Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное
учреждение высшего профессионального образования
«Московский физико-технический институт (государственный университет)»

Факультет управления и прикладной математики
Кафедра теоретической и прикладной информатики

Исследование алгоритмов принятия консенсуса в сети ненадежных вычислителей

Выпускная квалификационная работа
(магистерская диссертация)

Направление подготовки: 01.09.56
“Математические и информационные технологии”

Выполнил: студент 973 группы
Ланцов Александр Валерьевич

Научный руководитель:
д.ф.-м.н., профессор
Тормасов Александр Геннадьевич

Научный консультант:
Непорада Андрей Леонидович

Москва — 2015

Содержание

1. Введение

Эта работа описывает, анализирует, уточняет и сравнивает алгоритмы принятия консенсуса *Raft* и *Paxos*, которые служат для построения отказоустойчивых приложений на кластере из нескольких узлов, используя концепцию распределенного конечного автомата.

Алгоритм принятия консенсуса позволяет строить надежные системы из ненадежных компонентов, поэтому является основой многих распределенных систем. Эта актуальная тема находит свое применение в отказоустойчивых базах данных, групповых системах, и других системах. Настоящая работа сравнивает реализации алгоритмов *Raft* и *Paxos* в различных ситуациях, как отказ простых узлов, отказ лидера; сравнивает их производительность.

Алгоритм Лесли Лампорта *Paxos*[11] является одним из главных исследуемых алгоритмов решения задачи консенсуса. Несмотря на это, его трудно понять и реализовать, потому что он построен на неинтуитивном подходе. Другой проблемой является то, что отсутствует полное четкое описание алгоритма *Multi-Paxos*[16].

Алгоритм *Raft*[18], созданный Диего Онгаро и Джоном Аустерхортом, является алгоритмом решения задачи консенсуса, который, будучи основан на тех же предположениях, обеспечивает те же гарантии, что и *Multi-Paxos*. Его плюсом является относительная простота, интуитивность и доказанная безопасность.

2. Постановка задачи

Как уже упоминалось, задача принятия консенсуса является одной из основополагающих для современных распределенных компьютерных систем. Задача достижения консенсуса - это задача получения согласованного значения группой участников, члены которой называются *узлами*, в ситуации, когда возможны отказы отдельных участников, предоставления им некорректной информации или искажения переданных значений средой передачи данных.

Главными задачами этой работы было исследовать скорость работы алгоритмов Raft и Multi-Paxos, скорость восстановления после различных ошибок относительно друг друга. Измерения производились при разном числе узлов в кластере.

2.1 Описание задачи принятия консенсуса

Консенсус должен удовлетворять следующим условиям:

1. **Согласованность**: все корректные работающие узлы принимают одно и тоже значение (*"свойство безопасности"*);

2. **Корректность**: выбранное значение должно быть одним из тех, которое было предложено каким-либо корректно работающим узлом (*"свойство нетривиальности"*);

3. **Конечность**: каждый корректно работающий узел должен делать выбор за конечное количество шагов (*"свойство завершенности"*).

Теорема Фишера, Линча и Патерсона гласит, что не существует асинхронного детерминированного алгоритма принятия консенсуса, который был бы устойчив к выходу из строя одного узла и гарантировал бы все свойства консенсуса[5]. Асинхронизм означает, что нельзя достоверно различить работает ли узел медленно, или долго идет сообщение, или же отказал узел, даже если предположить, что связь является надежной. Алгоритмы *Raft* и *Paxos* гарантируют только свойство безопасности и нетривиальности, но не всегда удовлетворяют свойству завершенности.

2.2 Возможные типы ошибок

Стандартная задача консенсуса требует, чтобы в асинхронной системе было выполнено свойство безопасности и свойство нетривиальности, вне зависимости от количества невизантийских ошибок, и чтобы все три свойства были выполнены при числе невизантийских ошибок меньше, чем пороговое значение. Число корректно работающих узлов, при которых алгоритм работает корректно, называется *размером кворума.* Для *Multi-Paxos* и *Raft* допустимо не более F ошибок при размере кворума $F + 1$ для кластера из $2F + 1$ узлов. Очевидно, что два любых кворума (множества из всех корректно работающих узлов) для кластера из $2F + 1$ узлов будут пересекаться по крайней мере в одном узле. Таким образом, если есть кворум, то по крайней мере один узел любого будущего кворума будет принадлежать текущему кворуму, а значит, будет знать все его операции.

