Основы построения файловых систем

Консенсус в распределённой системе

Сегодня мы обсудим, как сделать надёжный распределённый конечный автомат.

Напоминание

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.
2. При получении запроса $\mathrm{propose}(k, w)$, где k больше запомненного номера эпохи, принять значение (k, w) и ответить $\mathrm{accept}(k, w)$.

Напоминание

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.
2. При получении запроса $\mathrm{propose}(k, w)$, где k больше запомненного номера эпохи, принять значение (k, w) и ответить $\mathrm{accept}(k, w)$.

Оптимизация: acceptor при получении запроса $\mathrm{propose}(k, w)$, где номер эпохи k устаревший, может ответить NACK (Negative ACKnowledgement) proposer'у. PAXOS разрешает проигнорировать такой запрос, но NACK позволит proposer'у быстрее узнать, что другой участник разослал promise с более новой эпохой.

Будем считать, что acceptor отвечает NACK на propose с устаревшей эпохой.

Replicated FSM

Будем предполагать, что каждый процесс-участник PAXOS исполняет все три роли: proposer, acceptor и learner, притом один из процессов играет роль выделенного proposer’а, т.е. только он испускает предлагаемые значения.

Тогда N-я итерация рабочего цикла реплицированного журнала выглядит так:

1. подготовить новый номер эпохи и разослать prepare,
2. дождаться promise от остальных участников,
3. разослать propose со значением шага N,
4. дождаться ответов от других участников,
5. разослать другим участникам commit, чтобы надёжно сохранить длину журнала,
6. дождаться ответа на commit и ответить клиенту, инициировавшему шаг N.

Replicated FSM

Будем предполагать, что каждый процесс-участник PAXOS исполняет все три роли: proposer, acceptor и learner, притом один из процессов играет роль выделенного proposer'а, т.е. только он испускает предлагаемые значения.

Тогда N-я итерация рабочего цикла реплицированного журнала выглядит так:

1. подготовить новый номер эпохи и разослать prepare,
2. дождаться promise от остальных участников,
3. разослать propose со значением шага N,
4. дождаться ответов от других участников,
5. разослать другим участникам commit, чтобы надёжно сохранить длину журнала,
6. дождаться ответа на commit и ответить клиенту, инициировавшему шаг N.

Вопросы:

1. Что делать участникам, которые отстали (например, из-за того, что были отключены от сети)?
2. Сколько записей на диск надо сделать на каждой итерации?

Replicated FSM

Будем предполагать, что каждый процесс-участник PAXOS исполняет все три роли: proposer, acceptor и learner, притом один из процессов играет роль выделенного proposer'а, т.е. только он испускает предлагаемые значения.

Тогда N-я итерация рабочего цикла реплицированного журнала выглядит так:

1. подготовить новый номер эпохи и разослать prepare,
2. дождаться promise от остальных участников,
3. разослать propose со значением шага N,
4. дождаться ответов от других участников,
5. разослать другим участникам commit, чтобы надёжно сохранить длину журнала,
6. дождаться ответа на commit и ответить клиенту, инициировавшему шаг N.

Вопросы:

1. Что делать участникам, которые отстали (например, из-за того, что были отключены от сети)?
   - Каждый участник вместе с сообщением prepare/promise указывает номер последнего закоммиченного шага. По этим номерам процессы, которые обменялись сообщениями, могут понять, кто из них отстал, и запросить недостающую часть журнала до того, как обработать prepare/promise.
2. Сколько записей на диск надо сделать на каждой итерации?

Replicated FSM

Будем предполагать, что каждый процесс-участник PAXOS исполняет все три роли: proposer, acceptor и learner, притом один из процессов играет роль выделенного proposer'а, т.е. только он испускает предлагаемые значения.

Тогда N-я итерация рабочего цикла реплицированного журнала выглядит так:

1. подготовить новый номер эпохи и разослать prepare,
2. дождаться promise от остальных участников,
3. разослать propose со значением шага N,
4. дождаться ответов от других участников,
5. разослать другим участникам commit, чтобы надёжно сохранить длину журнала,
6. дождаться ответа на commit и ответить клиенту, инициировавшему шаг N.

Вопросы:

1. Что делать участникам, которые отстали (например, из-за того, что были отключены от сети)?
2. Сколько записей на диск надо сделать на каждой итерации?
   - proposer должен сохранить номер эпохи, чтобы не переиспользовать его,
   - acceptor'ы должны сохранить номер эпохи, чтобы отсекать более старые сообщения,
   - acceptor'ы должны сохранить предложенное им значение,
   - acceptor'ы должны сохранить длину журнала при commit'е.

   Всего получается **4 последовательных fsync**.

Multi-PAXOS

Будем предполагать, что каждый процесс-участник PAXOS исполняет все три роли: proposer, acceptor и learner, притом один из процессов играет роль выделенного proposer'а, т.е. только он испускает предлагаемые значения.

Тогда N-я итерация рабочего цикла реплицированного журнала выглядит так:

1. подготовить новый номер эпохи и разослать prepare,
2. дождаться promise от остальных участников,
3. разослать propose со значением записи N,
4. дождаться ответов от других участников,
5. разослать другим участникам commit, чтобы надёжно сохранить длину журнала,
6. дождаться ответа на commit и ответить клиенту, инициировавшему шаг N.

Наблюдение: proposer'у достаточно исполнить шаги 1 и 2 один раз, а потом в цикле повторять 3-6, пока он не получает NACK от других участников. Получение NACK означает, что другой участник решил назначить себя выделенным proposer'ом.

Действительно, такой цикл можно рассматривать как один раунд PAXOS, где выбирается значение «записи с номерами $[N_0, N_1]$», где N_1 – номер, до которого удавалось успешно исполнять шаги 3-6.

Теперь на каждую дозапись в журнал требуется два fsync вместо четырёх.

