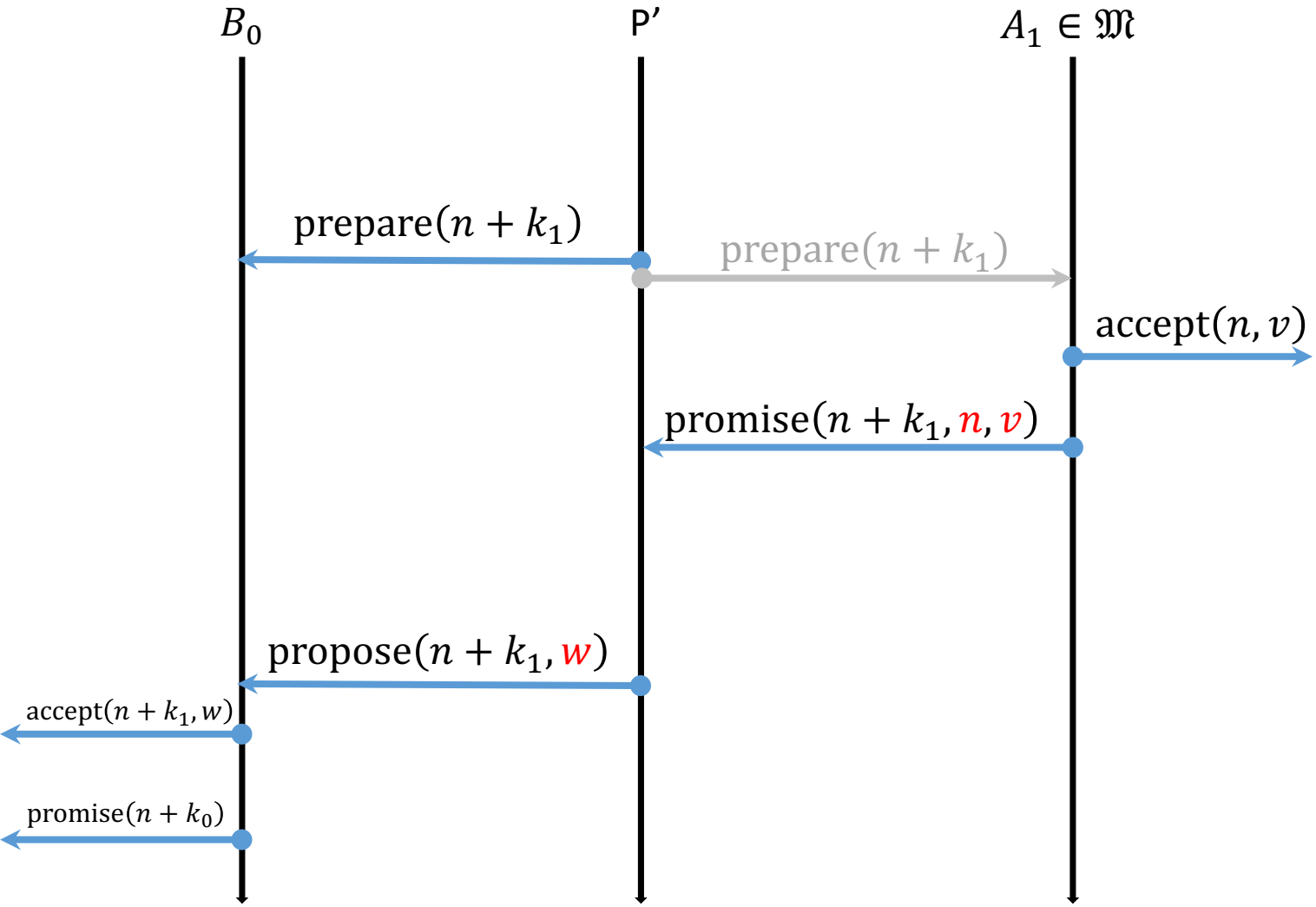


Посмотрим на B_0 внимательнее.

1. B_0 отправит $\text{promise}(n + k_0, n + k_1, w)$ только после $\text{accept}(n + k_1, w)$.
2. Чтобы B_0 принял $(n + k_1, w)$, надо, чтобы его кто-то предложил.
3. P' предложит $(n + k_1, w)$ только после того, как получит $\text{promise}(n + k_1)$ от большинства acceptor'ов, в частности, от одного из acceptor'ов из $\mathfrak{M}$.
4. A_1 ответил $\text{promise}(n + k_1, n, v)$.

Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.

