Распределенные информационно-аналитические системы

Лекция № 6.

Синхронизация времени и выбор лидера (координатора) в распределенных системах

Профессор кафедры КБ-2: д.т.н. Шатовкин Р.Р.

Учебные цели:

Изучить основы научных знаний по синхронизации времени; выбору лидера (координатора); консенсусу.

Учебные вопросы:

1. Синхронизация времени.

2. Выбор лидера (координатора).

3. Консенсус.

Обычно децентрализованные алгоритмы имеют следующие свойства:

- информация распределена среди ЭВМ;

- процессы принимают решение на основе только локальной информации;

- не должно быть единственной критической точки, выход из строя которой приводил бы к краху алгоритма;

- не существует общих часов или другого источника точного глобального времени.

Первые три пункта говорят о недопустимости сбора всей информации для принятия решения в одно место. Обеспечение синхронизации без централизации требует подходов, отличных от используемых в традиционных ОС.

Последний пункт также очень важен – в распределенных системах достигнуть согласия относительно времени совсем непросто. Важность наличия единого времени можно оценить на примере программы make в ОС UNIX. Главные теоретические проблемы – отсутствие глобальных часов и невозможность зафиксировать глобальное состояние (для анализа ситуации – обнаружения дедлоков, для организации промежуточного запоминания).

1. Синхронизация времени

Синхронизация необходима процессам для организации совместного использования ресурсов, таких как файлы или устройства, а также для обмена данными.

В однопроцессорных системах решение задач взаимного исключения, критических областей и других проблем синхронизации осуществлялось с использованием общих методов, таких как семафоры и мониторы. Однако эти методы не совсем подходят для распределенных систем, так как все они базируются на использовании разделяемой оперативной памяти. Например, два процесса, которые взаимодействуют, используя семафор, должны иметь доступ к нему. Если оба процесса выполняются на одной и той же машине, они могут иметь совместный доступ к семафору, хранящемуся, например, в ядре, делая системные вызовы. Однако, если процессы выполняются на разных машинах, то этот метод не применим, для распределенных систем нужны новые подходы.

В централизованной однопроцессорной системе, как правило, важно только относительное время и не важна точность часов. В распределенной системе, где каждый процессор имеет собственные часы со своей точностью хода, ситуация резко меняется: программы, использующие время (например, программы, подобные команде make в UNIX, которые используют время создания файлов, или программы, для которых важно время прибытия сообщений и т.п.) становятся зависимыми от того, часами какого компьютера они пользуются.

В распределенных системах синхронизация физических часов (показывающих реальное время) является сложной проблемой, но с другой стороны очень часто в этом нет никакой необходимости: то есть процессам не нужно, чтобы во всех машинах было правильное время, для них важно, чтобы оно было везде одинаковое, более того, для некоторых процессов важен только правильный порядок событий. В этом случае мы имеем дело с логическими часами.

Алгоритм синхронизации логических часов. В 1978 году Лэмпорт (Lamport) показал, что синхронизация времени возможна, и предложил алгоритм для такой синхронизации. При этом он указал, что абсолютной синхронизации не требуется. Если два процесса не взаимодействуют, то единого времени им не требуется. Кроме того, в большинстве случаев согласованное время может не иметь ничего общего с астрономическим временем, которое объявляется по радио. В таких случаях можно говорить о логических часах.

Введем для двух произвольных событий отношение «случилось до». Выражение $a \rightarrow b$ читается «a случилось до b» и означает, что все процессы в системе считают, что сначала произошло событие a, а потом – событие b.

Отношение «случилось до» обладает свойством транзитивности: если выражения $a \rightarrow b$ и $b \rightarrow c$ истинны, то справедливо и выражение $a \rightarrow c$.

Для двух событий одного и того же процесса всегда можно установить отношение «случилось до», аналогично может быть установлено это отношение и для событий передачи сообщения одним процессом и приемом его другим, так как прием не может произойти раньше отправки. Однако, если два произвольных события случились в разных процессах на разных машинах, и эти процессы не имеют между собой никакой связи (даже косвенной через третьи процессы), то нельзя сказать с полной определенностью, какое из событий произошло раньше, а какое позже.

Ставится задача создания такого механизма ведения времени, который бы для каждого события a мог указать значение времени T(a), с которым бы были согласны все процессы в системе. При этом должно выполняться условие: если $a \rightarrow b$, то $T(a) < T(b)$. Кроме того, время может только увеличиваться и, следовательно, любые корректировки времени могут выполняться только путем добавления положительных значений, и никогда – путем вычитания.

Рассмотрим алгоритм решения этой задачи, который предложил Лэмпорт. Для отметок времени в нем используются события:

1. Часы T_i увеличивают свое значение с каждым событием в процессе P_i: $T_i = T_i + d$ ($d > 0$, обычно равно 1).

2. Если событие a есть посылка сообщения m процессом P_i, тогда в это сообщение вписывается временная метка $t_m = T_i(a)$. В момент получения этого сообщения процессом P_j его время корректируется следующим образом: $T_j = \max(T_j, t_m + d)$.

На рисунке 1 показаны три процесса, выполняющихся на разных машинах, каждая из которых имеет свои часы, идущие со своей скоростью.

Как видно из рисунка, когда часы процесса 0 показали время 6, в процессе 1 часы показывали 8, а в процессе 2 – 10. Предполагается, что все эти часы идут с постоянной для себя скоростью.

В момент времени 6 процесс 0 посылает сообщение A процессу 1. Это сообщение приходит к процессу 1 в момент времени 16 по его часам. В логическом смысле это вполне возможно, так как 6 < 16. Аналогично, сообщение B, посланное процессом 1 процессу 2 в момент времени 24, пришло к последнему в момент времени 40, то есть его передача заняла 16 единиц времени, что также является правдоподобным.

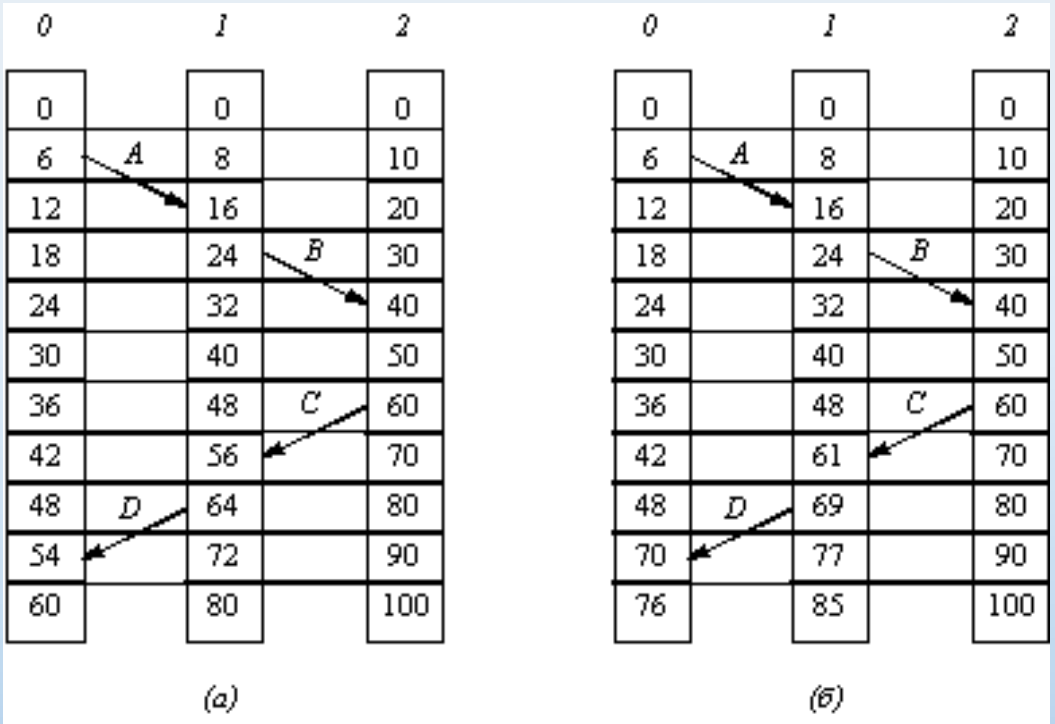

Рисунок 1 – Синхронизация логических часов:
а – три процесса, каждый со своими собственными часами;
б – алгоритм синхронизации логических часов

Ну а далее начинаются весьма странные вещи. Сообщение C от процесса 2 к процессу 1 было отправлено в момент времени 60, а поступило в место назначения в момент времени 56. Очевидно, что это невозможно. Такие ситуации необходимо предотвращать.

Решение Лэмпорта вытекает непосредственно из отношений «случилось до». Так как C было отправлено в момент 60, то оно должно дойти в момент 61 или позже. Следовательно, каждое сообщение должно нести с собой время своего отправления по часам машины-отправителя. Если в машине, получившей сообщение, часы показывают время, которое меньше времени отправления, то эти часы переводятся вперед, так, чтобы они показали время, большее времени отправления сообщения. На рисунке 1б видно, что C поступило в момент 61, а сообщение D – в 70.

Алгоритм Кристиана – один из простейших алгоритмов распределенных систем. Каждый из узлов, которым требуется синхронизированное время, посылает сообщение серверу времени (узлу-носителю эталонного времени) с запросом времени. Сервер времени максимально оперативно посылает клиенту ответное сообщение, содержащее текущее время сервера (рисунок 2).

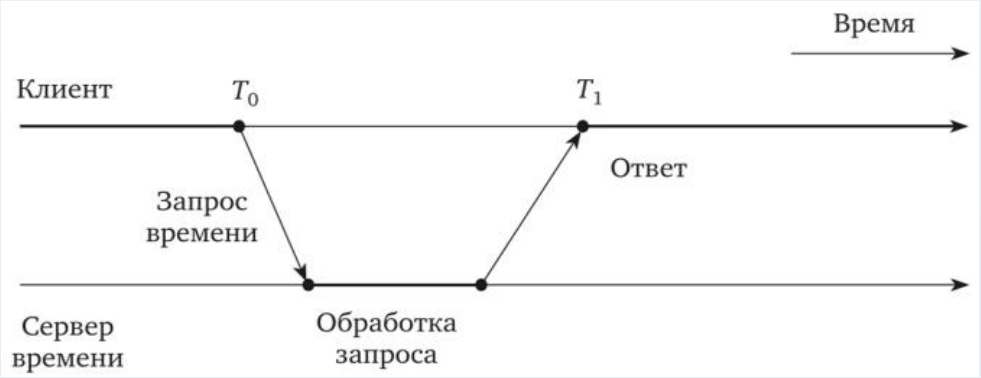

Рисунок 2 – Получение текущего времени с сервера времени

Но следует помнить, что между отправкой сообщения эталоном и его получением узлом проходит некоторое время, которое непостоянно и зависит от текущей загрузки сети передачи данных. Для оценки этого времени Кристиан предложил принимать время передачи ответа как половину времени между отправкой запроса и получением ответа. Для повышения точности следует производить серию таких замеров.