Multi-PAXOS

Будем предполагать, что каждый процесс-участник PAXOS исполняет все три роли: proposer, acceptor и learner, притом один из процессов играет роль выделенного proposer'а, т.е. только он испускает предлагаемые значения.

Тогда N-я итерация рабочего цикла реплицированного журнала выглядит так:

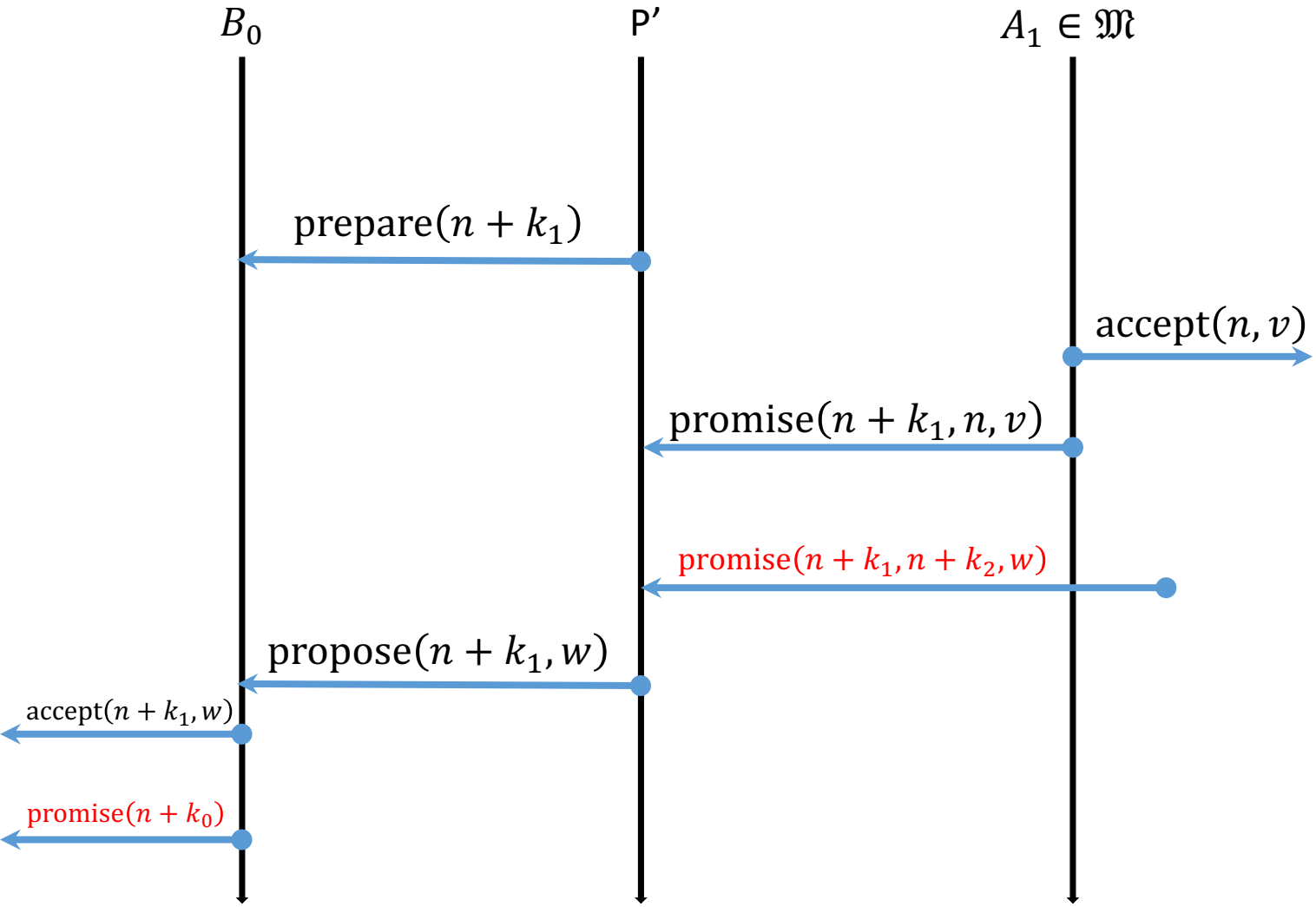
1. подготовить новый номер эпохи и разослать prepare,
2. дождаться promise от остальных участников,
3. разослать propose со значением записи N,
4. дождаться ответов от других участников,
5. разослать другим участникам commit, чтобы надёжно сохранить длину журнала,
6. дождаться ответа на commit и ответить клиенту, инициировавшему шаг N.

Наблюдение: количество fsync'ов можно дальше уменьшить, если за одну итерацию 3-6 рассылать не одну запись в журнал, а несколько. Это возможно, если имеется несколько клиентов, делающих независимые запросы.

Вопрос: как выбирать длину группы одновременно записываемых шагов (batch)? – Adaptive batching.