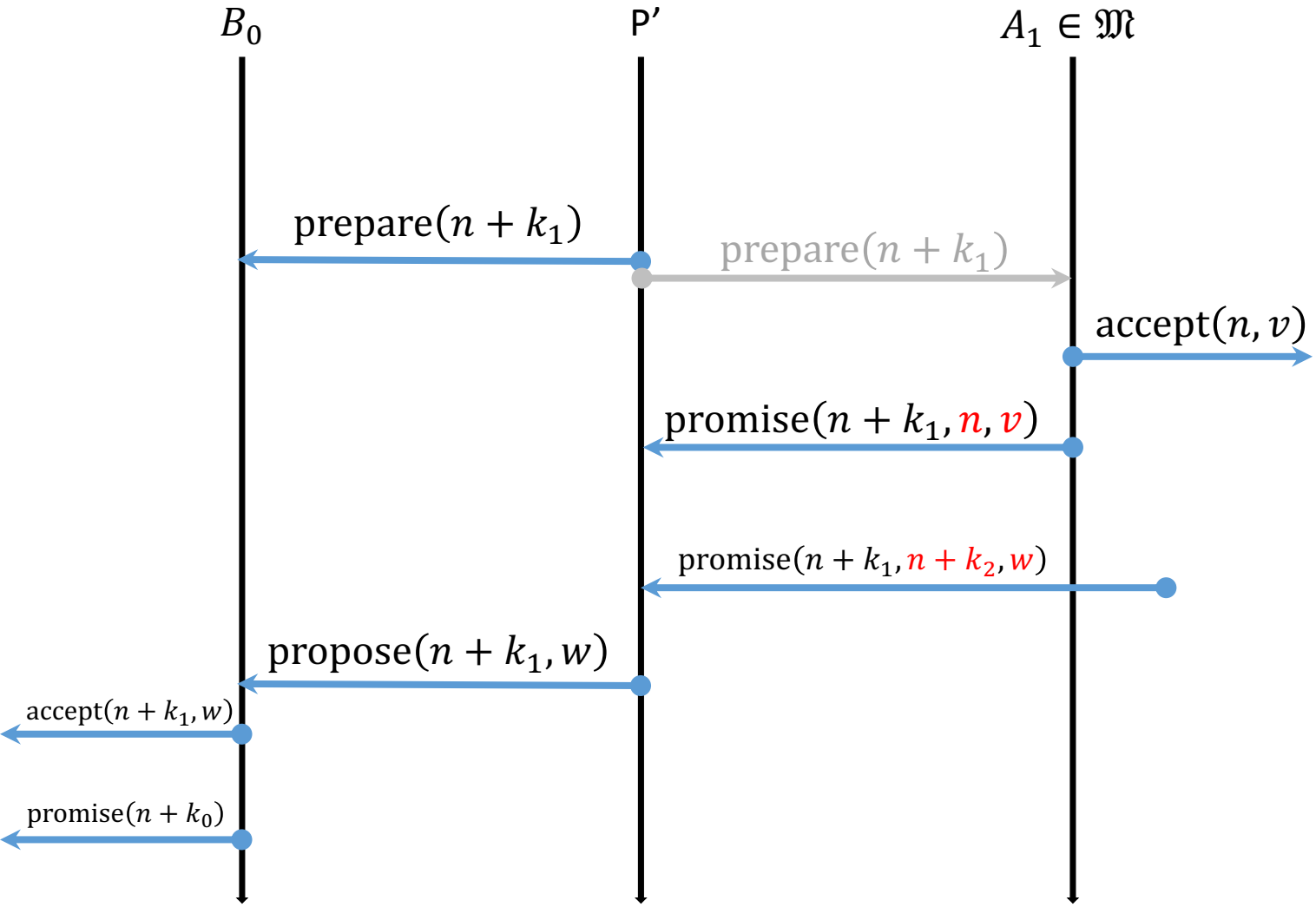


Посмотрим на B_0 внимательнее.

1. B_0 отправит $\text{promise}(n + k_0, n + k_1, w)$ только после $\text{accept}(n + k_1, w)$.
2. Чтобы B_0 принял $(n + k_1, w)$, надо, чтобы его кто-то предложил.
3. P' предложит $(n + k_1, w)$ только после того, как получит $\text{promise}(n + k_1)$ от большинства acceptor'ов, в частности, от одного из acceptor'ов из $\mathfrak{M}$.
4. A_1 ответил $\text{promise}(n + k_1, n, v)$.
5. При наличии ответа $\text{promise}(n + k_0, n, v)$, как P' мог отправить $\text{propose}(n + k_1, w)$?

Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.



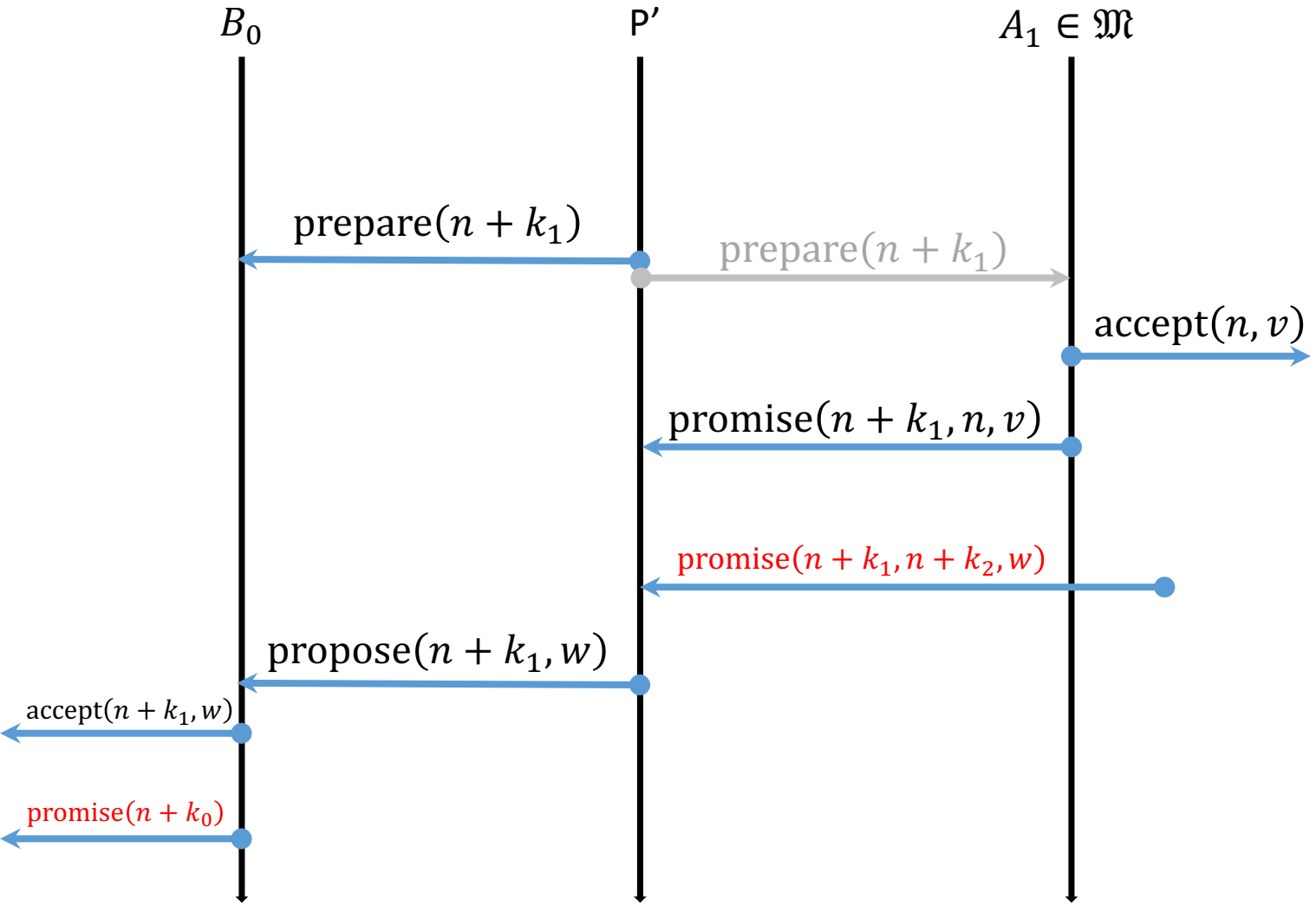
Посмотрим на B_0 внимательнее.

1. B_0 отправит $\text{promise}(n + k_0, n + k_1, w)$ только после $\text{accept}(n + k_1, w)$.
2. Чтобы B_0 принял $(n + k_1, w)$, надо, чтобы его кто-то предложил.
3. P' предложит $(n + k_1, w)$ только после того, как получит $\text{promise}(n + k_1)$ от большинства acceptor'ов, в частности, от одного из acceptor'ов из $\mathfrak{M}$.
4. A_1 ответил $\text{promise}(n + k_1, n, v)$.
5. При наличии ответа $\text{promise}(n + k_0, n, v)$, как P' мог отправить $\text{propose}(n + k_1, w)$?

   Только при условии, что он ещё получил $\text{promise}(n + k_1, n + k_2, w)$, где $n < n + k_2 < n + k_1$.

Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.



Существует acceptor, который послал ответы $\text{accept}(n, v)$ и $\text{accept}(n + k_0, w)$.

$\Downarrow$

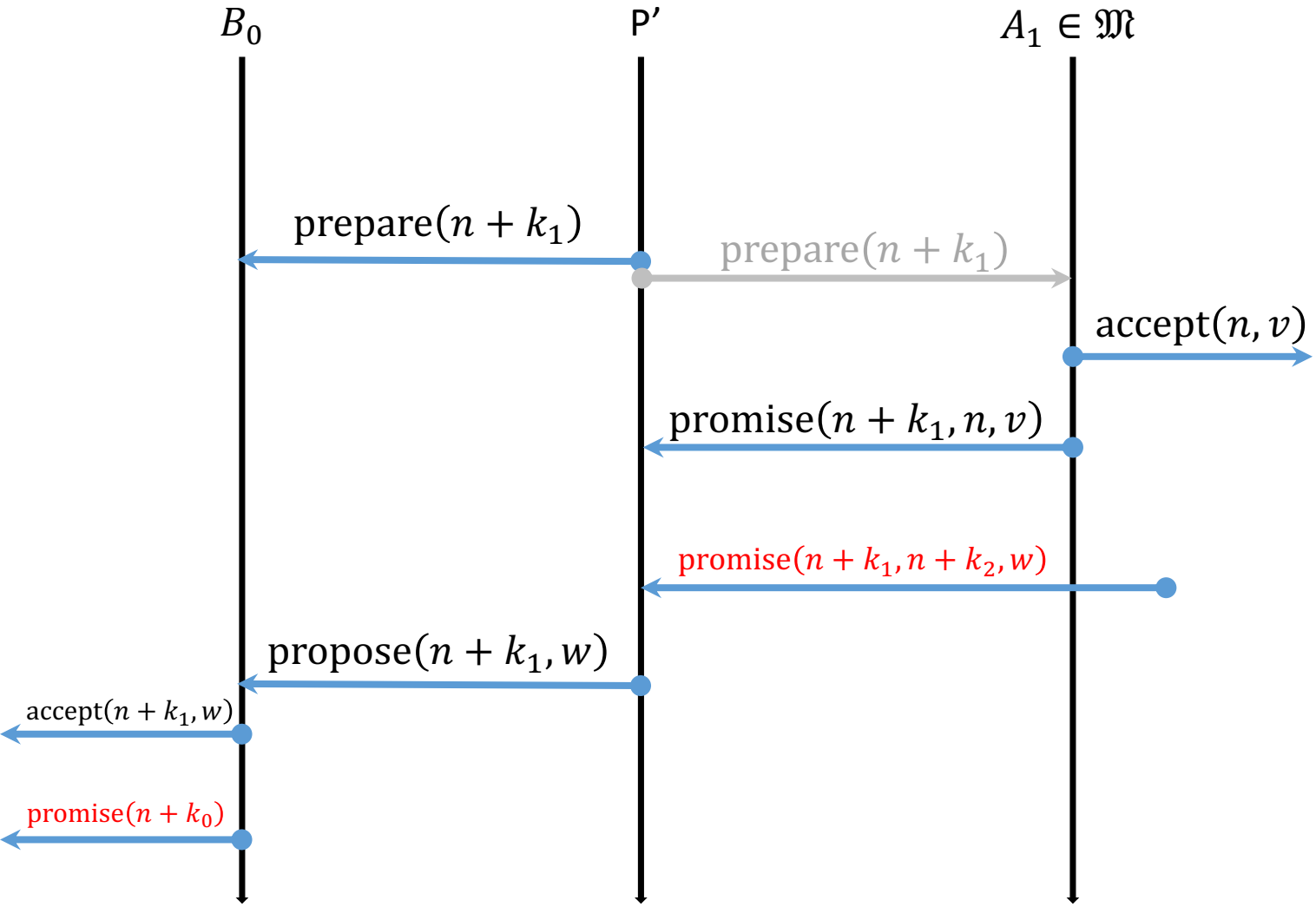
Существует acceptor, который послал $\text{promise}(n + k_0, n + k_1, w)$, где $n < n + k_1 < n + k_0$.

$\Downarrow$

Существует acceptor, который послал $\text{promise}(n + k_1, n + k_2, w)$, где $n < n + k_2 < n + k_1 < n + k_0$.

Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.



Существует acceptor, который послал ответы $\text{accept}(n, v)$ и $\text{accept}(n + k_0, w)$.

$\Downarrow$

Существует acceptor, который послал $\text{promise}(n + k_0, n + k_1, w)$, где $n < n + k_1 < n + k_0$.