Алгоритм Кристиана работает между процессом P и сервером времени S, подключенным к источнику привязки времени:

1. P запрашивает время у S в момент времени T_0.
2. После получения запроса от P, S готовит ответ и добавляет время T из своих собственных часов.
3. P получает ответ в момент времени T_1, затем устанавливает его время равным $T + R / 2$, где $R = T_1 – T_0$.

Если R фактически разделен поровну между запросом и ответом, синхронизация проходит без ошибок. Но из-за непредсказуемых воздействий это предположение часто неверно. Более длинные R указывают на асимметричные помехи. Таким образом, смещение и дрожание синхронизации сводятся к минимуму путем выбора подходящего R из множества пар запрос/ответ. Возможность принятия R в данный момент времени зависит от отклонения часов и от статистики R. Эти величины можно измерить в процессе синхронизации, что само по себе оптимизирует метод.

Достоинства: простота реализации, высокая эффективность в небольших сетях и сетях с малой загрузкой: поскольку в системе присутствует источник точного времени и синхронизация производится по нему, то время в системе в целом соответствует реальному.

Недостатки: требует внешнего источника точного времени, не имеет встроенной защиты от перевода часов в обратную сторону, некорректно работает в сетях с резкими скачками загрузки сети, не позволяет корректно устанавливать время в случае, если запрос и ответ передаются по разным маршрутам (требуется разное время для передачи данных сообщений).

Алгоритм Беркли. Данный алгоритм предназначен для случая, когда источник точного времени отсутствует. Активный компонент, называемый в данном алгоритме ***сервером времени*** (заметим, что такое название противоречит обычно принятому, так как серверами принято называть пассивные компоненты, активизирующиеся только по запросу), опрашивает все узлы для выяснения их текущего времени, усредняет это значение и рассылает команды на установку нового значения времени (консолидированного времени) либо замедление часов.

В отличие от алгоритма Кристиана, процесс сервера в алгоритме Беркли, называется *лидером*, который периодически опрашивает другие ведомые процессы.

Этот алгоритм применяют для синхронизации физического времени в замкнутой системе.

Если распределенная система имеет значительные размеры, то при подсчете консолидированного времени рекомендуется не принимать во внимание партнеров с наибольшим расхождением часов.

Алгоритм достаточно прозрачен:

1. Вычислить все расхождения времени.

2. Сортировать их по абсолютной величине.

3. Пометить какое-то количество наиболее расходящихся по времени (с наибольшим значением), которые не будут учитываться при подсчете консолидированного времени.

4. Усреднить время, полученное от тех партнеров, у которых удалось получить показания часов. Заметим, что мы не можем просто сложить показания всех часов и поделить их на количество, так как для представления времени мы обычно используем целочисленный тип данных, имеющий конечную разрядную сетку.

5. Разослать всем партнерам консолидированное время.

Усредняющие алгоритмы. Несмотря на то, что рассмотренные алгоритмы Кристиана и Беркли имеют различный механизм синхронизации, у них имеется одно свойство, не очень любимое в распределенных системах – они оба централизованные. Децентрализованные алгоритмы тоже существуют. Рассмотрим один из них.

Один из параметров алгоритма – интервал времени A_T, в течение которого должны договориться взаимодействующие процессы. Все глобальное время (точнее сказать, время каждого из процессов) делится на такие интервалы.

В начале каждого интервала каждый из процессов широковещательно рассылает всем участникам алгоритма свое значение времени. Система распределенная, следовательно, инициация посылок произойдет для всех процессов не в один единственный момент времени, а в пределах какого-то интервала. Одновременно процесс должен запустить процесс получения всех таких сообщений, посланных другими процессами.

Важно выбрать интервал A_T таким, чтобы в течение этого времени все сообщения от корректно функционирующих процессов были доставлены и обработаны.

Обработка сообщений в простейшем виде заключается в усреднении присланных значений времени и корректировке собственных часов в стиле алгоритма Кристиана.

Защитой от недобросовестных процессов, посылающих неверное значение, может служить фильтр, отбрасывающий определенное количество сообщений, содержащих наименьшие и наибольшие значения.

Отметим два важных момента, которые должны учитывать все алгоритмы синхронизации времени:

- часы не должны ходить назад (надо ускорять или замедлять их для проведения коррекции);

- наличие ненулевого времени прохождения сообщения о времени (можно многократно замерять время прохождения и брать среднее).

2. Выбор лидера (координатора)

Синхронизация может стоить достаточно дорого: когда каждый шаг алгоритма вовлечен в контакт с каждым иным участником, наблюдается значительный расход ресурса на организацию и поддержание взаимодействия. Это в особенности справедливо для крупных и географически распределенных сетевых сред. Для снижения расхода ресурса, обусловленного синхронизацией, а также общего числа передаваемых сообщений для достижения определенного решения, некоторые алгоритмы полагаются на наличие некого процесса ***лидера*** (***координатора***), ответственного за исполнение или координацию этапов распределенного алгоритма.

Как правило, процессы в распределенной системе единообразны и любой процесс может взять на себя роль такого лидера. Процессы предполагают лидерство на протяжении длительного времени, но это не постоянная роль. Как правило, такой процесс остается лидером до своего отказа. После отказа любой иной процесс способен запустить новый цикл выборов, предложить лидерство и, если он будет избран, продолжить работу вместо отказавшего лидера.

Сама *жизнеспособность* такого алгоритма гарантирует, что *в большинстве случаев* будет иметься некий лидер, а также такие выборы в конечном счете будут завершены (то есть данная система никогда не будет пребывать в подобном состоянии выборов неопределенно долго).

В идеале мы также предполагаем *безопасность*, и гарантируем что в некий определенный момент времени может быть *не более одного* лидера, и целиком исключаем возможность ситуации *расщепления сознания* (когда выбраны два лидера, обслуживающие одну и ту же цель, но не знают друг о друге). Тем не менее, на практике, многие алгоритмы выбора нарушают данное соглашение.

Процессы выбора лидера, к примеру, могут применяться для достижения общего порядка сообщений при их широковещательной раздаче. Лидер собирает и удерживает свое глобальное состояние, получает сообщения и распространяет их по всем имеющимся процессам. Он также может использоваться для координации реорганизации системы после ее отказа, в процессе инициализации или когда происходят важные изменения состояния.

Выборы включаются, когда инициализируется данная система, и лидер выбирается в самый первый раз, или же когда предыдущий лидер претерпевает отказ или перестает взаимодействовать. Выборы должны быть детерминистскими: по результатам этого процесса должен появиться лишь один лидер. Такое решение должно быть действенным для всех участников.

Хотя выборы лидера и распределенная блокировка (то есть исключительное обладание неким совместно используемым ресурсом) и выглядят с теоретической точки зрения схоже, они слегка отличаются. Когда один процесс удерживает некую блокировку для исполнения какого-то критического раздела, для прочих процессов не важно знать кто в точности удерживает блокировку прямо сейчас до тех пор, пока удовлетворяется сам процесс жизнеспособности (то есть такая блокировка в конечном итоге высвобождается, позволяя прочим овладевать ею). В противоположность этому, процесс выборов обладает некими особыми свойствами и обязан быть известен всем прочим участникам, следовательно его новый лидер обязан уведомить своих одноранговых партнеров о своей роли.

Когда некий алгоритм распределенной блокировки имеет определенный вид предпочтения для какого-то процесса или группы процессов, он в конечном итоге будет подавлять процессы без предпочтений в отношении этого разделяемого ресурса, что противоречит необходимому свойству живучести. В противоположность этому, выбранный лидер может оставаться в своей роли до тех пор, пока не претерпит отказа, при этом более предпочтительными являются долгоживущие лидеры.

Наличие некого стабильного лидера в рассматриваемой системе помогает избежать состояние рассинхронизации между удаленными участниками, снижает общее число обмена сообщениями и направление исполнения с какого-то отдельного процесса вместо требования координации одноранговой сети. Одна из основных потенциальных проблем в системах с представлением о лидерстве состоит в том, что такой лидер способен становиться узким местом. Для преодоления этого многие системы разделяют данные на разделы в не пересекающиеся независимые наборы реплик. Вместо наличия отдельного лидера по всей системе, каждый набор реплик обладает своим собственным лидером.

По причине того, что каждый лидирующий процесс в конечном итоге отказывает, отказ должен быть обнаружен, о нем должно быть выдано сообщение, а также на него должна быть реакция: система обязана выбрать иного лидера для замены вышедшего из строя.

Некоторые алгоритмы применяют временных лидеров для снижения общего числа сообщений необходимых для достижения некого согласия между имеющимися участниками. Тем не менее, эти алгоритмы применяют свои собственные особенные для алгоритма средства для выбора лидера, выявления отказа и разрешения конфликтов между соперничающими за лидерство процессами.

Алгоритм задиры. Один из алгоритмов выбора лидера, именуемый как ***алгоритм задиры*** (***bully algorithm***), использует ранги процессов для выявления своего нового лидера. Каждый процесс получает некий уникальный назначаемый ему ранг. В процессе проведения выборов процесс с наивысшим значением ранга становится лидером.

Этот алгоритм стал известным благодаря своей простоте. Данный алгоритм носит название *задиры* по той причине, что узел с наивысшим значением ранга «подавляет» прочие узлы, чтобы они приняли его в качестве лидера. Он также известен под названием ***монархического выбора лидера***: монархом становится собрат с наивысшим рейтингом после того, как прежний перестает существовать.

Выборы запускаются, когда один из имеющихся процессов замечает, что в его системе отсутствует лидер (он не был инициализирован) или предыдущий лидер перестал отвечать на запросы, и выполняются в три этапа:

1. Этот процесс отправляет сообщение о выборах процессам с более высокими идентификаторами.

2. Данный процесс ожидает, позволяя откликнуться процессам с более высоким рангом. Если ни один из процессов с более высоким рангом не отвечает, он выполняет шаг 3. В противном случае этот процесс отмечает ответивший ему процесс с наивысшим рангом и позволяет этому процессу выполнить шаг 3.

3. Процесс на этом шаге полагает, что больше нет активных процессов с более высоким рангом и уведомляет все процессы с более низким рангом о своем лидерстве.

Рисунок 3 иллюстрирует алгоритм задиры:

1) Процесс 3 замечает, что его предыдущий лидер 6 претерпел отказ и запустил новые выборы отправив сообщение ELECTION (выборы) процессам с более высокими значениями идентификаторов.

2) Процессы 4 и 5 отвечают ALIVE (живем), поскольку у них более высокий ранг чем у 3.

3) Процесс 3 обнаруживает, что откликнулся процесс 5 с наивысшим рангом.