Корректность алгоритма

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

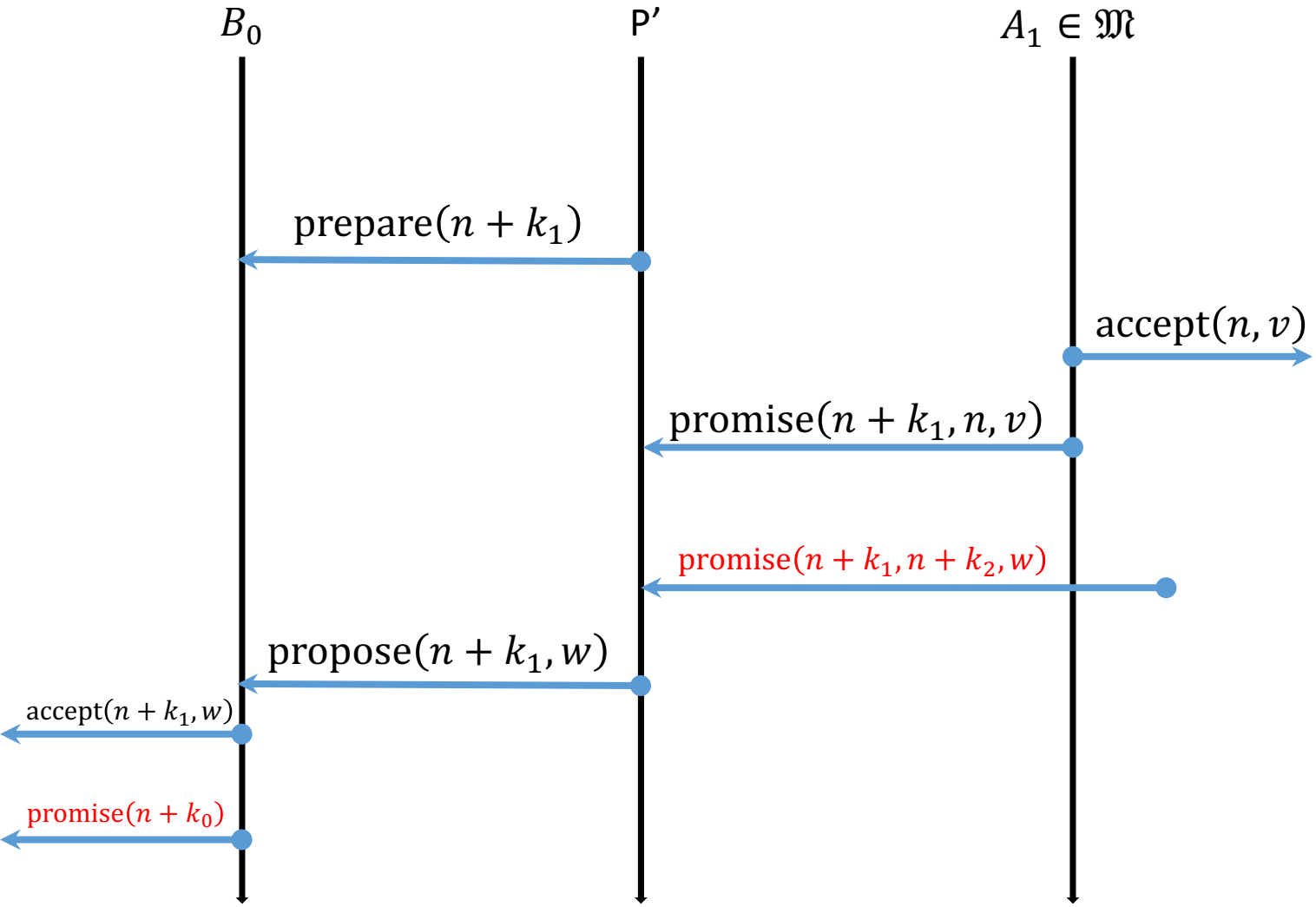


Рассуждение с прошлой лекции доказывает следующее утверждение:

Если в эпоху n было выбрано значение (n, v), то все запросы $\text{propose}(n + k, \cdot)$, где $k > 0$, предлагают значение v.

Корректность алгоритма

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



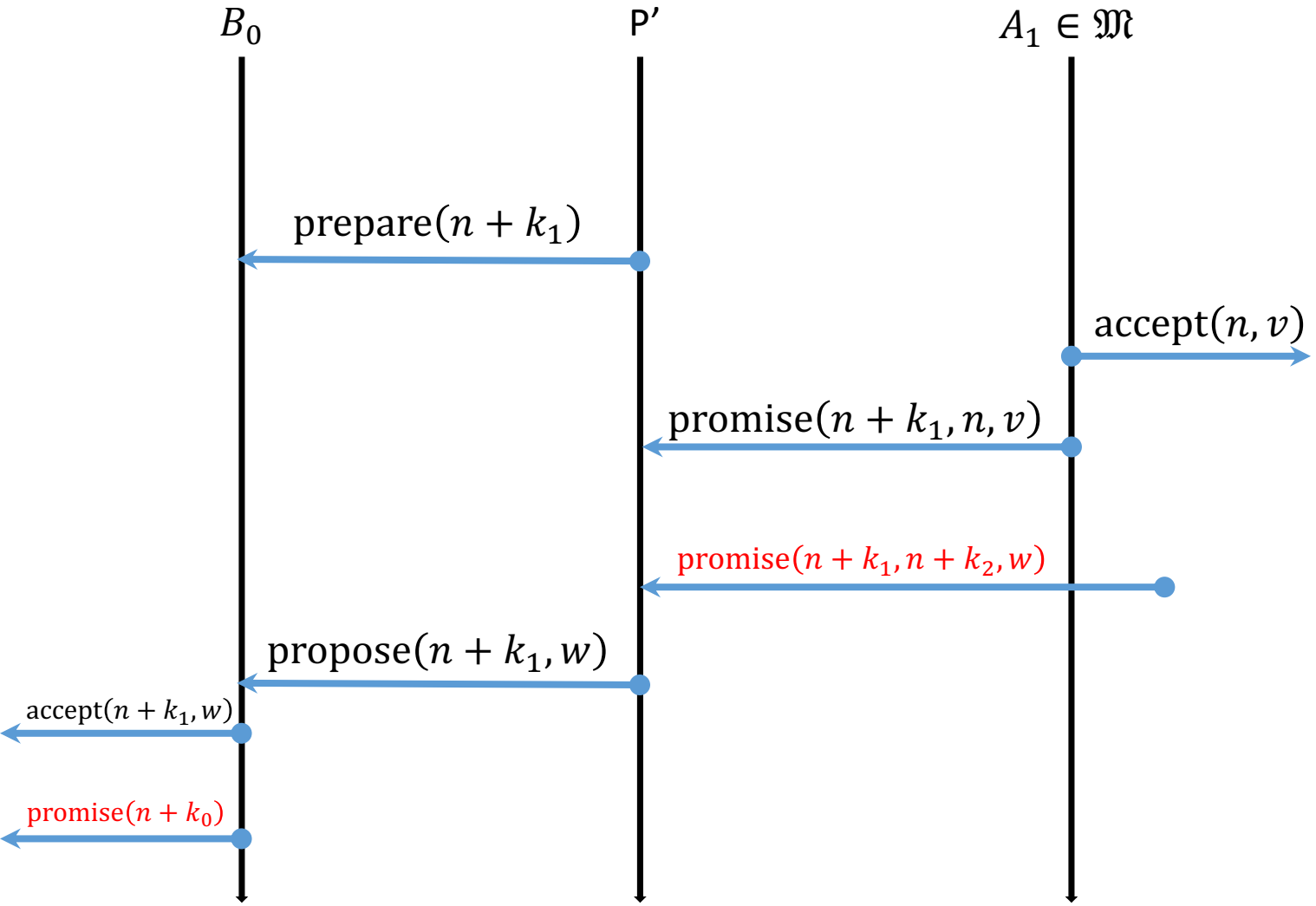
Рассуждение с прошлой лекции доказывает следующее утверждение:

Если в эпоху n было выбрано значение (n, v), то все запросы $\text{propose}(n + k, \cdot)$, где $k > 0$, предлагают значение v.

Действительно, запрос $\text{propose}(n + k, \cdot)$ рассылается большинству acceptor'ов, поэтому будет послан хотя бы одному acceptor'у из большинства, принявшего (n, v). В прошлый раз мы показали, что это невозможно.

Корректность алгоритма

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



Рассуждение с прошлой лекции доказывает следующее утверждение:

Если в эпоху n было выбрано значение (n, v), то все запросы $\text{propose}(n + k, \cdot)$, где $k > 0$, предлагают значение v.

Действительно, запрос $\text{propose}(n + k, w), w \neq v$ рассылается большинству acceptor'ов, поэтому будет послан хотя бы одному acceptor'у из большинства, принявшего (n, v). В прошлый раз мы показали, что это невозможно.

Обратно, из указанного свойства запросов propose следует, что PAXOS выбирает не более одного из предложенных значений.

Ослабление требований к кворумам

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.

Если в эпоху n было выбрано значение (n, v), то все запросы $\mathrm{propose}(n + k, \cdot)$, где $k > 0$, предлагают значение v.

Ослабление требований к кворумам

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.

Если в эпоху n было выбрано значение (n, v), то все запросы $\mathrm{propose}(n + k, \cdot)$, где $k > 0$, предлагают значение v.

Это свойство остаётся верным, если множество acceptor'ов, ответивших promise на шаге 3, пересекается с множеством acceptor'ов, выбравших значение (n, v). Множества acceptor'ов из пунктов 3 и 5 не обязаны быть большинствами – достаточно, чтобы они пересекались.

Ослабление требований к кворумам

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.

Если в эпоху n было выбрано значение (n, v), то все запросы $\mathrm{propose}(n + k, \cdot)$, где $k > 0$, предлагают значение v.

Это свойство остаётся верным, если множество acceptor'ов, ответивших promise на шаге 3, пересекается с множеством acceptor'ов, выбравших значение (n, v). Множества acceptor'ов из пунктов 3 и 5 не обязаны быть большинствами – достаточно, чтобы они пересекались.

Можно рассматривать кворумы для фазы 1 (prepare/promise) и фазы 2 (propose/accept), т.е. семейства множеств таких, что каждый кворум фазы 1 пересекается с кворумом фазы 2.

Такая модификация PAXOS называется "Flexible PAXOS".

- https://arxiv.org/pdf/1608.06696.pdf,
- https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-935.pdf, глава "Quorum intersection revised".

Ослабление требований к кворумам

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.

Если в эпоху n было выбрано значение (n, v), то все запросы $\mathrm{propose}(n + k, \ \cdot)$, где $k > 0$, предлагают значение v.

Это свойство остаётся верным, если множество acceptor'ов, ответивших promise на шаге 3, пересекается с множеством acceptor'ов, выбравших значение (n, v). Множества acceptor'ов из пунктов 3 и 5 не обязаны быть большинствами – достаточно, чтобы они пересекались.

Можно рассматривать кворумы для фазы 1 (prepare/promise) и фазы 2 (propose/accept), т.е. семейства множеств таких, что каждый кворум фазы 1 пересекается с кворумом фазы 2.

Пример: если система состоит из N процессов, то в качестве кворумов фазы 1 можно взять подмножества из N-1 процессов, а кворумов фазы 2 – подмножества из двух процессов.

Такая модификация PAXOS называется "Flexible PAXOS".

- https://arxiv.org/pdf/1608.06696.pdf,
- https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-935.pdf, глава "Quorum intersection revised".

Multi-PAXOS + Flexible PAXOS

Будем предполагать, что каждый процесс-участник PAXOS исполняет все три роли: proposer, acceptor и learner, притом один из процессов играет роль выделенного proposer’а, т.е. только он испускает предлагаемые значения.

Тогда N-я итерация рабочего цикла реплицированного журнала выглядит так:

1. подготовить новый номер эпохи и разослать prepare,
2. дождаться promise от остальных участников,
3. разослать propose со значением записи N,
4. дождаться ответов от других участников,
5. разослать другим участникам commit, чтобы надёжно сохранить длину журнала,
6. дождаться ответа на commit и ответить клиенту, инициировавшему шаг N.

Наблюдение: трудоёмкость шагов 3-6 определяется размером кворума второй фазы PAXOS. Его можно сделать меньше, увеличив кворумы для шагов 1-2.

Replicated FSM

Проблема: журнал изменений состояния FSM растёт неограниченно, а место на жёстком диске для хранения состояния реплики конечно.

Replicated FSM

Проблема: журнал изменений состояния FSM растёт неограниченно, а место на жёстком диске для хранения состояния реплики конечно.