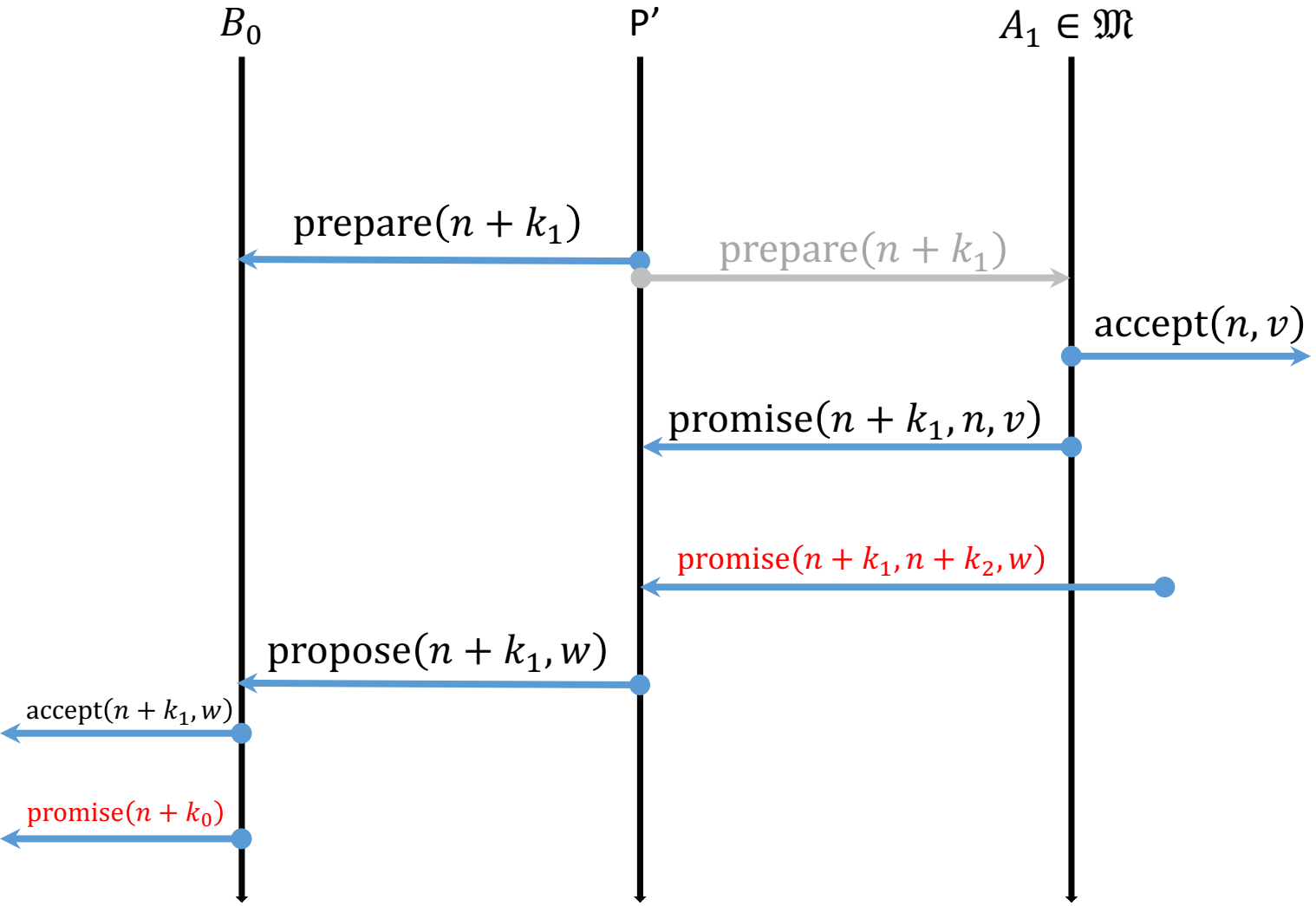
$\Downarrow$

Существует acceptor, который послал $\text{promise}(n + k_1, n + k_2, w)$, где $n < n + k_2 < n + k_1 < n + k_0$.

Продолжим процесс и получим противоречие, поскольку количество эпох между n и $n + k_0$ конечно.

Корректность алгоритма

Казалось бы, мы разрешаем ситуацию, когда большинство $\mathfrak{M}$ acceptor'ов приняли значение (n, v), а потом некоторые из них приняли значение $(n + k, w)$, где $k > 0$ и $w \neq v$.



Существует acceptor, который послал ответы $\text{accept}(n, v)$ и $\text{accept}(n + k_0, w)$.

$\Downarrow$

Существует acceptor, который послал $\text{promise}(n + k_0, n + k_1, w)$, где $n < n + k_1 < n + k_0$.

$\Downarrow$

Существует acceptor, который послал $\text{promise}(n + k_1, n + k_2, w)$, где $n < n + k_2 < n + k_1 < n + k_0$.

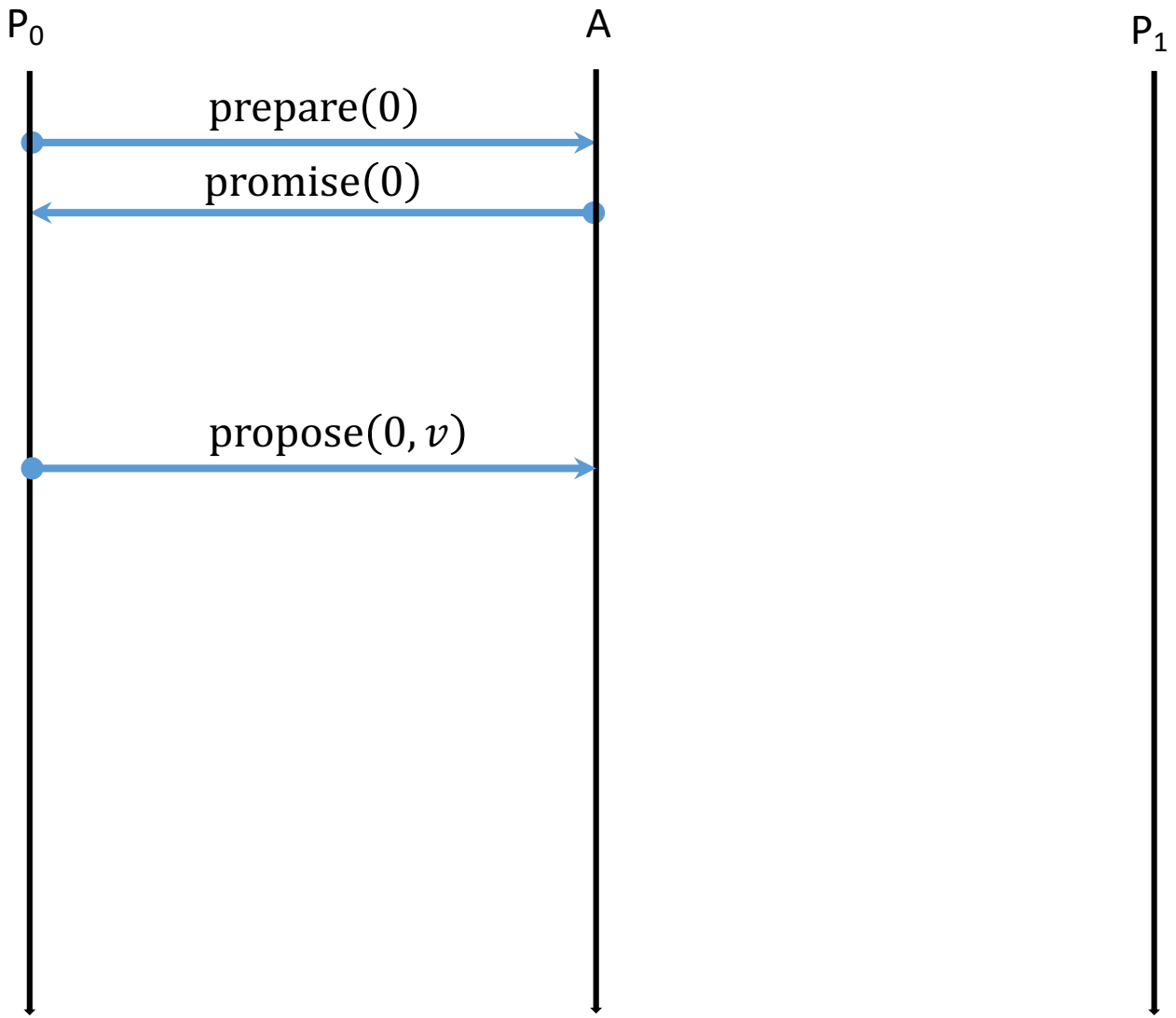
Продолжим процесс и получим противоречие, поскольку количество эпох между n и $n + k_0$ конечно.

$\Downarrow$

Если acceptor ответил $\text{accept}(n, v)$, он уже не может ответить $\text{accept}(n + k_0, w)$.