4) Процесс 5 выбирается в качестве нового лидера. Он широковещательно рассылает сообщения ELECTED (избран), уведомляя процессы с более низким рангом о результатах этих выборов.

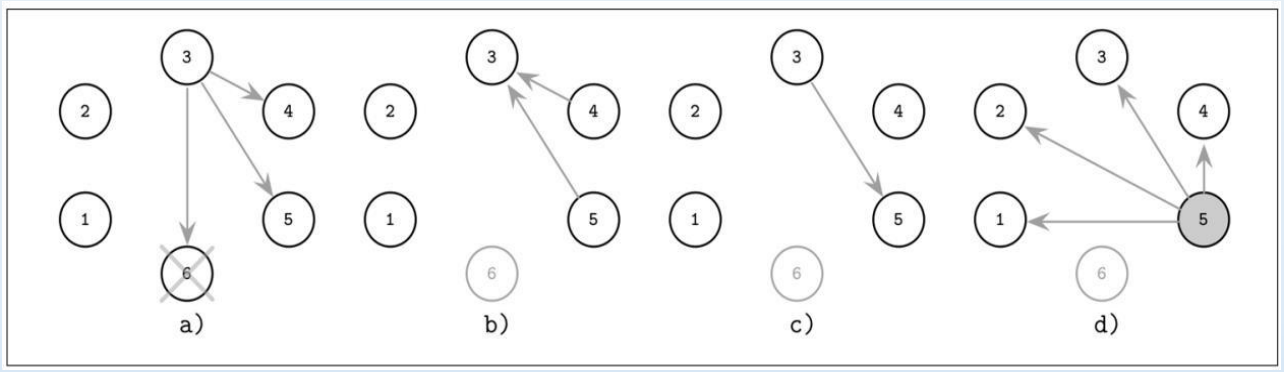

Рисунок 3 – Алгоритм задиры: предыдущий лидер (процесс 6) отказал и процесс 3 запустил новые выборы

Одной из очевидных проблем для данного алгоритма является то, что он нарушает необходимые гарантии безопасности (относительно того, что за раз может быть выбран лишь один лидер) при наличии разбиения сетевой среды на разделы. Развитие событий может запросто завершиться ситуацией, при которой узлы разбиваются на два или более независимо работающих подмножества и каждый из наборов выбирает своего собственного лидера. Такое положение дел носит название ***расщепления сознания***.

Другой проблемой для данного алгоритма является строгое предпочтение для узлов с самыми высокими рангами, что становится трудным случаем, когда они нестабильны и способны вызывать непрерывную череду перевыборов. Некий нестабильный узел с наивысшим рангом, предлагающий себя в качестве лидера, вскоре после этого откажет, выборы будут запущены, опять последует отказ и весь этот процесс повторится целиком. Эту задачу можно разрешить, распространяя замеры качества и принимая их в рассмотрении при самих выборах.

Отказоустойчивый алгоритм задиры с предпочтением. Существует множество вариаций алгоритма задиры, которые улучшают его различные свойства. К примеру, мы можем воспользоваться альтернативными предпочтительными по отказоустойчивости процессами для скорейших повторных выборов.

Всякий избранный лидер производит некий перечень отказоустойчивых узлов. Когда один из имеющихся процессов определяет отказ лидера, он запускает новые выборы, отправляя сообщение *альтернативе* с наивысшим рангом из списка, составленного отказавшим лидером. Когда одна из предлагаемых альтернатив поднята, она становится новым лидером без какой бы то ни было необходимости проходить полный раунд выборов.

Если тот процесс, который был определил отказ лидера сам по себе обладает наивысшим рангом из данного перечня, он может уведомлять прочие процессы о новом лидере напрямую.

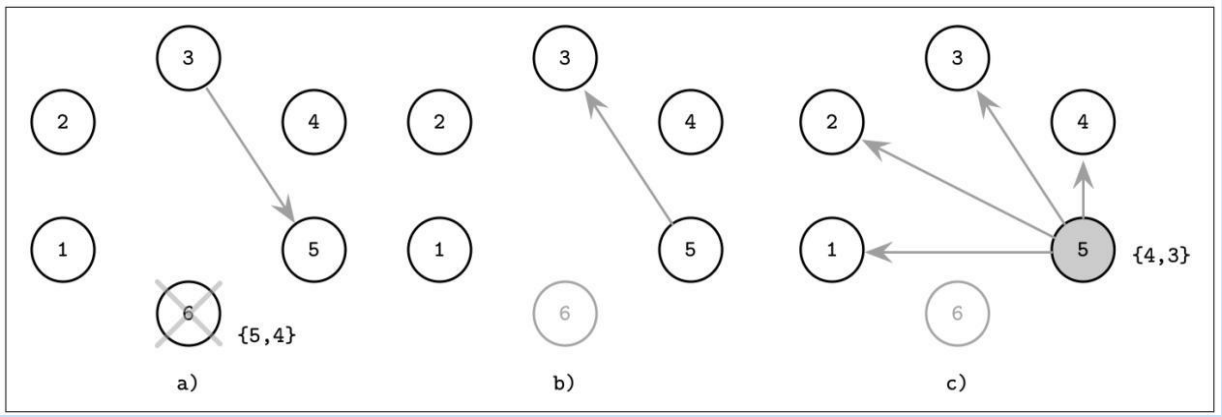

Рисунок 4 – Алгоритм задиры с предпочтением: предыдущий лидер (процесс 6) отказал и процесс 3 запустил новые выборы, контактируя с имеющейся альтернативой наивысшего ранга

На рисунке 4 показан алгоритм с данной оптимизацией:

1) Процесс 6 является лидером с обозначенными альтернативами {5,4}, претерпевает отказ, процесс 3 отмечает этот отказ и контактирует с процессом 5 – альтернативой из имеющегося списка с наивысшим рангом.

2) Процесс 5 отвечает процессу 3, что он работает, чтобы предотвратить тому контакты с прочими узлами из имеющегося перечня альтернатив.

3) Процесс 5 уведомляет все прочие узлы, что он новый лидер.

В результате нам требуется меньше шагов в алгоритме выбора с предпочтением.

Алгоритм задиры с оптимизацией «соискатель/ординарный процесс». Другой алгоритм пытается снизить требования в общем числе сообщений, расщепляя все узлы на два подмножества: ***соискателей*** и ***ординарных процессов***, в которых лишь один из имеющихся узлов кандидатов может в конечном итоге становиться лидером.

Ординарный процесс инициирует выборы контактами с узлами кандидатов, собирая от них отклики, указывая рабочего кандидата с наивысшим рангом в качестве нового лидера, а затем оповещая все оставшиеся узлы о полученных результатов выборов.

Для решения проблемы с множеством одновременных выборов данный алгоритм предлагает пользоваться переменной δ схемы разрешения конфликтов, некой конкретной для процесса задержкой, значительно меняющейся между имеющимися узлами, которая позволяет одному из существующих узлов инициировать необходимые выборы до того, как этим займутся прочие. Значение времени разрешения конфликтов обычно больше времени прохода сообщения в оба конца. Узлы с большими приоритетами обладают более низкими δ, и наоборот.

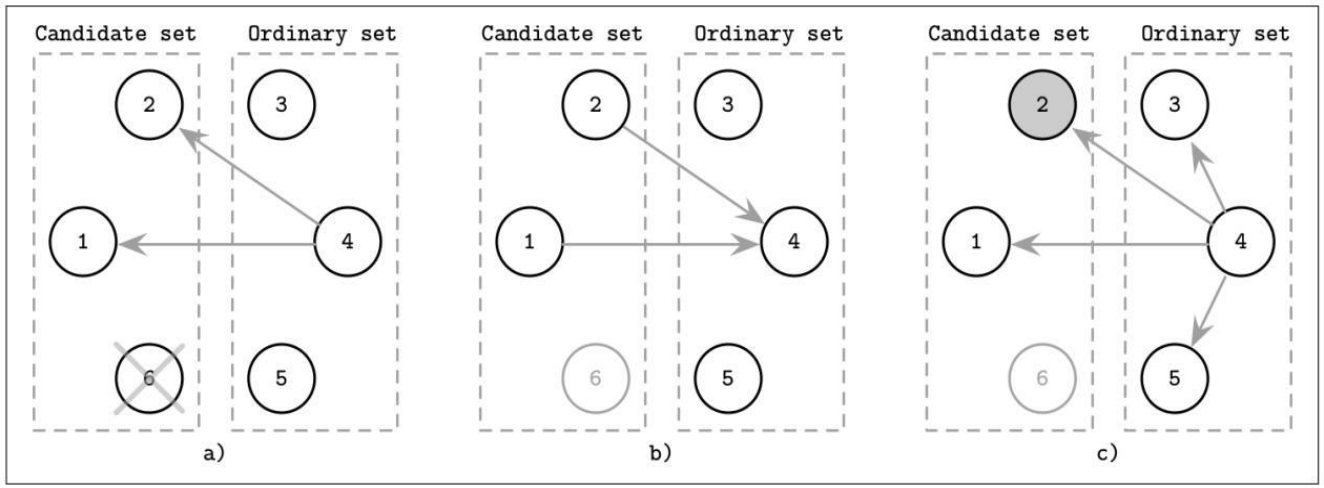

Рисунок 5 – Модификация алгоритма задиры с оптимизацией «соискатель/ординарный процесс»: предыдущий лидер (6) отказал и процесс 4 запустил новые выборы

Рисунок 5 отображает установленные этапы данного алгоритма выборов:

1) Процесс 4 из множества обычных отмечает отказ процесса-лидера 6. Он запускает новый раунд выборов контактируя с остающимися в своем наборе кандидатами.

2) Процессы кандидатов откликаются процессу 4, указывая что они в рабочем состоянии.

3) Процесс 4 уведомляет все процессы о новом лидере – процессе 2.

Алгоритм приглашения. Некий *алгоритм приглашения* позволяет процессам «зазывать» прочие процессы, соединяться с ними в группы вместо того, чтобы выставлять более высокий ранг им. Этот алгоритм допускает множество лидеров, *по определению*, поскольку всякая группа обладает своим собственным лидером.

Каждый процесс запускается как некий лидер какой-то новой группы, в которой только этот процесс и является участником группы. Лидеры групп контактируют одноранговым образом с не относящимися к их группам участниками, приглашая их к объединению. Если такой одноранговый процесс сам по себе является лидером, две группы сливаются. В противном случае находящийся в контакте процесс отвечает идентификатором лидера группы, позволяя двум лидерам групп наладить контакт и слить группы меньшим числом шагов.