Решение: периодически делать снимки состояния FSM и хранить только снимок состояния и хвост журнала.

Replicated FSM

Проблема: журнал изменений состояния FSM растёт неограниченно, а место на жёстком диске для хранения состояния реплики конечно.

Решение: периодически делать снимки состояния FSM и хранить только снимок состояния и хвост журнала.

Трудности:

1. Журнал должен начинаться с момента времени, когда сделан снимок состояния: реализация журнала должна взаимодействовать с приложением, которое ведёт журнал, чтобы согласовывать его со снимками состояния.
2. Создание снимка состояния не должно блокировать модификацию журнала.
3. Отставшие узлы больше не могут догонять (catch-up) кластер простым копированием хвоста журнала. Иногда им надо скачать целиком снимок состояния. Что будет, если узел, с которого скачивают снимок состояния, решит в это время сделать новый снимок?

Напоминание: 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.

Network-ordered PAXOS

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.

Программируемые сетевые свитчи позволяют реализовать Ordered Unreliable Multicast, посылку сообщений группе процессов со следующими свойствами:

1. Если сообщения m и m' рассылаются множеству процессов R, то каждый процесс из R, получивший m и m', получил их в одном и том же порядке.
2. Если сообщение m рассылается множеству процессов R, то
   1. Каждый процесс из R получает сообщение m или сообщение "m было потеряно", либо,
   2. Ни один из процессов из R не получает ни сообщение m, ни сообщение "m было потеряно".

С помощью Ordered Unreliable Multicast можно рассылать сообщения propose и accept. Когда в сети нет потерь, OUM обеспечивает, что все acceptor'ы получают propose в одном и том же порядке и могут их сразу же считать их выбранными. При наличии потерь в сети сообщение "propose или accept был потерян" служат сигналом к возвращению к фазе 1 multi-PAXOS, перевыбору выделенного proposer'а и перезапуску фазы 2 multi-PAXOS.

- https://www.usenix.org/system/files/conference/osdi16/osdi16-li.pdf
  Just say NO to PAXOS overhead: replacing consensus with network ordering.

Корректность процессов-участников

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.

В общем случае это предположение неверно. Нездоровые участники кластера могут отвечать мусором на запросы:

- диск или память процесса-участника могут измениться и содержать мусор,
- в реализации процесса-участника могут быть ошибки.

- Проверки согласованности состояния, BUG_ON,
- Рандомизированное тестирование и fault injection.

Корректность процессов-участников

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.

В общем случае это предположение неверно. Нездоровые участники кластера могут отвечать мусором на запросы:

- диск или память процесса-участника могут измениться и содержать мусор,
- в реализации процесса-участника могут быть ошибки.

- Проверки согласованности состояния, BUG_ON,
- Рандомизированное тестирование и fault injection.

См. также:

- Distributed systems safety research.
  https://jepsen.io
- Chaos engineering.
  https://github.com/netflix/chaosmonkey.

Ещё один пример сложного пространства состояний

```
pthread_mutex_t mtx;
pthread_cond_t cv;
...
std::deque<int> queue;
constexpr sizeLimit = ...;
...


void put(int x)
{
    pthread_mutex_lock(&mtx);

    while (queue.size() >= sizeLimit)
        pthread_cond_wait(&cv, &mtx);
    queue.push_back(x);

    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&mtx);
}
```

```
int get()
{
    int x;

    pthread_mutex_lock(&mtx);

    while (queue.empty())
        pthread_cond_wait(&cv, &mtx);
    x = queue.pop_front();

    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&mtx);

    return x;
}
```

Задача простая: реализовать очередь ограниченной длины, поддерживающую много одновременных читателей и писателей.

В реализации есть проблема, которая

1. проявляется только когда число клиентов очереди велико по сравнению с максимальной длиной,
2. проявляется редко даже при выполнении №1.

Ещё один пример сложного пространства состояний

```
pthread_mutex_t mtx;
pthread_cond_t cv;
...
std::deque<int> queue;
constexpr sizeLimit = ...;
...


void put(int x)
{
    pthread_mutex_lock(&mtx);

    while (queue.size() >= sizeLimit)
        pthread_cond_wait(&cv, &mtx);
    queue.push_back(x);

    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&mtx);
}
```

```
int get()
{
    int x;

    pthread_mutex_lock(&mtx);

    while (queue.empty())
        pthread_cond_wait(&cv, &mtx);
    x = queue.pop_front();

    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&mtx);

    return x;
}
```

Задача простая: реализовать очередь ограниченной длины, поддерживающую много одновременных читателей и писателей.

В реализации есть проблема, которая

1. проявляется только когда число клиентов очереди велико по сравнению с максимальной длиной,
2. проявляется редко даже при выполнении №1.

Ещё один пример сложного пространства состояний

```
pthread_mutex_t mtx;
pthread_cond_t cv;
...
std::deque<int> queue;
constexpr sizeLimit = ...;
...


void put(int x)
{
    pthread_mutex_lock(&mtx);

    while (queue.size() >= sizeLimit)
        pthread_cond_wait(&cv, &mtx);
    queue.push_back(x);

    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&mtx);
}
```

```
int get()
{
    int x;

    pthread_mutex_lock(&mtx);

    while (queue.empty())
        pthread_cond_wait(&cv, &mtx);
    x = queue.pop_front();

    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&mtx);

    return x;
}
```

Задача простая: реализовать очередь ограниченной длины, поддерживающую много одновременных читателей и писателей.

В реализации есть проблема, которая

1. проявляется только когда число клиентов очереди велико по сравнению с максимальной длиной,
2. проявляется редко даже при выполнении №1.

Читатель, вместо того, чтобы разбудить спящего писателя, может разбудить другого читателя, и наоборот.

Модель ограниченной очереди

Напишем спецификацию для данной программы на TLA+:
Specifying systems, https://lamport.azurewebsites.net/tla/book-02-08-08.pdf.