В контексте задачи консенсуса существуют разные типы ошибок:

1. **Остановка с сигналом**: узел прекращает работу, а затем сообщает другим узлам о своем сбое. Этот тип ошибки позволяет точно определить, отказал ли узел или нет; такие ошибки предоставляют самые удобные для реализации предположения относительно сбоев в системе;

2. **Остановка**: узел преждевременно прекращает какую-либо активность, несмотря на то, что до остановки он работал полностью корректно. После остановки этот узел приостанавливае его работу. Предполагается, что узел может восстанавливается после остановки и включается в работу кластера;

3. **Потери сообщений**: при отсылке и при отправлении сообщений могут происходить потери сообщений, противоположная сторона никак не может узнать, что сообщение, предназначенное для неё, было потеряно;

4. **Византийския ошибка**: наиболее серьезный тип ошибки, процесс начинает себя вести некорректно. Например, он может посылать лишние или противоречивые сообщения, не работать какое-то время и так далее. Данный тип ошибок в работе не рассматривается, так как Raft вообще не приспособлен для работы с наличием византийских ошибок, а Paxos требует значительных изменений[7].

2.3 CAP-теорема

CAP-теорема была сформулирована Эриком Брюэром в 2000 году, а через два года официально доказана. CAP-теорема утверждает, что можно создать распределенную систему, которая будет удовлетворять свой-

ству согласованности (то есть её операции записи атомарны, все последующие операции чтения видят новое значение), свойству доступности (система работает, пока работает хотя бы один участник), и устойчивой к потери связности (система продолжает работу, даже если связи между серверами временно отсутствуют), но одновременно можно гарантировать только два из этих трех свойств[6].

Если два узла получают два противоречивые запроса от клиентов, они оба должны принять и обработать запросы, таким образом, под ставя под угрозу свойство согласованности или, по крайней мере один из них не должен принимать запрос, таким образом, ставя под угрозу свойство доступности.

Как будет видно далее, алгоритмы Raft и Paxos полностью подчиняются этой теореме, они не всегда могут быть доступны, но всегда гарантируют согласованность.

2.4 Распределенный конечный автомат

Простейший способ реализовать распределённую систему - создать набор клиентов, которые посылают команды избранному центральному узлу-серверу. Сервер должен быть описан как детерминированный конечный автомат, который выполняет в определенном порядке команды от клиентов. Конечный автомат имеет собственное текущее состояние; шаг работы автомата состоит в обработке очередной команды, в результате автомат возвращает ответ клиенту и меняет своё текущее состояние.

Например, клиентами распределённой банковской системы могут быть клерки, а состояние детерминированного конечного автомата может состоять из счетов и балансов пользователей банка. Снятие налич-

ных могло бы быть выполнено через выполнение конечным автоматом команды, которая уменьшала бы баланс определённого счёта тогда и только тогда, когда баланс был бы больше суммы снятия, а затем генерировала бы старый и новый баланс в качестве результата обработки команды автомата для клиента.

Реализация, использующая избранный центральный сервер, не является отказоустойчивой, так как в случае отказа центрального сервера, система перестанет обрабатывать команды клиентов. Таким образом, для того, чтобы обеспечить отказоустойчивость, системе следует использовать набор серверов, каждый из которых представляет собой конечный автомат. Так как конечный автомат - детерминированный, все серверы обязаны выполнять одинаковые последовательности переходов состояний при условии одинаковых входных данных. Клиент, для которого выполняется команда, может использовать любой сервер.

Алгоритмы принятия консенсуса, как правило, возникают в контексте применения распределенных конечных автоматов. При таком подходе, конечные автоматы, расположенные на множестве серверов кластера выполняют идентичную команду, находясь в одном и том же состоянии - таким образом, кластер может продолжать работу, даже если некоторые из его серверов отказывают. Таким образом, распределенные автоматы используются для решения разнообразных проблем отказоустойчивости в распределенных системах.