Рисунок 6 отображает шаги, исполняемые этим алгоритмом приглашения:

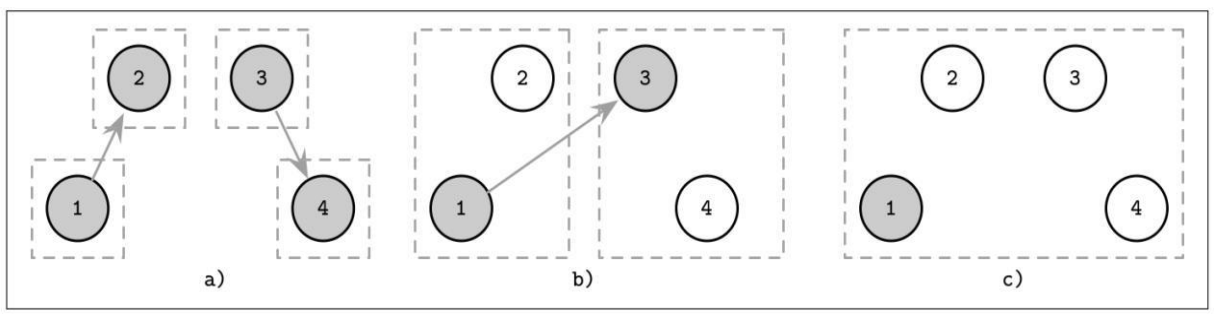

Рисунок 6 – Алгоритм приглашения

1) Четыре процесса начинают как лидеры групп, каждая из которых содержит единственного участника. Процесс 1 приглашает процесс 2 объединиться в группу, а процесс 3 «зазывает» в объединенную группу процесс 4.

2) Процесс 2 объединяется в группу с процессом 1, а процесс 4 объединяется в группу с процессом 3. Процесс 1, лидер своей первой группы, выходит на контакт с процессом 3, лидером другой группы. Остающиеся участники группы (в данном случае процесс 4) уведомляются о новом лидере группы.

3) Две группы сливаются, и процесс 1 становится лидером некой расширенной группы.

Поскольку группы сливаются, не имеет значения будет ли приглашенный процесс становиться лидером сливаемой группы или нет. Чтобы удерживать минимальным общее число сообщений для слияния групп, лидером для новой группы может становиться лидер большей группы. Тем самым, уведомляться о смене лидера будут лишь члены меньшей группы.

Аналогично ранее рассмотренным алгоритмам, этот алгоритм позволяет работать с множеством групп и иметь множество лидеров. Алгоритм приглашений допускает создание групп процессов и их слияние без необходимости запуска новых выборов с нуля, снижая общее число сообщений, требующихся для завершения необходимых выборов.

Алгоритм кольца. Алгоритм основан на использовании кольца (физического или логического), но без маркера.

В алгоритме кольца все узлы рассматриваемой системы формируют некое кольцо и осведомлены об установленной топологии кольца (то есть о своих предшественниках и последователях в этом кольце). Когда определенный процесс замечает отказ имевшегося лидера, он запускает соответствующие новые выборы. Такое сообщение о выборах пускается далее по этому кольцу: каждый процесс вступает в контакт со своим последователем (следующим ближайшим к нему в кольце узлом). Когда такой узел недоступен, процесс пропускает недостижимый узел и пытается вступить в контакт со следующим после него узлом в кольце, до тех пор, пока один из них не ответит.

Узлы контактируют со своими соседями, следуя по кругу и собирая набор рабочих узлов, добавляя себя в это множество перед тем, как передать его следующему узлу.

Данный алгоритм продолжается до полного обхода своего кольца. Когда сообщение приходит обратно к тому узлу, который запустил эти выборы, в качестве лидера выбирается узел с самым высоким рангом в данном наборе работающих.

На рисунке 7 представлен пример такого обхода.

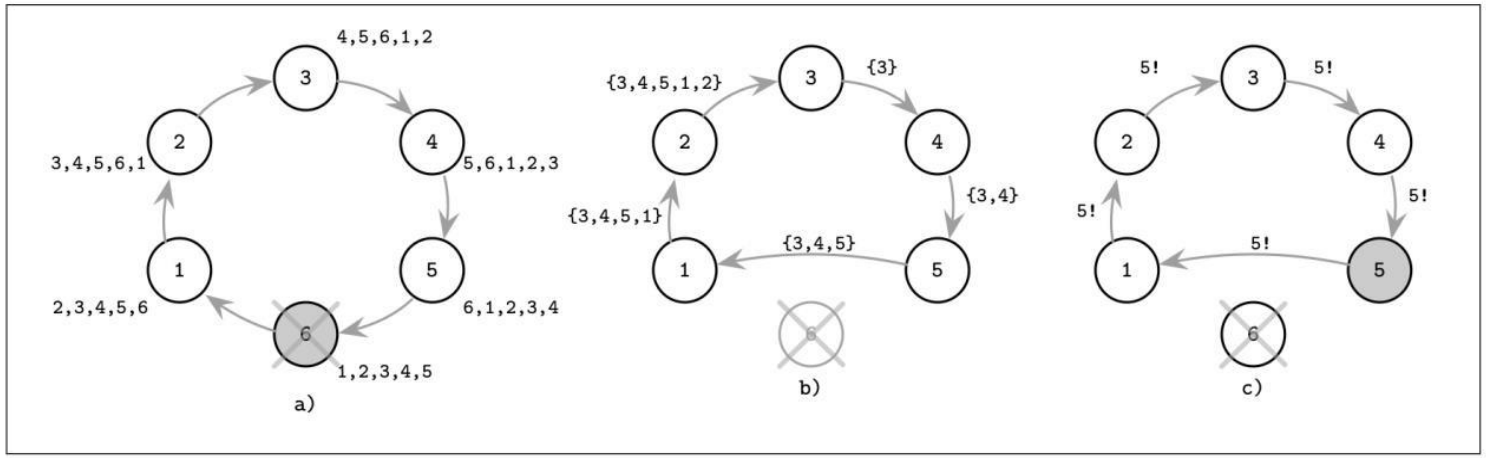

Рисунок 7 – Алгоритм кольца: предыдущий лидер (процесс 6) отказал и процесс 3 запустил новые выборы

1) Предыдущий лидер – процесс 6 отказал, и все процессы обладают неким представлением данного кольца со своей точки зрения.

2) Процесс 3 запускает цикл выбора. На каждом этапе имеется некий набор обойденных узлов по установленному пути к этому моменту. Процесс 5 не способен достичь узла 6, поэтому он пропускает его и переходит сразу к узлу 1.

3) Поскольку именно 5 был узлом с наивысшим значением ранга, процесс 3 инициирует следующий цикл обмена сообщениями, распространяя сведения о новом избранном лидере.

Вариации этого алгоритма содержат отбор отдельного идентификатора с наивысшим рангом вместо набора активных узлов с целью сохранения пространства: так как функция **max** коммуникативна, достаточно обладать текущим максимальным значением. Когда данный алгоритм возвращается обратно к узлу, запустившему данные выборы, значение наивысшего известного идентификатора вновь запускается по этому кругу.

Так как данное кольцо может быть разбито на две или более части, с возможностью потенциального выбора такой частью своего собственного лидера данный подход также не сохраняет свойство безопасности.

Как можно видеть, для правильной работы систем с лидером необходимо знать состояние текущего лидера (жив он или нет), при этом лидер должен быть рабочим и достижимым, чтобы исполнять свои обязанности. Для выявления отказа лидера можно применять алгоритмы выявления отказов.

Выбор лидера – это важная тема распределенных систем, так как использование назначенного лидера способствует снижению затрат ресурсов на координацию и улучшает производительность применяемого алгоритма. Циклы выборов могут быть ресурсозатратными, однако, поскольку они не часты, они не оказывают отрицательного воздействия на общую производительность системы. Некий отдельный лидер способен становиться узким местом, но в большинстве случаев это решается разбиением данных на части и применением лидеров для каждой части или использованием различных лидеров для отличающихся действий.

К сожалению, все обсуждавшиеся алгоритмы склонны к проблеме расщепления сознания: мы можем приходить к двум лидерам в двух независимых подсетях, которые даже и не догадываются о существовании друг друга. Во избежание расщепления сознания нам приходится получать большинство голосов всего кластера.

Многие алгоритмы консенсуса полагаются на некого лидера для координации. Тем не менее, являются ли выборы лидера тем же самым что и консенсус? Для выбора лидера нам требуется достичь согласия о его идентичности. Если мы способны достичь консенсуса относительно идентичности лидера, мы можем применять то же самое средство для достижения согласия по чему угодно.

Значение идентичности лидера может изменяться без знания этого процессами, потому будет справедлив также и вопрос о том знает ли о своем лидере локальный процесс.

К примеру, алгоритм ***стабильного выбора лидера*** (stable leader election) использует циклы с уникальным стабильным лидером и выявление отказов на основании таймаутов для обеспечения того, что этот лидер может оставаться в своем положении пока он не испытает отказа и является доступным.

Полагающиеся на выборы лидера алгоритмы часто *допускают* наличие присутствия множества лидеров и пытаются разрешать конфликты между этими лидерами настолько быстро, насколько это возможно.

Присутствие лидера – это некий способ гарантии *живучести* (когда отказывает текущий лидер, нам требуется новый), а процессам не требуется неопределенно длительного времени для того, чтобы понять отказали ли они на самом деле. Отсутствие *безопасности* и допуск множества лидеров – это некая оптимизация производительности: алгоритмы способны продолжать фазу репликации, а *безопасность* гарантируется выявлением и разрешением конфликтов.

3. Консенсус

Алгоритмы консенсуса в распределенных системах делают возможным для множества процессов достигать согласия по некому значению. Невозможно обеспечить согласие в некой целиком асинхронной системе за ограниченное время. Даже когда доставка сообщения гарантирована, одному процессу невозможно узнать испытал ли другой отказ, либо то, что он медленно работает.

Существует компромисс между точностью выявления отказов и тем, насколько быстро такой отказ может быть определен. Алгоритмы консенсуса полагаются на модель асинхронности и гарантию безопасности, в то время как некий определитель внешних отказов может предоставлять сведения относительно прочих процессов, гарантируя жизнеспособности. Так как выявления отказов не всегда целиком точны, будут иметься ситуации, при которых алгоритмы консенсуса ожидают выявления некого отказа процесса, либо когда данный алгоритм перезапускается потому как некоторые процессы неверно подозреваются в отказе.

Процессы должны прийти к согласию по некоему представляемому одним из участников значению, даже когда некоторые из них испытали отказ. Процесс называется ***корректным***, когда он не испытал отказ и продолжает исполнение шагов алгоритма. Консенсус чрезвычайно полезен для помещения событий в определенном порядке и обеспечивает согласованность между имеющимися участниками. Применяя консенсус, мы можем обладать системой, в которой процессы перемещаются от одного к следующему без утраты уверенности относительно того, какие значения наблюдаются их клиентами.

С теоретической точки зрения, алгоритмы консенсуса обладают тремя свойствами:

Согласованность – значение данного решения одно и то же для всех *корректных* процессов.