TLA+ интегрируется с доказателями теорем. Его можно использовать для доказательства свойств алгоритмов.

Нам будет более интересен TLA+ Model Checker, который производит поиск по пространству состояний системы и может находить кратчайшую последовательность действий, приводящую к нарушению заданных инвариантов.

Модель ограниченной очереди

TLA+ code

```
CONSTANT BufferCapacity
CONSTANTS Readers, Writers
```

LaTeX-formatted

Для начала определим константы, задающие нашу систему:

- максимальную длину очереди,
- множества потоков-читателей и потоков-писателей.

Модель ограниченной очереди

TLA+ code

```
ASSUME Readers /= {}
ASSUME Writers /= {}
ASSUME (BufferCapacity \in Nat) /\ (BufferCapacity > 0)
```

LaTeX-formatted

$$\begin{aligned}&\text{ASSUME } Readers \neq \{\}\\&\text{ASSUME } Writers \neq \{\}\\&\text{ASSUME } (BufferCapacity \in Nat) \land (BufferCapacity > 0)\end{aligned}$$

- {} – пустое множество,
- /= – не равно,
- /\ и \/ – логические «и» и «или», соответственно.

Модель ограниченной очереди

TLA+ code | LaTeX-formatted

```
VARIABLES bufferLen, threadStates
```

Список переменных, определяющих состояние модели. Пока что они могут быть произвольными множествами.

Модель ограниченной очереди

TLA+ code

```
Threads == Readers \cup Writers

(* "R" is for Running,
   "M" is "owns Mutex and running",
   "S" is sleeping *)

ThreadState == {"R", "M", "S"}

threadStates_TypeInvariant ==
  threadStates \in [Threads -> ThreadState]
```

LaTeX-formatted

$$Threads \triangleq Readers \cup Writers$$

$$ThreadState \triangleq \{\text{“R”}, \text{“M”}, \text{“S”}\}$$
$$threadStates\_TypeInvariant \triangleq threadStates \in [Threads \to ThreadState]$$

Здесь мы требуем, чтобы переменная threadStates на каждом шаге была отображением, которое сопоставляет потоку одно из возможных его состояний.

- == – «по определению»,
- {A, B, …} – множество из элементов A, B, …
- [X -> Y] – множество отображений из X в Y.

Модель ограниченной очереди

TLA+ code

```
Init ==
  /\ bufferLen = 0
  /\ threadStates_TypeInvariant
  /\ \A t \in Threads: threadStates[t] = "R"
```

LaTeX-formatted

$$
\begin{aligned}
&Init \triangleq \\
&\land\ bufferLen = 0 \\
&\land\ threadStates\_TypeInvariant \\
&\land\ \forall\, t \in Threads : threadStates[t] = \text{“R”}
\end{aligned}
$$

Условие на начальное состояние системы. Мы потребуем его выполнения позже.

Модель ограниченной очереди

TLA+ code

```
(* t acquires the mutex *)
AcquireLock(t) ==
  /\ threadStates[t] = "R"
  /\ \forall ta \in Threads: threadStates[ta] /= "M"
  /\ threadStates' = [threadStates EXCEPT ![t] = "M"]
  /\ UNCHANGED bufferLen
```

LaTeX-formatted

$$
\begin{aligned}
&AcquireLock(t) \triangleq \\
&\land\ threadStates[t] = \text{“R”} \\
&\land\ \forall\, ta \in Threads : threadStates[ta] \neq \text{“M”} \\
&\land\ threadStates' = [threadStates \text{ EXCEPT } ![t] = \text{“M”}] \\
&\land\ \text{UNCHANGED } bufferLen
\end{aligned}
$$

Выражение AcquireLock(t) истинно, если на текущем шаге поток t становится владельцем мьютекса.

- F[t] – значение отображения F в точке t,
- x и x’ – значения переменной x на текущем и следующем шагах, соответственно,
- UNCHANGED <<x, y, …>> раскрывается в x’ = x /\ y’ = y /\ …

Модель ограниченной очереди

TLA+ code

```
ReadBlock(t) ==
  /\ t \in Readers
  /\ threadStates[t] = "M"
  /\ bufferLen = 0
  /\ threadStates' = [threadStates EXCEPT ![t] = "S"]
  /\ UNCHANGED bufferLen
```

LaTeX-formatted

$$
\begin{aligned}
&ReadBlock(t) \triangleq \\
&\land\ t \in Readers \\
&\land\ threadStates[t] = \text{“M”} \\
&\land\ bufferLen = 0 \\
&\land\ threadStates' = [threadStates\ \text{EXCEPT}\ ![t] = \text{“S”}] \\
&\land\ \text{UNCHANGED}\ bufferLen
\end{aligned}
$$

Выражение ReadBlock(t) истинно, если на текущем шаге поток t попробовал прочесть из очереди, обнаружил, что она пуста, и отправился спать.

Модель ограниченной очереди

TLA+ code

```
ReadOk(t) ==
  /\ t \in Readers
  /\ threadStates[t] = "M"
  /\ bufferLen > 0
  /\ bufferLen' = bufferLen - 1
  /\ UnlockAndWakeOne(t)
```

LaTeX-formatted

$$
\begin{aligned}
&ReadOk(t) \triangleq \\
&\land\ t \in Readers \\
&\land\ threadStates[t] = \text{“M”} \\
&\land\ bufferLen > 0 \\
&\land\ bufferLen' = bufferLen - 1 \\
&\land\ UnlockAndWakeOne(t)
\end{aligned}
$$

Выражение ReadOk(t) истинно, если поток t на текущем шаге сумел прочесть значение из очереди и разбудил какой-то другой поток. Выражение UnlockAndWakeOne(t) определим ниже.