Распределенные автоматы, как правило, реализуется с помощью реплицированного журнала, как показано на рисунке2.1. На каждом сервере хранится реплицируемый журнал, содержащий последовательность команд, которые его собственный конечный автомат выполняет в фиксированном порядке. Каждый журнал содержит одинаковые команды, в

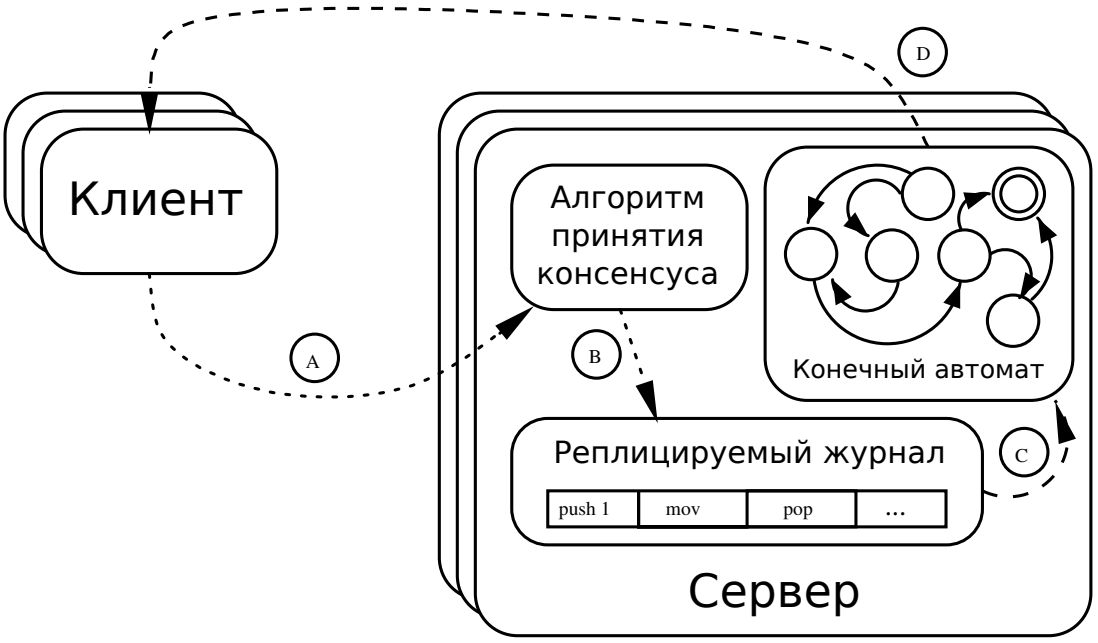


Рис. 2.1: Архитектура системы с использованием распределенного конечного автомата. Алгоритм принятия консенсуса управляет реплицированным журналом, который содержит команды для конечного автомата, которые пришли от клиентов. Конечные автоматы с идентичным состоянием выполняют идентичную последовательность команд из реплицированного журнала, поэтому они порождают одинаковые выходные данные.

одном и том же порядке, так что каждый реплицированный конечный аппарат обрабатывает идентичную последовательность команд. Поскольку конечные автоматы являются детерминированными, каждый обрабатывает идентичную команду при идентичном состоянии и получает идентичные результаты работы.

Поддержание реплицированного журнала согласованным является непосредственной задачей алгоритма принятия консенсуса. Модуль алгоритма принятия консенсуса на сервере получает команды от клиентов (A) и добавляет их в свой журнал (B). Он общается с модулями алгоритма принятия консенсуса на других серверах, чтобы гарантировать, что каждый журнал в конечном итоге содержит одни и те же команды в одинаковом порядке, даже если некоторые серверы отказали. После того, как команда клиента успешно реплицируется, конечный автомат каждо-

го сервера обрабатывает их согласно журналу (C), результаты работы возвращаются клиентам (D). В результате кластер серверов выглядит для клиента как единый, высоконадежный и распределенный конечный автомат.

Алгоритмы принятия консенсуса при практическом применении обладают следующими свойствами:

- обеспечивают безопасность (никогда не возвращают неправильный результат) при всех условиях (исключая византийские ошибки), в том числе при задержках в сети, потерьях сообщений, их дублировании и изменении порядка;
- полностью функциональны (система остается доступной) до тех пор, пока большинство серверов могут взаимодействовать и друг с другом, и с клиентами. Например, типичный кластер из пяти серверов может пережить отказ двух узлов. Узлы, как предполагается, при выходе из строя останавливаются; они могут позже восстановится и присоединиться к кластеру;
- не зависят от времени, чтобы обеспечить согласованность журналов: неисправные часы или большие задержки сообщений могут, в худшем случае, быть причиной проблем доступности;
- команда считается завершенной как только большинство серверов ответило с подтверждением успешного выполнения команды; меньшинство медленных узлов-серверов не должно повлиять на общую производительность системы.