Обоснованность – принятое значение было предложено одним из имеющихся процессов.

Завершаемость – все *корректные* процессы в конечном итоге достигают определенного решения.

Всякое из этих свойств чрезвычайно важно. Само согласие встроено в человеческое понимание консенсуса. Словарное определение консенсуса подразумевает «единодушие». Это означает что в соответствии с таким согласием никакому процессу не дозволяется иметь мнение, отличное от выдаваемого на выходе.

Обоснованность существенна, так как без нее консенсус может быть тривиальным. Алгоритмы консенсуса требуют всем процессам прийти к согласию по некоему значению.

Когда процессы используют какое-то заранее определенное, произвольное установленное по умолчанию значение в качестве выводимого решения независимо от предлагаемых значений, они достигнут единогласия, но вывод такого алгоритма не будет правомерным и не будет полезно в реальности.

Без завершаемости алгоритм будет продолжаться неограниченно долго без достижения какого бы то ни было согласия или будет дожидаться неопределенно долго отказа процесса, что ни в одном из случаев не является полезным. Процессы в конце концов обязаны достигать согласия и, чтобы некий алгоритм согласия был практичным, это должно происходить достаточно быстро.

Широковещание. ***Широковещание*** – это абстракция взаимодействия, часто применяемая в распределенных системах. Алгоритмы широковещания используются для рассеивания сведений по некому набору процессов. Существует много алгоритмов широковещания, делающих различные предположения и предоставляющих различные гарантии. Широковещание является неким важным примитивом и применяется во множестве мест, включая и алгоритмы консенсуса.

Средства широковещания часто используются для репликации базы данных, когда некий отдельный узел координатора должен распространять необходимые данные всем прочим участникам. Однако, превращение этого процесса в надежное не тривиальное средство: когда такой координатор испытывает крушение после распространения данного сообщения в некоторые узлы, но не к части прочих, это оставляет данную систему в каком-то несогласованном состоянии: часть из имеющихся узлов уже видит некое новое сообщение, а прочие нет.

Самый простой и наиболее прямолинейный способ для широковещательного распространения сообщений состоит в ***широкое вещание с максимальными усилиями***. В этом случае, сам отправитель отвечает за обеспечение доставки событий всем имеющимся целям. В случае его отказа все прочие участники не пытаются повторно широковещательно распространять это сообщение и в случае крушения координатора, этот вид широковещания тихо откажет.

Для *надежности* широкого вещания необходимо обеспечивать чтобы все Корректные процессы получили одни и те же сообщения, даже если сам отправитель испытывает отказ в процессе передачи.

Для реализации простой версии надежного широковещания, мы можем пользоваться определителем отказов и механизмом обхода. Наиболее прямолинейный механизм обхода состоит в разрешении каждому принимающему данное сообщение процессу переправления его всем прочим процессам, о которых он осведомлен. Когда соответствующий процесс источника отказывает, прочие процессы выявляют это падение и продолжают широковещательно распространять это сообщение, действенно заполняя данную сетевую среду $\mathbf{N^2}$ сообщениями (как это показано на рисунке 8). Даже если такой отправитель претерпел крушение, сообщения все еще подбираются и доставляются всей оставшейся системой, улучшая ее надежность и позволяя всем получателям видеть те же самые сообщения.

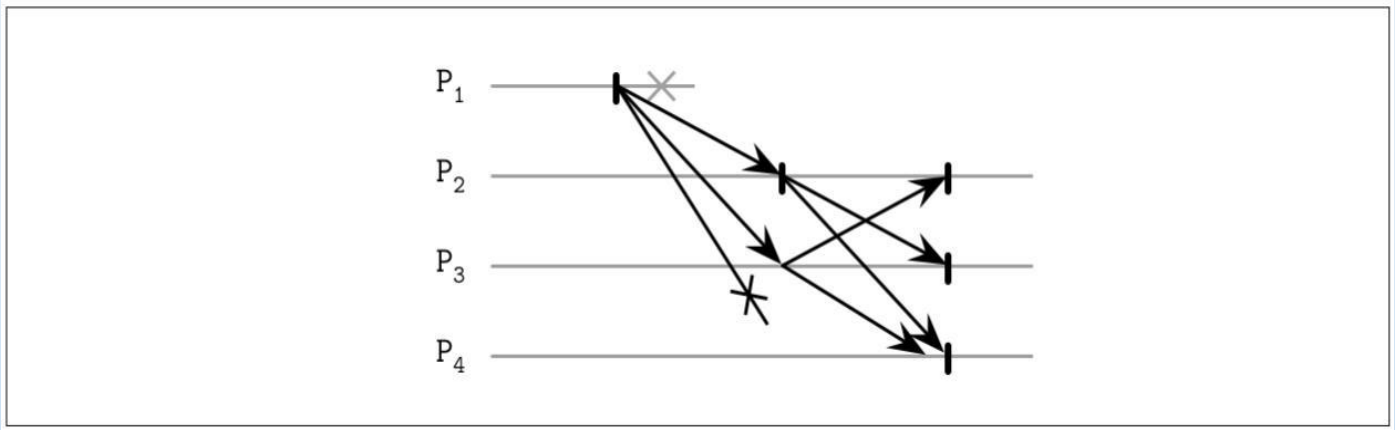

Рисунок 8 – Широковещание

Одной из обратных сторон данного подхода является тот факт, что он применяет $\mathbf{N^2}$ сообщений, где **N** это общее число остающихся получателей (так как каждый процесс широковещания исключает сам первоначальный процесс и его самого). В идеале мы бы желали снизить общее число сообщений, необходимых для надежного широковещания.

Атомарное широковещание. Даже несмотря на то, что рассмотренный алгоритм может гарантировать доставку, он не обеспечивает доставку в каком бы то ни было определенном порядке. Сообщения достигают в конце концов своего получателя, причем в некий неизвестный момент времени. Когда нам необходимо доставлять сообщения по порядку, нам приходится применять ***атомарное широковещание*** (также носящего название ***тотально упорядоченной групповой передачи***), которая гарантирует и надежную доставку, и тотальное упорядочивание.

В то время как надежное широковещание гарантирует что процессы согласовывают необходимый набор доставки сообщений, некое атомарное широковещание также обеспечивает что они согласовывают одну и ту же последовательность сообщений (то есть что порядок доставки сообщений одинаков для всех целей).

В сумме некое атомарное широковещание обязано гарантировать ***два существенных свойства:***

Атомарность. Процессы должны достичь согласия по установленному множеству полученных сообщений. Либо всем неотказавшим процессам доставлено данное сообщение, либо ни одному из них.

Упорядоченность. Всем не отказавшим процессам сообщения доставляются в одном и том же порядке.

Сообщения в данном случае доставляются *атомарно*: каждое сообщение либо доставляется всем процессам, либо ни одному из них, причем, когда такое сообщение доставлено, все прочие сообщения все прочие сообщения упорядочиваются до или после этого сообщения.

Алгоритм Paxos. Наиболее широко известным алгоритмам консенсуса является *Paxos*. Он впервые был предложен Лесли Лэмпортом в его статье «The Part-Time Parliament».

В этой статье консенсус описывается в выражениях терминологии, вдохновленной законодательным и избирательным процессом на Эгейском острове Паксос.

В 2001 году этот автор выпустил следующую статью, озаглавленную «Paxos Made Simple», которая вводила более простую терминологию, которая теперь и применяется для объяснения данного алгоритма.

Участники Paxos способны выполнять одну из трех ролей: *Заявителей*, *Получателей* или *Обучаемых*.

Заявители. Получают от клиентов значения, создают предложения по принятию этих значений и пытаются собрать голоса от Получателей.

Получатели. Голосуют за принятие или отклонение предложенного Заявителем значения. Для отказоустойчивости данный алгоритм требует наличия множества Получателей, однако для его жизнеспособности с целью принятия выставленного предложения требуется лишь кворум (большинство) голосов Получателей.

Обучаемые. Играют роль реплик, храня все выводимые сведения по принятым предложениям.

Всякий участник способен выполнять любую роль, причем большинство реализаций совмещают их: некий отдельный процесс может одновременно выступать в роли Заявителем, Получателя и Обучаемого.

Всякое предложение состоит из некого *значения*, предлагаемого соответствующим клиентом, а также монотонно возрастающим номером предложения. Этот номер затем используется для обеспечения общего порядка исполняемых операций и устанавливает взаимосвязи происхождения -до/-после между ними. Номера предложений также часто реализуются при помощи неких пар **(id, timestamp)**, где также сопоставляется и идентификатор узла, который может применяться в качестве разрывающих связей (break ties) для временных отметок.

Алгоритм. Сам алгоритм Paxos может быть в целом разделен на два этапа: фазу *голосования* (или *предложения*) и *репликацию*.

На протяжении этапа голосования, Вносящие предложение состязаются за установление своего лидерства. В процессе репликации, установленный Вносящий предложение распространяет соответствующее значение Получателям.

Вносящий предложение является некой точкой инициализации контакта с клиентом. Он получает некое значение, по которому следует принять решение, и пытается собрать голоса от своего кворума Получателей. Когда это выполнено, Получатели распространяют соответствующие сведения относительно данного согласованного значения Обучаемым, закрепляя (ратифицируя) полученный результат. Обучаемые увеличивают значение множителя репликаций данного значения, по которому было достигнуто согласие.

Только один Вносящий предложение способен собирать значение большинства голосов. При некоторых обстоятельствах голоса могут делиться поровну между имеющими Вносящими предложение и ни один из них не будет иметь возможности собирать большинство на протяжении данного цикла, принуждая его к повторному запуску.

На протяжении этапа предложения установленный *Вносящий предложение* отправляет некое сообщение **Prepare(n)** (где **n** это номер предложения) некому большинству Получателей и предпринимает попытку сбора их голосов.

Когда некий *Получатель* принимает этот подготовительный запрос, он обязан ответить, сохраняя следующие инварианты:

- Если этот Получатель еще не ответил на некий подготовительный запрос с каким- то более высоким последовательным номером, он *обещает*, что не будет принимать никакие предложения с меньшим последовательным номером.

- Если этот Получатель ранее уже принял (получил сообщение **Accept!(m,$v_{accepted}$)**) любое иное предложение, он отвечает сообщением **Promise(m,$v_{accepted}$)**, уведомляя своего Вносящего предложение, что он уже принял определенное предложение с порядковым номером **m**.

- Если этот Получатель уже откликнулся на некий подготовительный запрос с более высоким последовательным номером, он уведомляет своего Заявителя о присутствии предложения с более высоким номером.

- Получатель способен отвечать более чем на один подготовительный запрос, по мере того как самый последний из них имеет более высокий последовательный номер.