Модель ограниченной очереди

TLA+ code

```
UnlockAndWakeOne(self) ==
LET
 sleepingThreads == {t \in Threads: threadStates[t] = "S"}
IN
 IF sleepingThreads = {}
 THEN
  threadStates' = [threadStates EXCEPT ![self] = "R"]
 ELSE
  \E t \in sleepingThreads:
  threadStates' = [threadStates EXCEPT ![self] = "R", ![t] = "R"]
```

LaTeX-formatted

$$
\begin{array}{l}
UnlockAndWakeOne(self) \triangleq \\
\text{LET} \\
sleepingThreads \triangleq \{t \in Threads : threadStates[t] = \text{“S”}\} \\
\text{IN} \\
\text{IF } sleepingThreads = \{\} \\
\quad \text{THEN} \\
threadStates' = [threadStates \text{ EXCEPT } ![self] = \text{“R”}] \\
\quad \text{ELSE} \\
\exists\, t \in sleepingThreads : \\
threadStates' = [threadStates \text{ EXCEPT } ![self] = \text{“R”}, ![t] = \text{“R”}]
\end{array}
$$

Модель ограниченной очереди

TLA+ code

```
Next == \E t \in Threads:
  \/ AcquireLock(t)
  \/ WriteOk(t)
  \/ WriteBlock(t)
  \/ ReadOk(t)
  \/ ReadBlock(t)
```

LaTeX-formatted

$$
\begin{aligned}
&Next \triangleq \exists\, t \in Threads : \\
&\lor\ AcquireLock(t) \\
&\lor\ WriteOk(t) \\
&\lor\ WriteBlock(t) \\
&\lor\ ReadOk(t) \\
&\lor\ ReadBlock(t)
\end{aligned}
$$

Выражение, истинное, если система сделала один шаг. Шагом может быть одно из действий:

- поток взял мьютекс,
- поток попробовал записать или записал значение в очередь,
- поток попробовал прочесть или прочёл значение из очереди.

Модель ограниченной очереди

TLA+ code

```
Spec == Init /\ [][Next]_<<bufferLen, threadStates>>
```

LaTeX-formatted

$$Spec \triangleq Init \land \Box[Next]_{\langle bufferLen,\, threadStates \rangle}$$

Определение нашей модели: выполняется условие Init (на начальное состояние системы) и на каждом шаге выполняется условие Next.

- Выражение []E истинно тогда и только тогда, когда E истинно на каждом шаге.

Модель ограниченной очереди

TLA+ code

```
AtMostOneMutexOwner ==
 LET owners == {t \in Threads: threadStates[t] = "M"}
 IN Cardinality(owners) <= 1

Liveness ==
 LET
 runnable == {t \in Threads: threadStates[t] \in {"R", "M"}}
 IN runnable /= {}
```

LaTeX-formatted

$$
\begin{array}{l}
AtMostOneMutexOwner \triangleq \\
\text{LET } owners \triangleq \{t \in Threads : threadStates[t] = \text{“M”}\} \\
\text{IN } \quad Cardinality(owners) \leq 1
\end{array}
$$

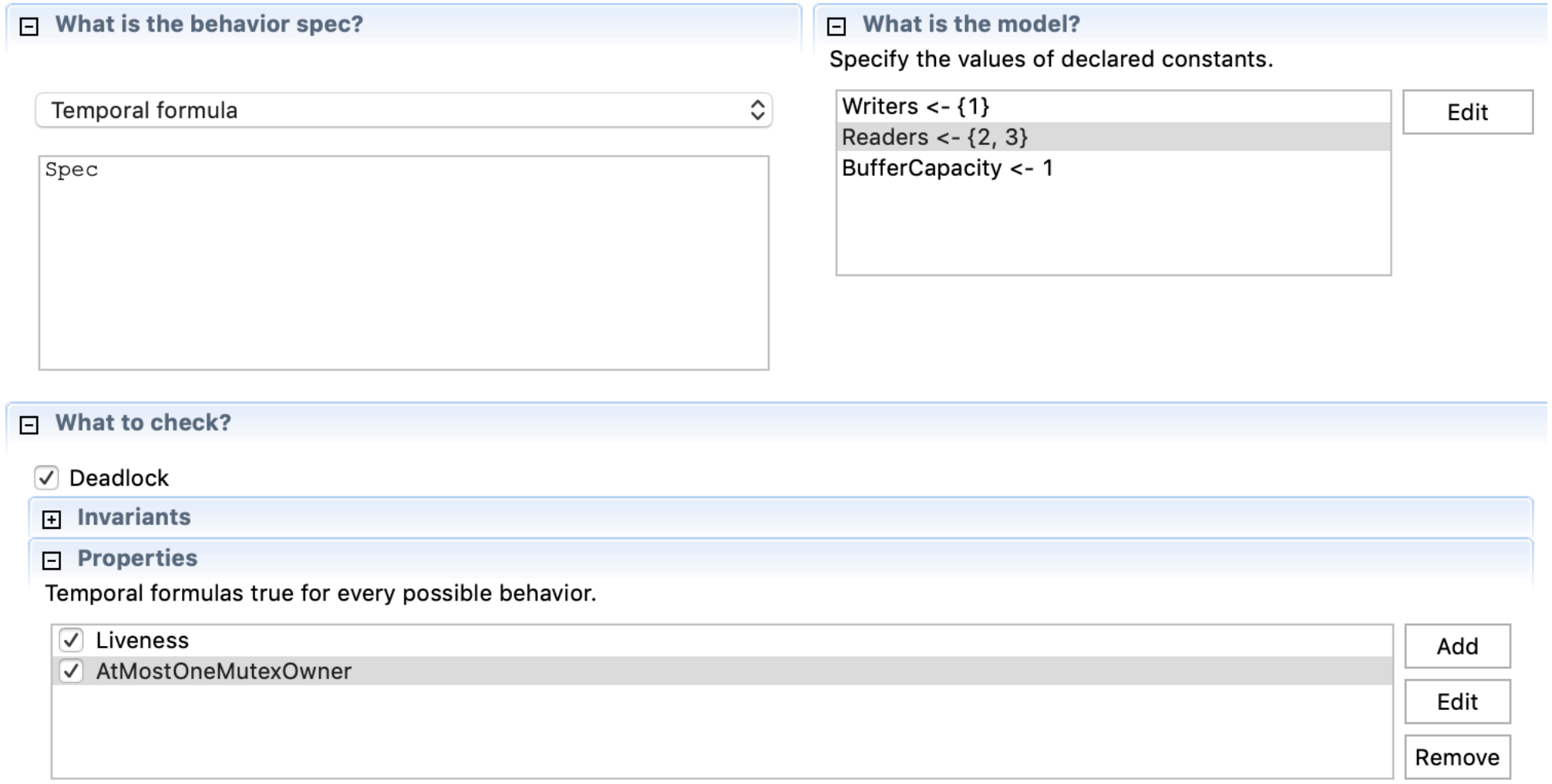
$$
\begin{array}{l}
Liveness \triangleq \\
\text{LET } runnable \triangleq \{t \in Threads : threadStates[t] \in \{\text{“R”}, \text{“M”}\}\} \\
\text{IN } \quad runnable \neq \{\}
\end{array}
$$