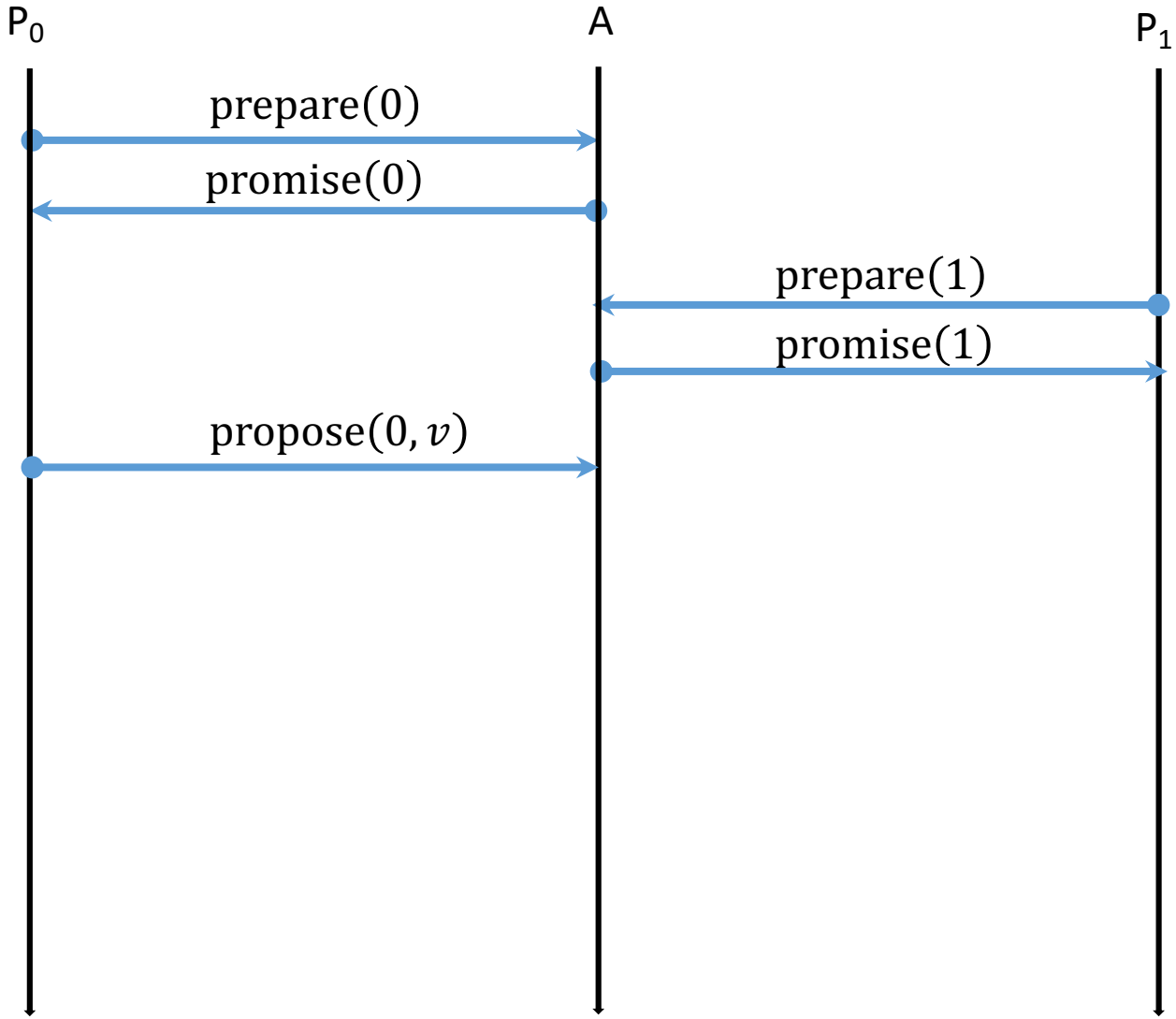
Liveness

PAXOS гарантирует единственность выбранного значения. Гарантируется ли существование выбора?