На протяжении соответствующего этапа репликации, после сбора голосов большинства, установленный Заявитель может запустить необходимую репликацию, в которой он фиксирует значение предложения, отправляя Получателям некое сообщение **Accept!(n, v)** со значением **v** и номером предложения **n**. **v** – это то значение, которое связано с имеющим наивысший номер из всех принятых им от Получателей откликов предложением или любое его собственное значение когда их отклики не содержат старых принятых предложений.

Соответствующий Получатель принимает это предложение с номером **n**, если только на протяжении этапа предложения он уже не ответил **Prepare(m)**, где **m** больше, чем **n**. Когда Получатель отвергает данное предложение, он уведомляет об этом своего Заявителя, отправляя самое большое последовательное число, которое он наблюдал в своем запросе содействуя своему Заявителю поймать его.

Можно рассмотреть обобщенное отображение некого цикла Paxos на рисунке 9.

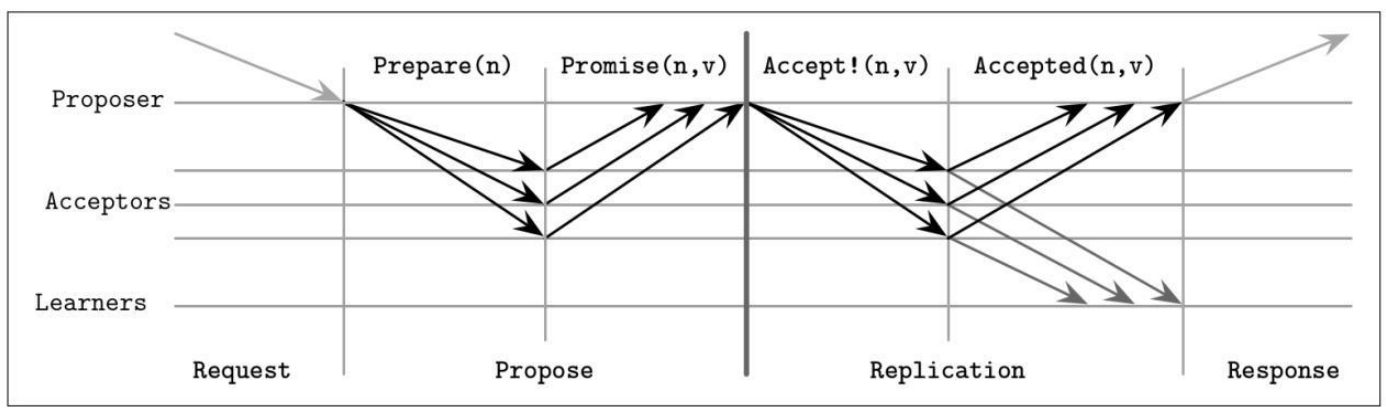

Рисунок 9 – Алгоритм консенсуса Paxos

После достижения консенсуса по обсуждаемому значению (иначе говоря, оно было принято по крайней мере одним Получателем), последующим Заявителям приходится принимать решение о том же самом значении для обеспечения необходимого согласия. Именно поэтому Получатели отвечают самым последним принятым ими значением. Когда ни один из Получателей не видел какого-то предыдущего значения, имеющийся Заявитель волен выбрать свое собственное значение.

Обучаемый обязан отыскать то значение, по которому было принято решение, с которым он может ознакомиться после получения уведомлений от большинства Получателей. Чтобы позволить имеющемуся Обучаемому узнать о новом установленном значении настолько быстро, насколько это возможно, Получатели могут уведомлять их о таком значении, как только они его приняли. Если имеются более одного Обучаемых, каждому Получателю придется уведомлять всех Обучаемых. Один или более обучаемых могут быть *Выдающимися* (distinguished), и в этом случае он будет уведомлять и прочих Обучаемых относительно принимаемых значений.

Итак, основная цель самого первого этапа алгоритма состоит в установлении некого лидера для данного раунда и понимания какое значение подлежит принятию, что делает возможным такому лидеру продолжить второй этап: широковещательное распространение этого значения. На практике, мы бы желали снизить общее число этапов в данном алгоритме, потому мы позволяем своему Заявителю предлагать более одного значения.

Кворум. Кворумы применяются для обретения уверенности в том, что несмотря на способность некоторых из имеющихся участников к отказу, мы все еще способны продолжать пока мы можем собирать голоса от прочих находящихся в работе. *Кворум* – это значение минимального числа голосов, необходимого для осуществления заданного действия. Это число обычно составляет некое большинство участников. Самая главная стоящая за кворумом мысль состоит в том, что даже когда участники отказывают, или возникает разбиение на сетевые разделы, существует по крайней мере один участник, выступающий неким арбитром, обеспечивающим корректность протокола.

После того как достаточное число участников приняло данное предложение, протокол гарантирует принятие данного значения, так как любые два большинства обладают по крайней мере одним общим участником.

Paxos обеспечивает безопасность при наличии любого числа отказов. Не существует конфигураций, которые способны производить некорректные или не состоятельные состояния, так как это противоречило бы самому понятию консенсуса.

Жизнеспособность гарантируется при наличии **f** отказавших процессов. Для этого данный протокол требует в общей сложности **2f + 1** процессов с тем, чтобы, когда произойдет **f** отказов, все еще будет иметься **f + 1** процессов, способных продолжать работу. Применяя кворумы, вместо того чтобы требовать присутствия всех процессов, Paxos обеспечивает результаты даже в случае возникновения **f** отказов.

Сценарии отказов. Обсуждение распределенных систем становится особенно занимательным при рассмотрении отказов. Один из возможных сценариев отказа, демонстрирующих отказоустойчивость, состоит в том, когда на втором этапе отказывает Заявитель, причем до того как он получил возможность широковещательно распространять необходимое значение всем имеющимся Получателям (некая подобная ситуация может произойти если данный Заявитель в рабочем состоянии, но медленный или не способен взаимодействовать с некоторыми Получателями). В этом случае избираемый новый Заявитель может подхватить и зафиксировать данное значение, распространяя его прочим участникам.

Рисунок 10 отображает такую ситуацию:

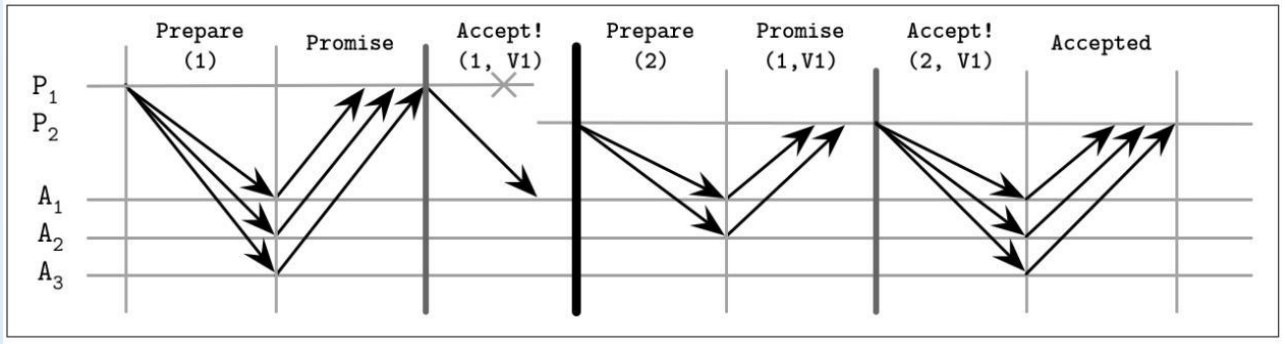

Рисунок 10 – Сценарий отказа Paxos: Вносящий предложение отказывает, полагаясь на свое старое значение

- Заявитель $\mathbf{P_1}$ проходит этап выборов с неким предложением номер **1**, однако отказывает после отправки соответствующего значения **V1** всего лишь одному получателю $\mathbf{A_1}$.

- Другой Заявитель $\mathbf{P_2}$ запускает новый цикл с более высоким номером предложения **2**, собирает некий кворум из откликов Получателей (в данном случае $\mathbf{A_1}$ и $\mathbf{A_2}$) и продолжает фиксацию своего старого значения **V1**, предложенного $\mathbf{P_1}$.

Так как значение состояния данного алгоритма реплицируется множеству узлов, отказ Заявителя не приводит в результате к отказу в достижении консенсуса.

Если установленный Заявитель отказывает даже после приема устанавливаемого значения единственным Получателем $\mathbf{A_1}$, его предложение может быть подхвачено следующим установленным Заявителем. Это также может подразумевать, что все это может происходить и без того, чтобы об этом узнал сам первоначальный Заявитель.

В приложении клиент/сервер, в котором определенный клиент соединяется лишь с первоначальным Заявителем, это может повлечь за собой ситуации, при которых данный клиент не знает об окончательном результате исполнения данного цикла Paxos.

Тем не менее, также возможны и иные сценарии, как это отображено на рисунке 11. Например:

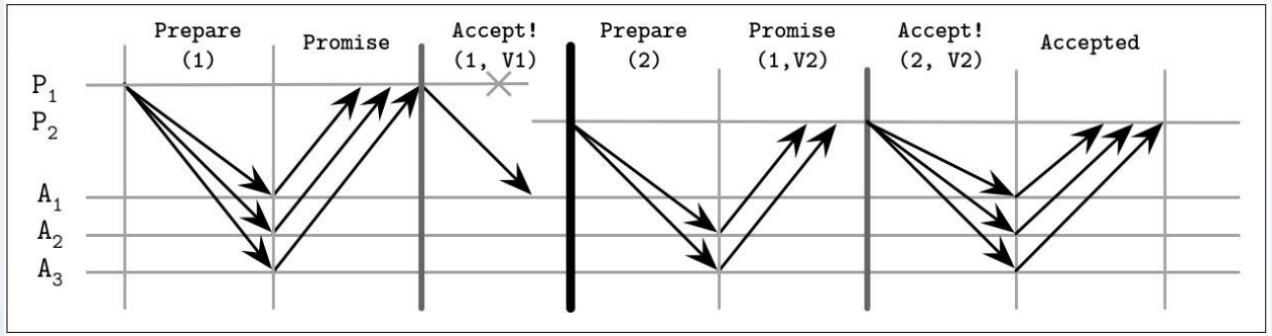

Рисунок 11 – Сценарий отказа Paxos: Вносящий предложение отказывает, полагаясь на свое новое значение

- $\mathbf{P_1}$ отказывает в точности так же, как и в предыдущем примере, после отправки своего значения **V1** лишь к $\mathbf{A_1}$.

- Наш следующий Заявитель, $\mathbf{P_2}$ запускает некий новый раунд с более высоким номером предложения **2** и собирает некий кворум откликов Получателей, однако на этот раз первыми откликаются $\mathbf{A_2}$ и $\mathbf{A_3}$. После сбора кворума, $\mathbf{P_2}$ фиксирует свое собственное значение, несмотря на тот факт, что теоретически присутствует некое иное фиксируемое значение в $\mathbf{A_1}$.