Добавим два выражения, которые истинны тогда и только тогда, когда

- мьютексом владеет не более одного потока,
- имеется хотя бы один поток, способный исполняться дальше.

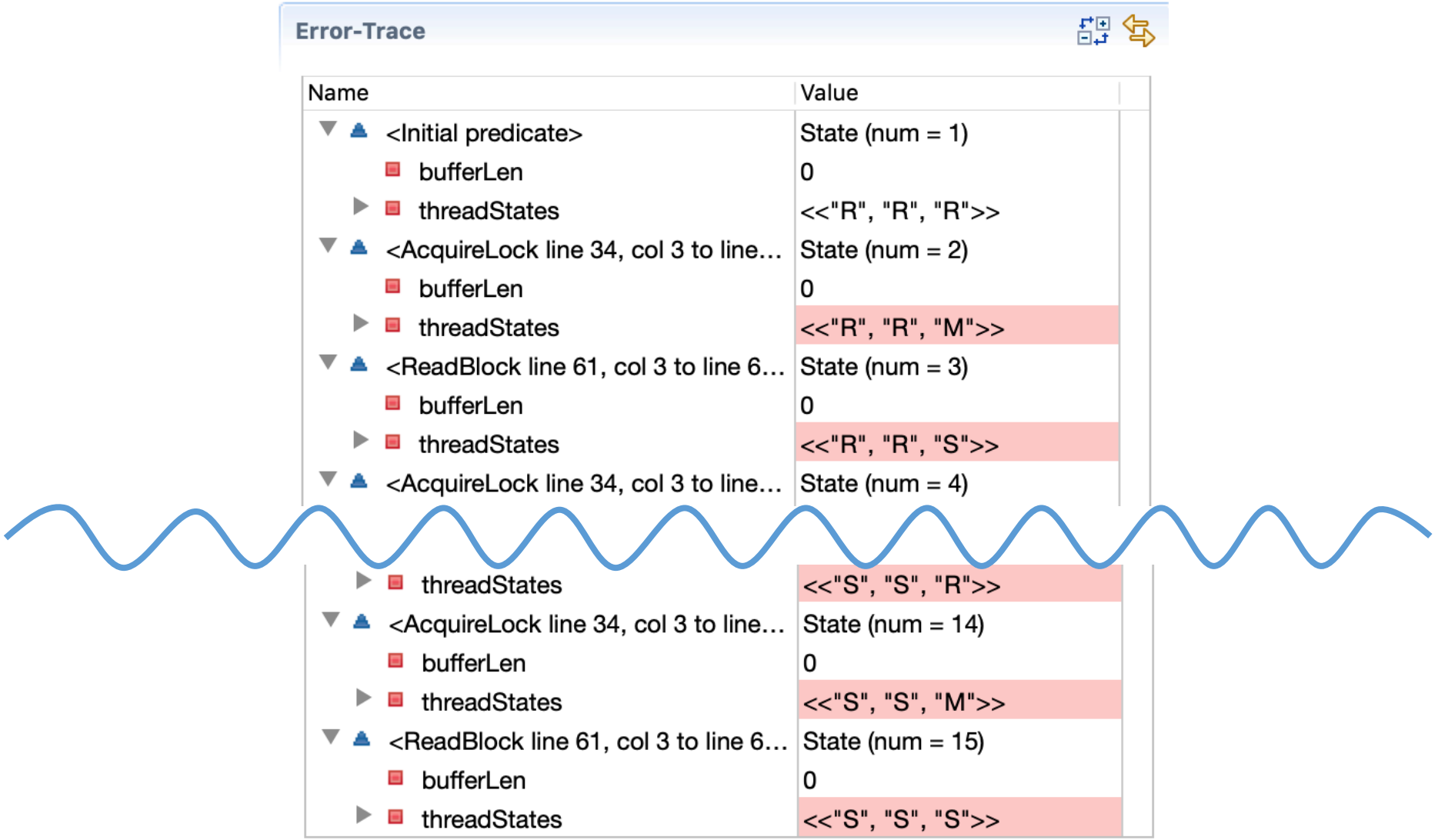
Модель ограниченной очереди

Теперь обойдём состояния нашей модели с помощью TLA+ Model Checker:



Модель ограниченной очереди

Теперь обойдём состояния нашей модели с помощью TLA+ Model Checker:



Домашнее задание

Найдите с помощью TLC решение задачи о волке, козе и капусте.

Список литературы

1. Specifying systems, https://lamport.azurewebsites.net/tla/book-02-08-08.pdf.
2. TLA+ Toolbox https://lamport.azurewebsites.net/tla/toolbox.html.