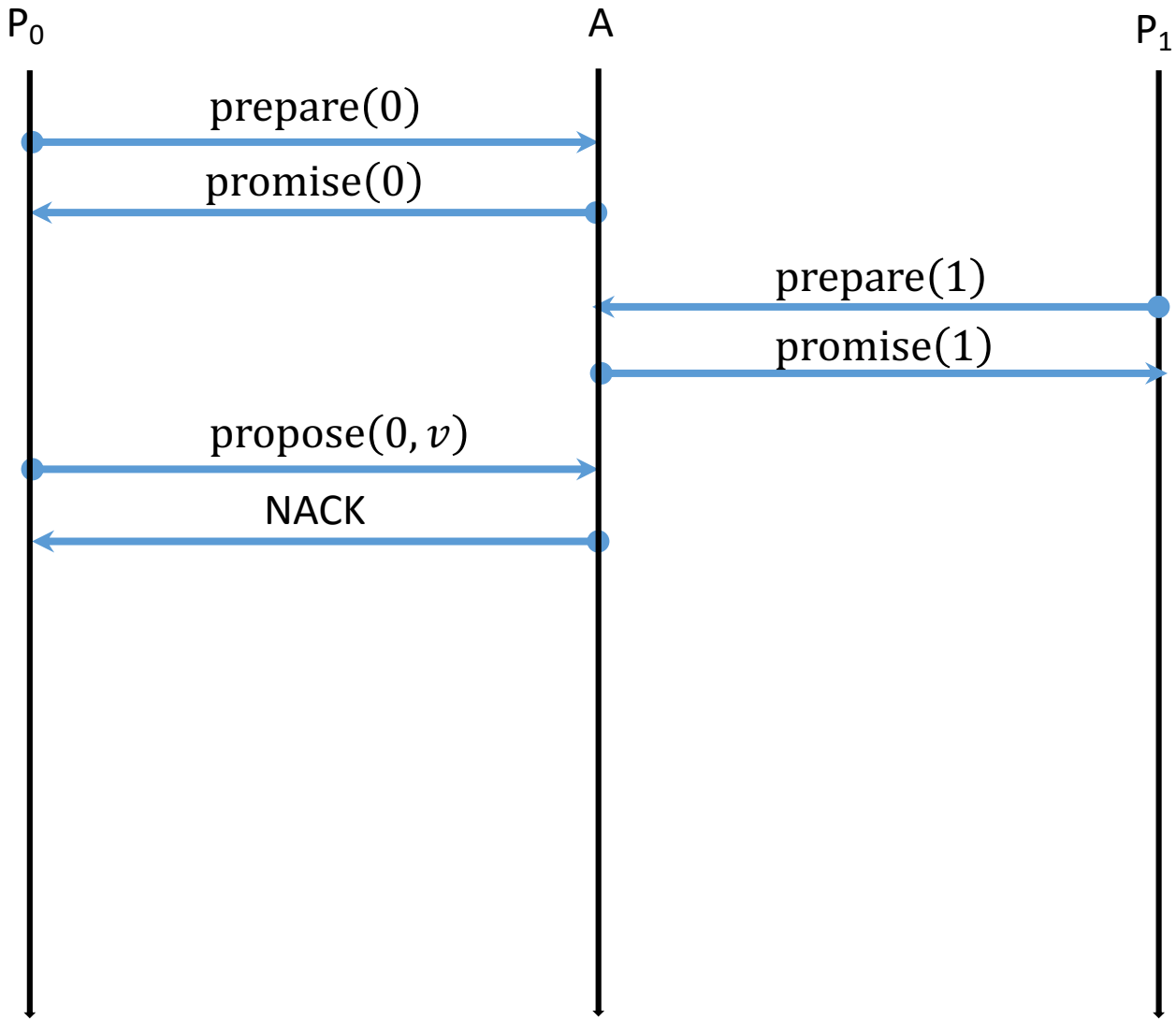
Liveness

PAXOS гарантирует единственность выбранного значения. Гарантируется ли существование выбора?



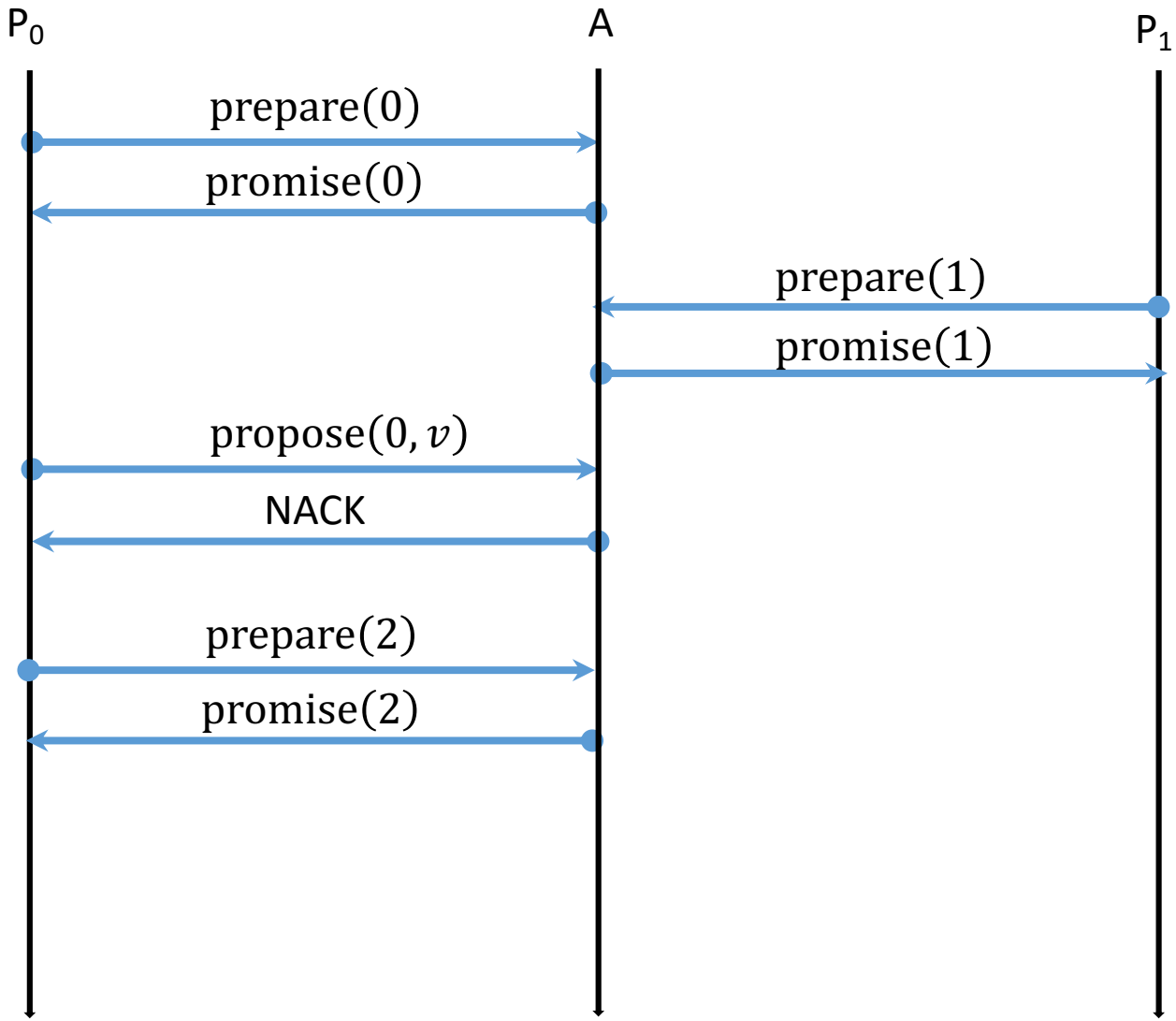
Liveness

PAXOS гарантирует единственность выбранного значения. Гарантируется ли существование выбора?