3. Алгоритм Paxos

Как известно, согласованность в распределённой системе можно поддержить одним из протоколов распределённых транзакций[2, 3]. Один из таких протоколов - протокол двухфазного подтверждения транзакций, активно применяемый в транзакционных системах. Другим алгоритмом распределённых транзакций часто называют Paxos, однако следует понимать, что Paxos является протоколом принятия решений, а не подтверждения транзакций, так как его требования к участникам отличаются от требований к участникам в протоколе двухфазных блокировок.

Алгоритм Paxos также дает иные гарантии: в случае двухфазного протокола блокировок, результатом является подтверждение ACID-транзакции на каждом из серверов. В случае Paxos результатом является решение, принятое в кластере и верное лишь для кластера в целом - если случится так, что в результате серии отказов выживет лишь один из узлов, который отсутствовал в кластере по причине сбоя во время процесса принятия решения, то решение на этом сервере окажется непринятым, что является существенно более слабым результатом по сравнению с двухфазными блокировками.

Рассматривается набор процессов, которые делают друг другу предложения, каждое из которых представлено некоторым значением. Задача алгоритма консенсуса состоит в том, чтобы гарантировать, что только одно из сделанных предложений принято. Также, если никаких предло-

жений не поступало, то ничего и не должно быть принято. Если определённое предложение, представленное некоторым значением, было выбрано, тогда каждый участник должен иметь возможность узнать о факте принятия решения и его значении.

Из данного описания задачи вытекают следующие требования:

- принято может быть только одно из предложенных значений
- процесс не может придти к выводу, что предложение было принято, если только это действительно не произошло

Алгоритм Paxos не пытается точно обозначить насколько оперативно должен происходить процесс принятия решения. Однако, целью алгоритма является обеспечение гарантий того, что одно из предложенных значений в конце концов будет выбрано, и, когда выбор был сделан, любой процесс иметь возможность узнать это значение.

Любой процесс может выступать в одной из трёх ролей: *инициатор*, *выборщик*, *исполнитель*. В реальности один и тот же процесс может выступать в разных ролях в разных ситуациях.

Предполагается, что для принятия решений, процессы коммуницируют друг с другом, посылая друг другу сообщения. Paxos опирается на асинхронную модель пересылки сообщений, но без “византийских” ошибок, то есть:

- процессы работают с произвольной скоростью, могут остановиться или перезапуститься. Так как процесс может завершиться аварийно после участия в принятии решения по одному из предложений, но до того как он узнает о принятом решении, процесс не сможет узнать о принятом решении, если не будет способа персистентно

сохранить его состояние в отношении этого конкретного решения на период между завершением и перезапуском

- сообщения могут быть доставлены с произвольной скоростью, могут дублироваться и могут быть не доставлены вообще, но их содержимое не может быть повреждено

3.1 Процедура принятия решения

Одним из самых простых способов принятия решения в поставленных условиях было бы каким-то образом определиться относительно одного процесса, который бы всегда выступал в роли выборщика. Тогда, любой другой процесс сможет послать своё предложение выборщику и узнать от него результат - было ли данное предложение принято. Выборщик, в свою очередь, будет выбирать первое предложение, которое он получит.

К сожалению, хотя это решение и очень простое, оно невозможно на практике,так как в случае выхода из строя выборщика никакие предложения больше не смогут быть приняты.

Другим способом принятия решения было бы вместо одного выборщика использовать несколько. Инициатор посылает предложение со своим значением нескольким выборщикам,каждый из них может его принять. Решение считается принятым, когда достаточно большое количество выборщиков сошлись на одном и том же предложении.

Для того, чтобы гарантировать что только одно предложение из множества вариантов было выбрано, необходимо обеспечить простое большинство в пользу данного предложения. Выборщик, в свою очередь, не может принять больше двух предложений, так как иначе может возникнуть ситуация наличия большинства голосов по более чем одному

предложению.