В данных обстоятельствах имеется еще одна возможность, отображаемая на рисунке 12:

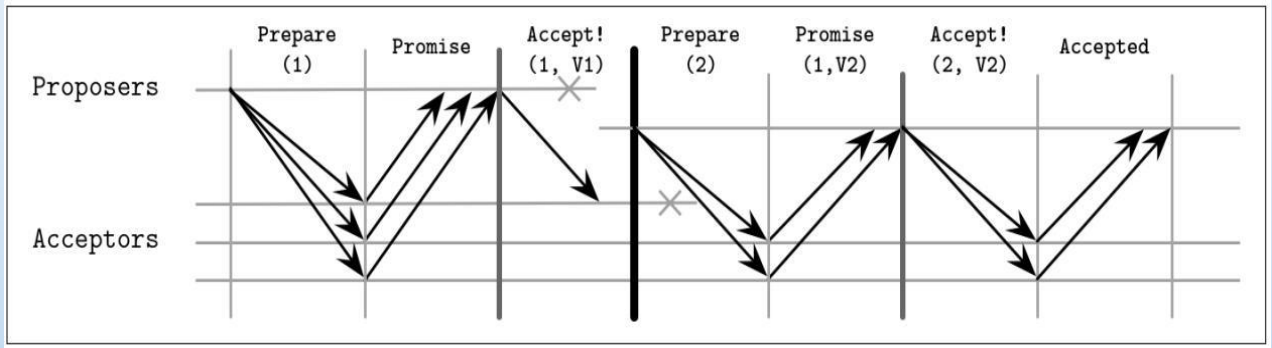

Рисунок 12 – Сценарий отказа Paxos: Вносящий предложение отказывает вслед за отказом своего Получателя

- Заявитель $\mathbf{P_1}$ отказывает лишь после единственного Получателя $\mathbf{A_1}$, принявшего выставленное значение **V1**. $\mathbf{A_1}$ вскорости после приема данного предложения отказывает, причем до того, как он смог оповестить своего следующего Заявителя о своем значении.

- Заявитель $\mathbf{P_2}$, который запустил необходимый новый раунд после отказа $\mathbf{P_1}$ не перекрывался с $\mathbf{A_1}$ и продолжает вместо этого фиксацией своего значения.

- Всякий Заявитель, который вступает после такого раунда и который будет перекрываться с $\mathbf{A_1}$, будет игнорировать значение $\mathbf{A_1}$ и выбирать вместо него более позднее принимаемое предложение.

Иной сценарий отказа происходит, когда два или более Заявителя начинают состязание, причем каждый пытается пройти через свой этап предложения, но не способен собрать большинство, и побеждает соперник.

Хотя Получатели и обещают не принимать никакие предложения с более низкими номерами, они все еще отвечают множеству подготовительных запросов до тех пор, пока не появится самый последний с неким более высоким последовательным номером. Когда некий Заявитель пытается фиксировать данное значение, он может обнаружить что Получатели уже ответили на некий подготовительный запрос с более высоким последовательным номером. Это может повлечь к тому, что множество процессов постоянно повторяют свою попытку и препятствуют друг другу в последующем развитии. Такая проблема обычно разрешается через вовлечение некого случайного отказа, который в конечном счете позволит одному из имеющихся Заявителей продолжать в то время, как другой засыпает.

Алгоритм Paxos может не обращать внимания на отказы Получателя, но только когда все еще имеется достаточно рабочих Получателей для формирования большинства.

Алгоритм Raft. Paxos был основным алгоритмом консенсуса на протяжении более десяти лет, однако в сообществе распределенных систем он был известен как трудный для обсуждения.

В 2013 году появился некий новый алгоритм с названием Raft (Плот). Разработавшие его исследователи хотели создать некий алгоритм, который просто понять и реализовать. Впервые он был представлен в статье с названием «В поисках понятного алгоритма консенсуса».

В распределенных системах и так имеется достаточно сложностей, а потому наличие более простых алгоритмов весьма желательно.

Локально участники хранят журнал, содержащий последовательность команд, выполняемых соответствующим конечным автоматом. Поскольку получаемые процессами входные данные идентичны, а журналы содержат одинаковые команды в одном и том же порядке, применение этих команд к соответствующему конечному автомату обеспечивает одинаковый вывод. Raft упрощает получение консенсуса, создавая понятие Лидирующего первоклассного гражданина. Лидер применяется для координации манипуляций и репликаций конечного автомата.

Имеется множество сходств между Raft и алгоритмами атомарного широковещания, а также Множественного Paxos: некий обособленный Лидер всплывает из репликаций, принимает атомарные решения и устанавливает порядок сообщений.

Каждый участник Raft может выполнять одну из трех ролей:

Кандидат. Лидерство является временным условием, и любой участник способен взять на себя эту роль. Чтобы стать Лидером этому узлу вначале придется перейти в состояние некого кандидата и попытаться набрать большинство голосов. Когда и не побеждает, и не проигрывает на выборах (полученные голоса распределяются между множеством кандидатов и ни один из них не имеет большинства голосов), выдвигается новый срок и выборы запускаются повторно.

Лидер. Некий текущий, временный Лидер кластера, который обрабатывает запросы клиента и взаимодействует с неким реплицируемым конечным автоматом. Этот лидер выбирается на период, именуемый ***сроком (term)***. Всякий срок идентифицируется монотонно приращиваемым числом и может продолжаться некий произвольный период времени. Новый Лидер выбирается в случае крушения текущего, когда он перестает отвечать, либо в других процессах имеется ожидание того, что он должен отказать, что может происходить по причине разбиения сетевой среды на разделы или задержек сообщений.

Последователь. Некий пассивный участник, который сохраняет регистрационные записи и откликается на запросы от установленного Лидера и Кандидатов. Последователь в Raft – это некая роль аналогичная Получателю *и* Обучаемому из Paxos. Все процессы начинают с Последователей (Follower).

Чтобы гарантировать глобальное частичное упорядочение без того, чтобы полагаться на синхронизацию часов, время подразделяется на ***сроки*** (также называемые ***эпохами***), на протяжении которых установленный Лидер является уникальным и стабильным. Сроки монотонно нумеруются и все команды уникально идентифицируются сроком и соответствующим номером сообщения внутри этого срока.

Может так получиться, что различные участники не пришли к согласию какой из сроков является *текущим*, поскольку они могли обнаружить соответствующий новый срок в разное время или могли пропустить выборы Лидера для одного из множества сроков. Так как всякое сообщение содержит некий идентификатор срока, когда один из участников обнаруживает, что его срок не актуален, он обновляет свой срок на имеющий более высокий номер. Это означает, что *могут* иметься несколько сроков в любой определенный момент времени, однако в случае некого конфликта побеждает тот, у которого более высокий номер. Некий узел обновляет свой срок только когда он запускает новый процесс выборов или обнаруживает что его срок не актуален.

При запуске, или когда Последователь не получает сообщения от своего Лидера и полагает что он потерпел крушение, он запускает процесс выборов необходимого Лидера. Некий участник пробует стать Лидером переходя в состояние Кандидата и собирая голоса от большинства узлов.

Рисунок 13 показывает диаграмму последовательности, представляющую все основные компоненты обсуждаемого алгоритма Raft:

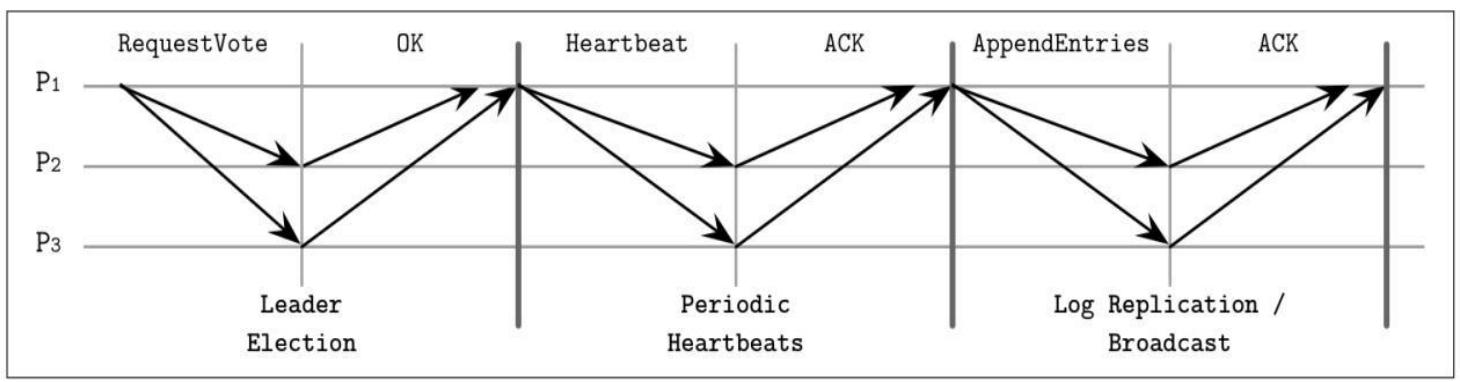

Рисунок 13 – Алгоритм консенсуса Raft

Выбор Лидера. Кандидат $\mathbf{P_1}$ отправляет сообщение **RequestVote** всем прочим процессам. Это сообщение содержит значение срока кандидата, самый последний известный ему срок и значение идентификатора самой последней наблюдавшейся им регистрационной записи. После сбора большинства голосов этот Кандидат успешно избирается в качестве Лидера на данный срок. Все процессы отдают свой голос не более чем за одного кандидата.

Периодические сердцебиения. Данный протокол пользуется механизмом сердцебиений для обеспечения необходимой жизнеспособности участников. Сам Лидер периодически отправляет сердцебиения всем Последователям для сопровождения своего срока. Когда Последователь не получает новых сердцебиений на протяжении некого периода, носящего название ***таймаута выборов***, он предполагает, что его Лидер отказал и запускает новые выборы.

Репликация журнала/широковещание. Установленный Лидер может неоднократно добавлять новые значения в конец своего реплицируемого журнала отправляя сообщения **AppendEntries**. Это сообщение содержит значение срока Лидера, индекс и срок той регистрационной записи, которая непосредственно предшествовала той, которую он отправляет в настоящий момент, а также одну или более записей для сохранения.

Роль лидера в алгоритме Raft. Некий Лидер может быть выбран только из тех узлов, которые хранят все фиксированные записи: если в процессе данных выборов сведения соответствующего Последователя более актуальны (иначе говоря, имеют более высокий идентификатор срока или более длинную последовательность зарегистрированных записей, когда сроки равны) чем у рассматриваемого Кандидата, его голос отрицательный.