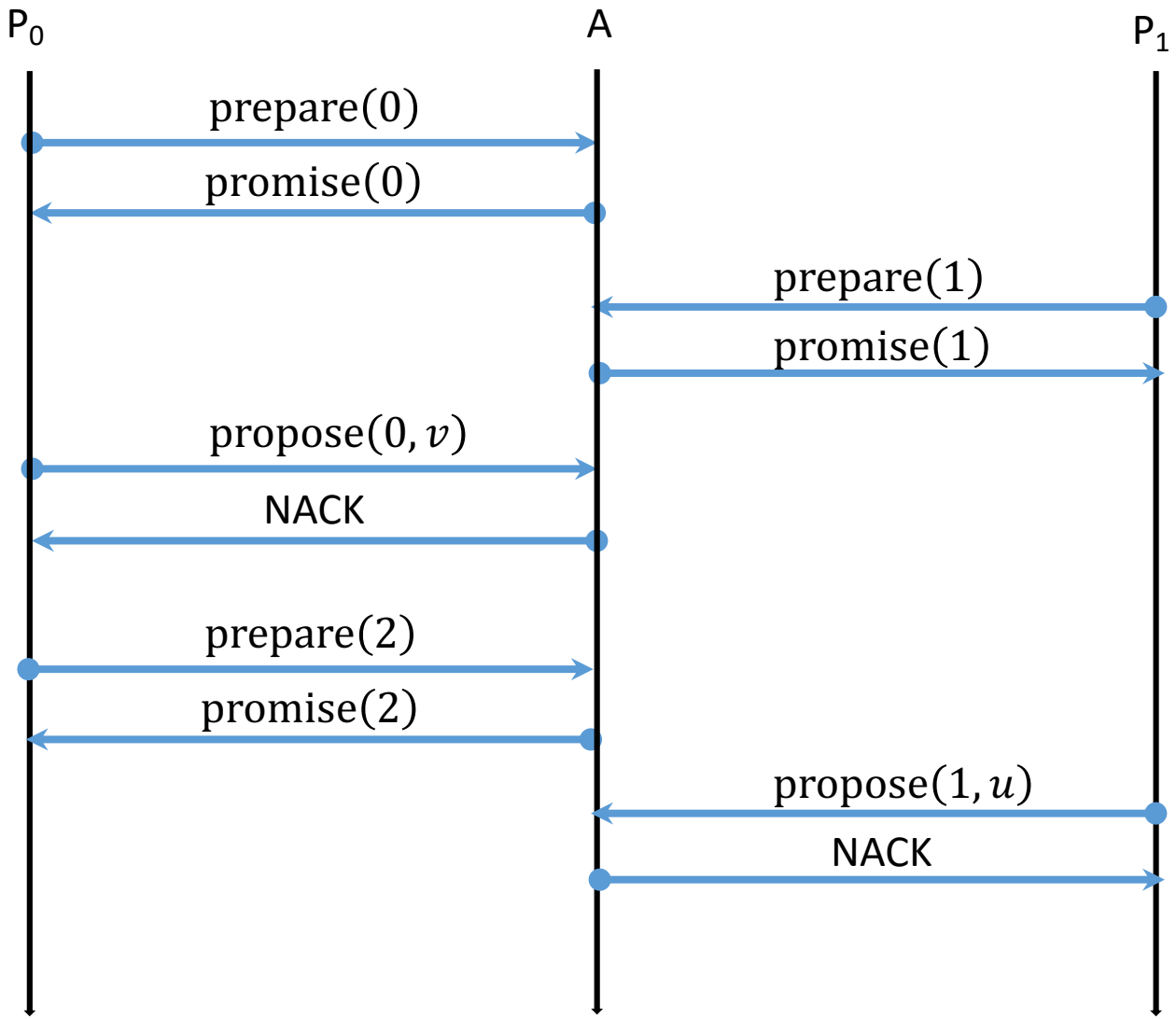
Liveness

PAXOS гарантирует единственность выбранного значения. Гарантируется ли существование выбора?



Liveness

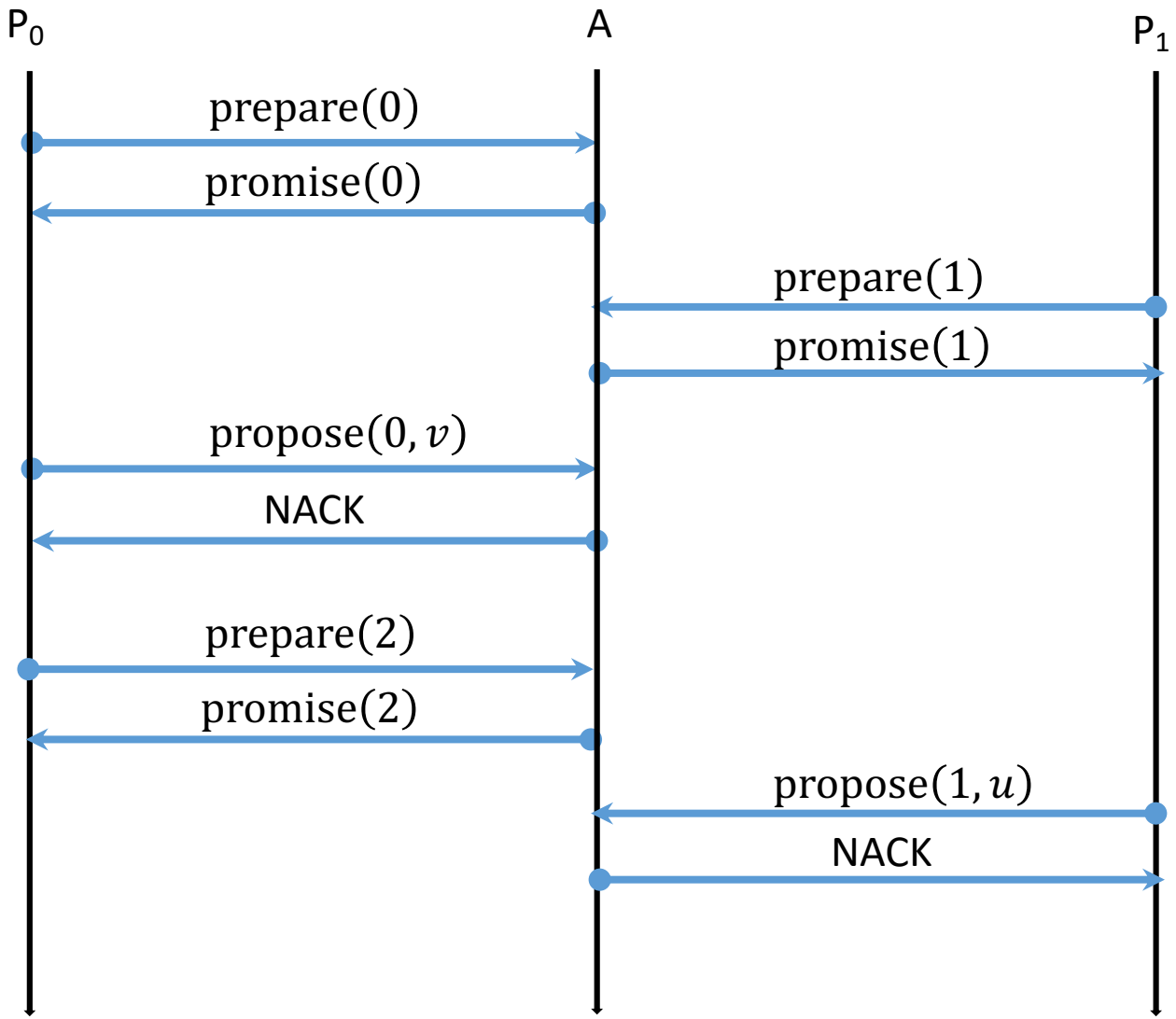
PAXOS гарантирует единственность выбранного значения. Гарантируется ли существование выбора?



P0 и P1 могут мешать друг другу неограниченно долго.

Liveness

PAXOS гарантирует единственность выбранного значения. Гарантируется ли существование выбора?