При получении первого предложения у выборщика нет никакого способа узнать, будет ли он получать предложения в дальнейшем. Особенно эта проблема актуальна в среде, где сообщение может быть потеряно. Также, решение должно быть принято, даже если есть только один инициатор. Отсюда следует:

P1: *Выборщик принимает первое предложение, которое он получает.*

С этим требованием, однако, есть одна проблема. Несколько значений могут быть предложены разными инициаторами в примерно одно и то же время, из-за этого может получатся ситуация, когда каждый выборщик принял решение, но каждый своё, поэтому ни одно из предложений не набрало большинства голосов. Даже если есть только два значения, может случиться так, что примерно половина выборщиков примет первое значение, а оставшиеся выберут второе значение. В этом случае, выход из строя даже одного акцептора може привести к тому, что понять, какое решение было принято станет невозможным, так как оставшиеся выборщики поделятся в своих мнениях поровну (см. рисунок 3.1a).

Условие **P1** вместе с требованием, что предложение считается принятым тогда и только тогда, когда оно принято большинством акцепторов, приводит к тому, что выборщикам позволяется принимать более чем одно предложение. Различные предложения могут быть идентифицированы выборщиком с помощью присвоения им числового идентификатора, таким образом предложение должно быть представлено парой чисел ***(значение, номер предложения)***, а различные предложения должны получать различные номера.

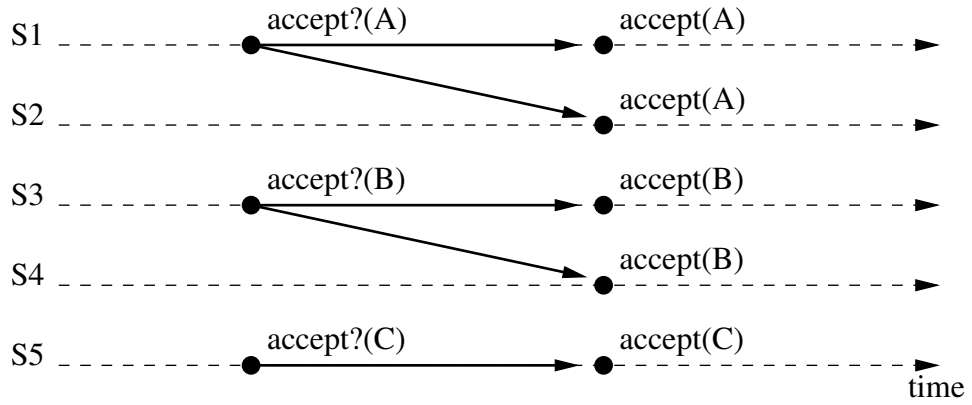
Как присваивать предложениям различные номера зависит от реализации. Один из вариантов: у каждого процесса есть уникальный иден-

тификатор, и каждый процесс хранит у себя в переменной max_round максимальный номер рауда протокола, который ему известен. Для того, чтобы создать предложение с новым номером, процессу нужно увеличить max_round на 1 и задать номер приложения как конкатенацию переменной max_round и уникального идентификатора процесса. Переменная max_round должна хранить свое значение персистентно, чтобы не потерять свое значение в случае отказа процесса.

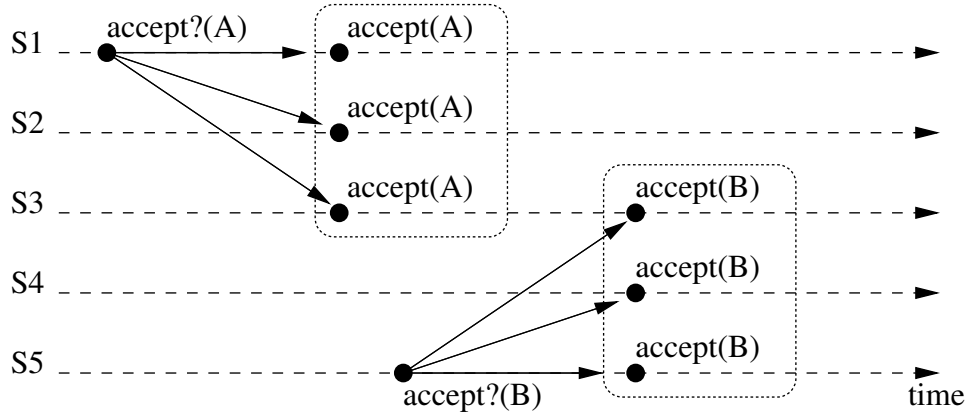
Таким образом, расширив понятие предложения, предложение считается принятым тогда, когда за предложенное значение проголосовало большинство выборщиков. При этом, так как различным предложениям присваиваются различные номера, одно и то же значение может быть использовано в различных предложениях.