Чтобы победить на выборах, некий Кандидат должен собрать большинство голосов. Записи всегда реплицируются по порядку, поэтому чтобы понять является или нет один из участников актуальным, всегда достаточно сравнивать идентификаторы с самыми последними записями.

Будучи избранным, этот Лидер обязан принимать запросы клиентов (которые также могут передаваться ему с прочих узлов) и реплицировать их своим Последователям. Это осуществляется добавлением в конец его журнала соответствующей записи и ее параллельной отправки всем Последователям.

Когда Последователи получают некое сообщение **AppendEntries**, они добавляют все записи из этого сообщения в конец своего локального журнала, давая своему Лидеру знать, что они приняли их на постоянное хранение. Как только достаточное число реплик отправило свои подтверждения, данная запись рассматривается фиксированной и соответствующим образом помечается в журнале установленного Лидера.

Так как Лидером может становиться лишь самый актуальный Кандидат, Последователям никогда не приходится приводить своего Лидера в актуального и регистрационный записи перетекают только от Лидера к Последователю, а не наоборот.

Рисунок 14 показывает данный процесс:

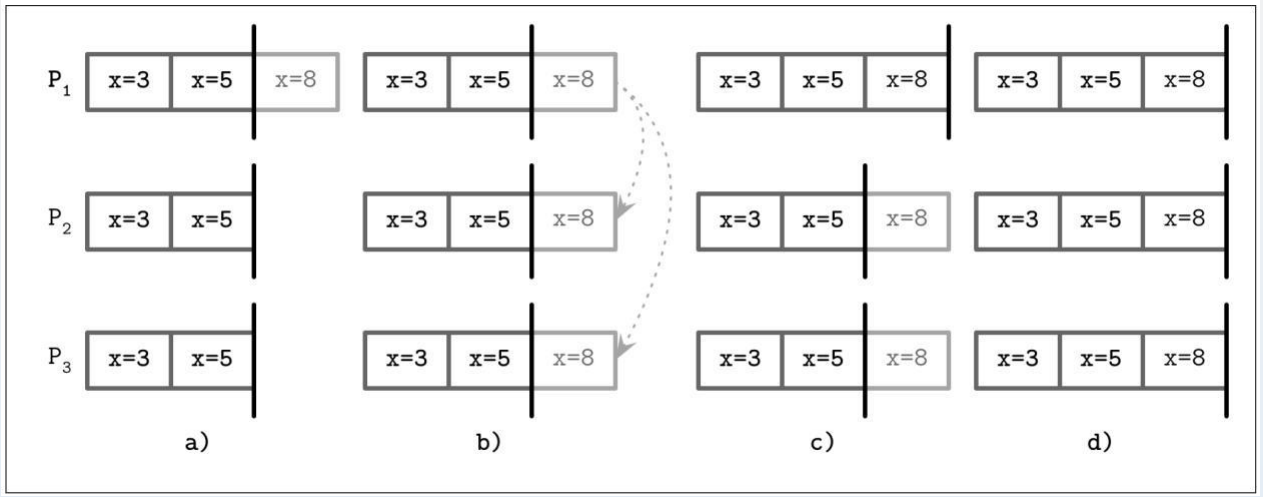

Рисунок 14 – Процедура фиксации в Raft с P_1 в качестве лидера

- В конец журнала установленного Лидера добавляется новая команда **x = 8**.

- Прежде чем это значение может быть фиксировано, его следует реплицировать в большинство участников.

- Как только установленный Лидер покончит с репликациями, он фиксирует это значение локально.

- Решение о данной фиксации реплицируется всем Последователям.

Рисунок 15 показывает некий образец раунда консенсуса, в котором **P_1** выступает имеющим наиболее последнее представление событий Лидером.

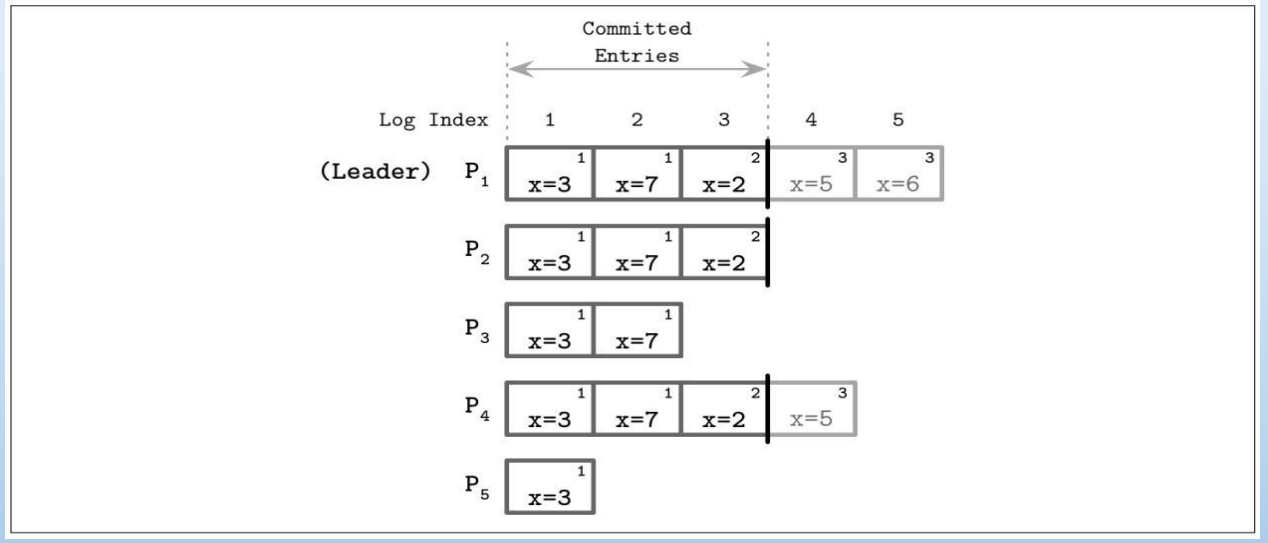

Рисунок 15 – Конечный автомат состояний Raft

Этот Лидер продолжает реплицирование всех записей своим Последователям и фиксирует их после сбора подтверждений. Фиксация некой записи также фиксирует все предшествующие ей записи в соответствующем журнале. Только установленный Лидер способен принимать решение по вопросу будет или нет фиксирована данная запись. Все регистрируемые записи помечаются идентификатором срока (числом в правом верхнем углу каждого блока регистрационной записи) и неким индексом регистрации, указывающим его положение в данном журнале. Фиксированные записи гарантированно реплицируются в установленный кворум участников и безопасно применяются к установленному конечному автомату именно в том порядке, в котором они возникают в соответствующем журнале.

Сценарии отказов. Когда множество Последователей принимают решение выступить Кандидатами и никакой из Кандидатов не набирает большинства голосов, такая ситуация носит название ***разделения голосов (split vote)***. Для снижения значения вероятности множества последовательных выборов, заканчивающихся разделением голосов, Raft применяет выставляемые случайным образом флажки часов. Один из имеющихся Кандидатов может запустить свои следующие выборы раньше и набрать достаточное число голосов, в то время как прочие проспят и предоставят дорогу ему. Такой подход ускоряет подобные выборы без необходимости дополнительно координации между кандидатами.

Последователи могут останавливаться или замедляться с откликом, и установленному Лидеру придется предпринять все от него зависящее чтобы обеспечить доставку сообщения. Он может попробовать отправку сообщения снова, когда не получает некого подтверждения за ожидаемый по времени срок. В качестве оптимизации производительности, он может отправлять параллельно множество сообщений.

Так как реплицируемые установленным Лидером сообщения идентифицируются уникально, повторная доставка сообщений гарантированно не нарушает порядок соответствующего журнала. Последователи выполняют дедупликацию сообщений пользуясь их последовательными идентификаторами и обеспечивая отсутствие сторонних эффектов со стороны дублированных доставок.

Последовательные идентификаторы также применяются для обеспечения необходимого упорядочения в журнале. Некий Последователь отвергает запись с более высоким номером, когда значение идентификатора и срока той отправленной установленным Лидером записи, которая непосредственно ей предшествовала, не соответствует значению наивысшей записи в его собственных записях. Когда записи в двух журналах различных реплик обладают одним и тем же сроком и одним и тем же индексом, они хранят те же самые команды и все те же самые предшествующие им записи.

Алгоритм Raft гарантирует, что никогда не будет отображаться никакое не зафиксированное сообщение в качестве фиксированного, но, по причине замедления сетевой среды или репликации, уже зафиксированные сообщения могут все еще рассматриваться как пребывающие *в процессе выполнения*, что является достаточно безопасным качеством, причем его можно обойти, повторяя команду клиента до ее окончательной фиксации.

Для выявления отказов установленный Лидер обязан отправлять своим Последователям сердцебиения. Таким образом установленный Лидер поддерживает свой срок. Когда один из узлов замечает, что его текущий Лидер остановился, он пытается инициировать новые выборы. Вновь установленному Лидеру приходится восстанавливать значение состояния своего кластера до самой последней известной актуальной записи журнала. Это осуществляется через поиск *общей земли* (common ground, самой наивысшей записи, по которой имеют согласие и этот Лидер, и Последователь) и ориентирует Последователей *отвергать* все (не зафиксированные) записи, добавляемые в конец с этого момента. Затем он отправляет самые последние записи из своего журнала, перезаписывая историю своих Последователей. Собственные записи журнала установленного Лидера никогда не удаляются и не перезаписываются: он имеет возможность только добавлять записи в конец своего журнала.

Таким образом ***алгоритм Raft предоставляет следующие гарантии:***

- За раз для заданного срока может выбираться лишь один Лидер; никакие два Лидера не могут быть активными на протяжении одного и того же срока.

- Установленный лидер не имеет возможности удалять содержимое в своем журнале или изменять его порядок; он способен только добавлять в его конец новые записи.

- Фиксированные записи регистрации гарантированно присутствуют в журналах для последующих Лидеров и не могут отыгрываться обратно, так как до того, как данная запись зафиксирована, известно что она реплицирована установленным Лидером.

- Все сообщения уникально идентифицируются идентификаторами сообщения и срока; ни текущие, ни последующие Лидеры не могут повторно применять один и тот же идентификатор для иной записи.

С момента своего появления алгоритм Raft превратился в очень популярный и в настоящее время применяется во множестве баз данных и прочих распределенных систем, включая CockroachDB, Etcd и Consul. Это можно связать с его простотой, но также может означать и то, что Raft оправдал ожидания стал надежным алгоритмом консенсуса.

Выводы

В ходе лекции рассмотрены следующие вопросы:

- синхронизация времени;

- выбор лидера (координатора);

- консенсус.

Задание на самостоятельную работу

1. Конспект лекций.

Вид и тема следующего занятия

Практическое занятие №6. Основы в ASP.NET Core (ч. 6)