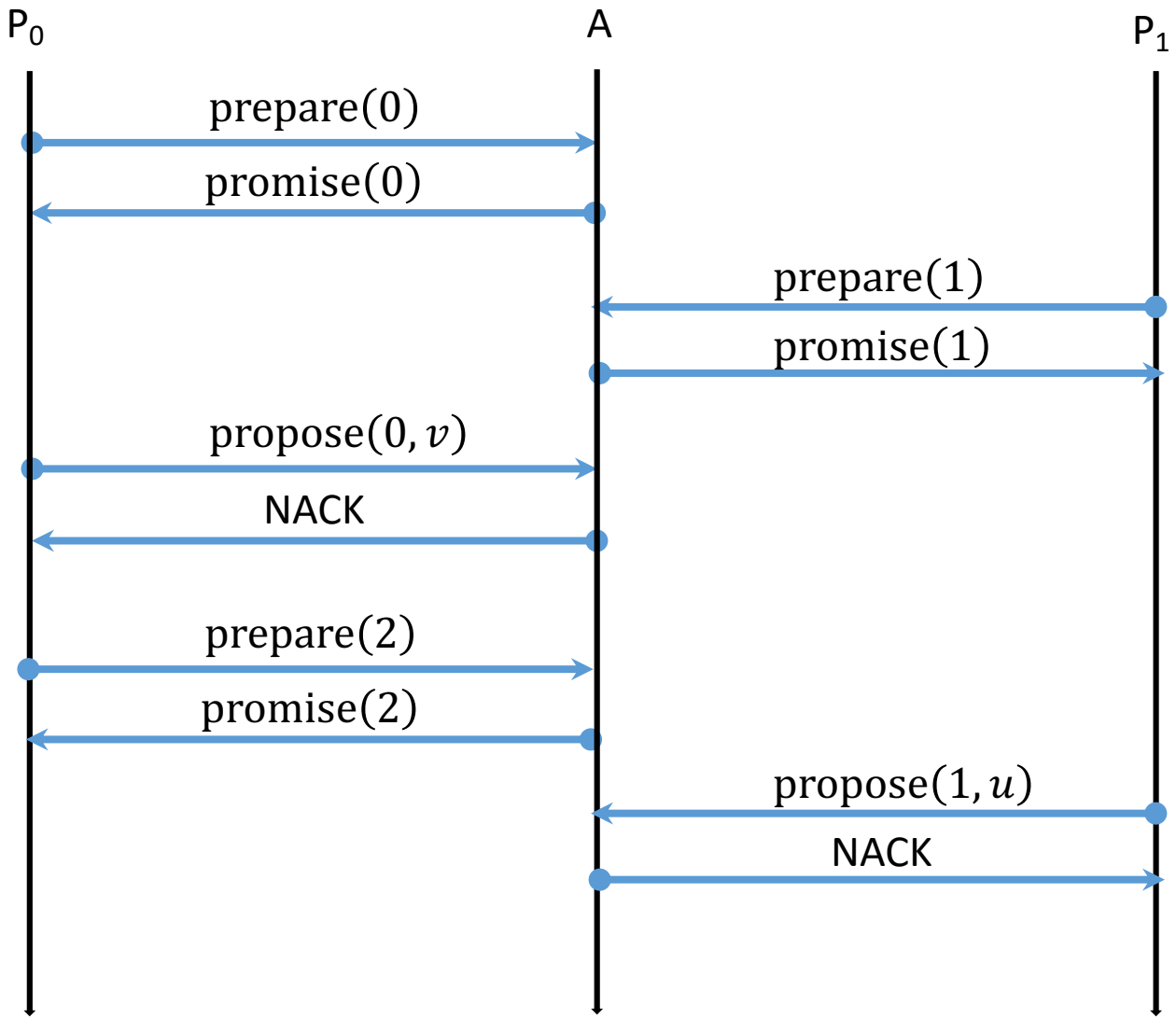


P0 и P1 могут мешать друг другу неограниченно долго.

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

Liveness

PAXOS гарантирует единственность выбранного значения. Гарантируется ли существование выбора?



P0 и P1 могут мешать друг другу неограниченно долго.

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

На практике проблема хорошо решается выбором выделенного proposer'а.

При выборе выделенного proposer'а хватает рандомизации моментов отправки предложений о кандидатах.

Напоминание: обработка повреждений ФС в различных приложениях

В работе [1] данные следующих приложений располагали на ФС, случайно подменяющей содержимое блоков:

- 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

Напоминание: обработка повреждений ФС в различных приложениях

В работе [1] данные следующих приложений располагали на ФС, случайно подменяющей содержимое блоков:

- Redis,
  - Не проверяет контрольные суммы пользовательских данных,
  - Как следствие, реплицирует некорректные данные между узлами,
  - Повреждения в ФС обрабатываются с помощью assert().
- ZooKeeper,
- Cassandra,
  - Забыли о контрольных суммах для несжатых данных,
  - При несоответствии данных и контрольной суммы выбирает последнюю запись как правильную, поэтому может распространять повреждения между репликами,
- 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

Напоминание: обработка повреждений ФС в различных приложениях

В работе [1] данные следующих приложений располагали на ФС, случайно подменяющей содержимое блоков:

- Redis,
  - Не проверяет контрольные суммы пользовательских данных,
  - Как следствие, реплицирует некорректные данные между узлами,
  - Повреждения в ФС обрабатываются с помощью assert().
- ZooKeeper,
- Cassandra,
  - Забыли о контрольных суммах для несжатых данных,
  - При несоответствии данных и контрольной суммы выбирает последнюю запись как правильную, поэтому может распространять повреждения между репликами,
- Kafka,
- RethinkDB,
- LogCabin.

Примеры Redis и Cassandra показывают важность модели сбоев, от которых защищает алгоритм консенсуса.
PAXOS предполагает fail-stop сбои участников и сеть, которая не меняет содержимое сообщений.

[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. Участники кластера обязаны следовать PAXOS даже при наличии изменённого/повреждённого содержимого дисков и памяти.

PAXOS made live

1. Участники кластера обязаны следовать PAXOS даже при наличии изменённого/повреждённого содержимого дисков и памяти.
2. Master leases: Основная нагрузка на распределённые key-value хранилища – это обычно запросы на чтение. Их хочется обслуживать одним узлом, не договариваясь всем кластером «на i-е чтение ответом будет x».

PAXOS made live

1. Участники кластера обязаны следовать PAXOS даже при наличии изменённого/повреждённого содержимого дисков и памяти.
2. Master leases: Основная нагрузка на распределённые key-value хранилища – это обычно запросы на чтение. Их хочется обслуживать одним узлом, не договариваясь всем кластером «на i-е чтение ответом будет x».
3. Добавление/удаление узлов кластера.

PAXOS made live

1. Участники кластера обязаны следовать PAXOS даже при наличии изменённого/повреждённого содержимого дисков и памяти.
2. Master leases: Основная нагрузка на распределённые key-value хранилища – это обычно запросы на чтение. Их хочется обслуживать одним узлом, не договариваясь всем кластером «на i-е чтение ответом будет x».
3. Добавление/удаление узлов кластера.
4. Снимки состояния системы: журнал изменений растёт неограниченно и весь его хранить нельзя. Периодически надо делать снимки состояния системы и отрезать начало журнала.

Список литературы

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