Таким образом, несколько предложений могут быть приняты, но тогда и только тогда, когда они различаются только номером, но предлагают одно и то же значение (см. рисунок 3.1b). Чтобы это, в конце концов, произошло, по индукции по номеру предложения, достаточно гарантировать:

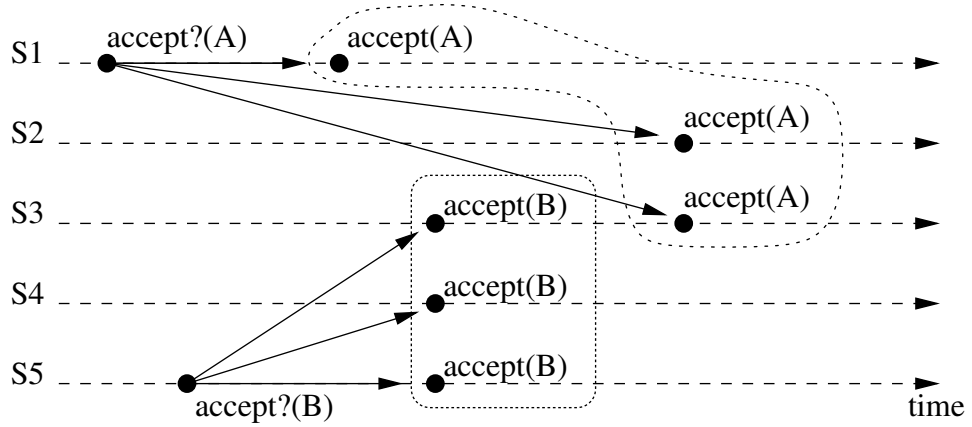
P2: *Если предложение* ***(v, N)*** *("принять значение* ***v*** *с номером предложения* ***N****") принято, тогда любое другое принятое тем же выборщиком предложение с номером, большим* ***N****, должно также предлагать значение* ***v***



(a) Разные выборщики выбрали разные значения, ни одно из значений не может быть принято.



(b) Противоречивость выбора в случае, если выборщик имеет возможность голосовать за несколько значений - несколько решений оказываются принятыми.



(c) Предложения должны быть упорядочены, чтобы устаревшие предложения можно было отвергнуть (предложение от *S1* должно быть отвергнуто *S3*)

Рис. 3.1: Примеры, показывающие необходимость определенных ограничений на логику выборщика

Так как идентификаторы предложений возрастают и различны, условие **P2** гарантирует что только одно значение выбрано. С другой стороны, для того, чтобы быть принятым, предложение должно быть принято хотя бы одним выборщиком. Следовательно, условие **P2** может быть удовлетворено если:

P2': *Если предложение* ***(v, N)*** *принято, тогда любое другое принятое любым выборщиком предложение с номером, большим* ***N****, должно также предлагать значение* ***v***

P2' не противоречит **P1**, следовательно, какие-то предложения будут приняты. Так как коммуникация между процессами асинхронная, предложение может быть принято даже если какой-то из выборщиков его вообще не получил. Пусть этот выборщик называется *"C"*

Представим себе, что новый инициатор пробуждается и делает предложение с номером **M**, **M** > **N**, с другим предлагаемым значением. **P1** обязывает "C" принять это предложение, таким образом нарушая **P2'**. Следовательно, для того чтобы **P1** и **P2'** были бы верны, необходимо:

P2": *Если предложенное значение* ***v*** *было принято, то любое предложение с номером* ***M****,* ***M*** $>$ ***N*** *должно иметь значение* ***v***

Из **P2"** следует **P2'** и **P2** т.к. любое принятое предложение должно быть предложено каким-то инициатором.

Инициатор должен иметь механизм, позволяющий ему узнать, какое предложение было принято, перед тем как сделать своё предложение. Для этого следует доказать, что **P2"** выполнено; необходимо продемонстрировать что, при условии когда некоторое предложение ***(v, M)***, любое предложение с номером ***N*** $>$ ***M*** имеет значение ***v***.

Применяется индукция по номеру предложения N. Пусть для любого $N > M$, все предложения в диапазоне ***M...N-1*** имеют значение v (см. рисунок 3.1c). Для того, чтобы M было принято, необходимо чтобы большинство выборщиков (пусть это множество называется множеством C) приняло его.

Тогда, по индукции, каждый выборщик в C также принял предложения ***M...N-1***, и каждое предложение в этом диапазоне имело значение v.

Так как любое множество, состоящее из большинства выборщиков, должно включать хотя бы одного из участников множества C, для того, чтобы предложение с номером N имело значение v необходимо, чтобы выполнялся следующий инвариант:

P3: *Для любых v и N предложение (v, N) делается только тогда, когда существует множество S, состоящее из большинства выборщиков, такое что выполняется одно из условий:*

- *Ни один выборщик из множества S не принял предложение с номером, меньшим N*
- *Значение v - это фиксированное предлагаемое значение предложений с номером меньше N, принятого выборщиками из S*

Это значит, что условие **P2"** может быть удовлетворено если инварианты из **P3** соблюдены. Для того, чтобы выполнить требования **P3**, инициатору, перед тем как сделать предложение с номером N необходимо узнать о последнем предложении с номером, меньшим N, которое было принято большинством выборщиков.

Узнать предложения, которые уже были приняты достаточно просто, а предсказать, какие предложения будут приняты в будущем - сложно.

Вместо того, чтобы пытаться предсказывать будущее, инициатор контролирует ход исполнения, требуя от выборщиков обещания, что в будущем ими не будут приняты неправильных решения. Другими словами, инициатор требует от выборщиков не принимать предложений с номером, меньшим $\boldsymbol{N}$. Такой подход ведёт к следующему алгоритму выдвижения предложений:

Инициатор выбирает номер предложения $\boldsymbol{N}$ и посылает запрос (называется "*Prepare*-запросом с номером $\boldsymbol{N}$") каждому выборщику, требуя вернуть:

- Обещание никогда не принимать предложения с номером $< \boldsymbol{N}$
- Информацию о последнем известном выборщику предложении, с максимальным номером, меньшим $\boldsymbol{N}$

Если инициатор получает ответ на запрос от большинства выборщиков, он может послать предложение $\boldsymbol{(v, N)}$ (причем выбирается значение с максимальным номером предложения $\boldsymbol{N}$), где $\boldsymbol{v}$ - значение выбранное среди:

- начального значения инициатора
- всех значений предложений которые инициатор получил от выборщиков в предыдущем пункте

Затем инициатор непосредственно делает своё предложения, посылая, множеству выборщиков сформированное по описанным выше принципам предложение. Это должно быть то же множество выборщиков, что и ответило на первоначальное предложение. Этот тип запроса называется *Accept-запросом.*

Выборщик может получить два вида запросов: *Prepare* и *Accept*. Так как любое сообщение всё равно может потеряться, выборщик может игнорировать любой запрос, без угрозы нарушения консенсуса. Выборщик всегда может ответить на *Prepare* запрос, в отличии от *Accept*-запроса. Выборщик может ответить на *Accept* только тогда, когда нужно принять предложение, то есть только тогда, когда выборщик на данный момент не обещал кому либо в ответе на какой-либо предыдущий *Prepare* запрос этого не делать.

Иными словами:

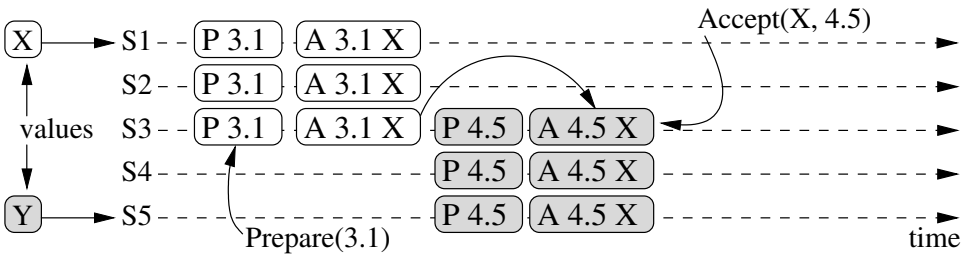
P1': *Выборщик может принять предложение с номером textbfN тогда и только тогда, когда он не отвечал положительно на запрос Prepare с номером, большим* ***N***

Конечная версия алгоритма может быть получена в результате применения небольшой оптимизации.

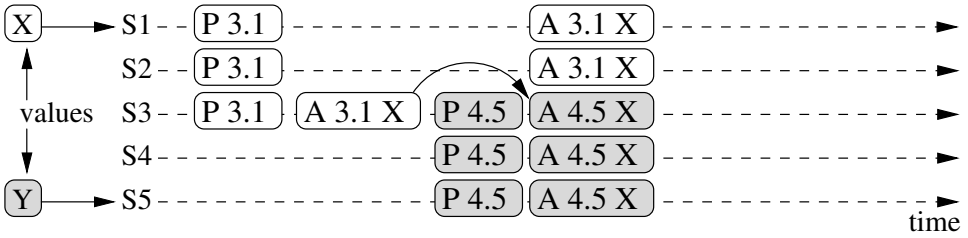
Предположим, выборщик получает *Prepare* запрос с номером **N**, но он уже ответил на **N** запрос с номером, большим **N**, таким образом гарантируя не принимать предложение с номером **N**. В этом случае выборщику нет причины отвечать на *Prepare* запрос, так как в любом случае он не примет предложение с номером **N**, которое хочет сделать инициатор, независимо от значения предложения. Таким образом, акцептор может свободно проигнорировать такой *Prepare* запрос. Он также может проигнорировать *Prepare* запрос для предложения, которое уже было принято.

В результате этой оптимизации, выборщик обязан помнить только последнее предложение которое он когда-либо принял (храниться номер последнего известного выборщику принятого предложения).

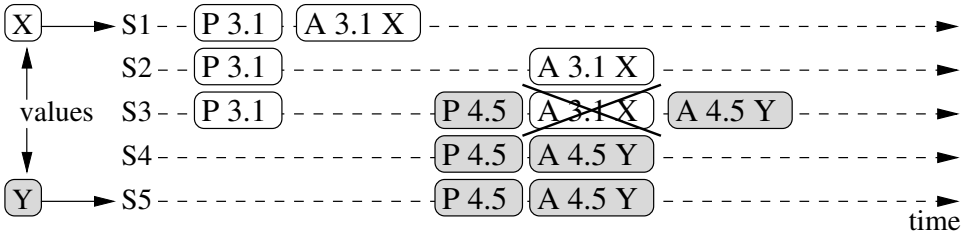
Так как условие **P3** должно выполняться даже в случае аварий и пе-



(a) Значение уже принято, новый инициатор узнает о нём и сменяет свое предлашаемое значение.



(b) Значение еще не принято, но инициатор узнает о нем в вызове *Prepare*. Новый инициатор пользуется значением, предложенным ранее. Оба инициатора успешно завершают голосование.



(c) Предыдущее значение не избрано и не известно новому инициатору. Новый инициатор использует собственное значение, голосование по предыдущему предложению заблокировано.

Рис. 3.2: Примеры сценариев работы Paxos

резапусков выборщика, эта информация должна хранится персистентно. С другой стороны, инициатор всегда может отказаться от своего предложения и забыть о нем, никода больше не делая предложений с тем же номером (т.е. предварительно не сделав *Prepare* запрос).

3.2 Описание фаз алгоритма Paxos

Учитывая действия выборщиков и инициаторов, алгоритм действует в двух следующих фазах.

Фаза 1

1. Инициатор выбирает предложение с номером **N** и выполняет *Prepare* запрос для всех выборщиков.

2. Если выборщик получает *Prepare* запрос с номером **N**, большим чем номер любого другого *Prepare* запроса, на который этот выборщик когда-либо отвечал, он отвечает на запрос, обещая никогда не принимать предложение с номером, меньшим **N** и с информацией о предложении с наибольшим $\mathbf{M} < \mathbf{N}$, которое выборщик принял последним.

Фаза 2

1. Если выборщик получает ответ на *Prepare* запрос с номером **N** от большинства выборщиков, он посылает *Accept* запрос каждому из выборщиков, содержащий предложение **(v, N)**, где значение **v** выбранно в результате поиска среди всех ответов на *Prepare* запрос по максимальному номеру предложения, или со значением, посланным инициатором, если *Prepare* запрос не вернул ни одного предложения.

2. Выборщик получает *Accept* запрос для предложением с номером **N**, он принимает предложение если только не отвечал на *Prepare* запрос с номером, большим **N**

Инициатор может сделать несколько предложений, только если для каждого из них он следует этому алгоритму. Инициатор также может отказаться от предложения в любой момент времени. Также, инициатору следует отказываться от предложения, если какой-то другой инициатор сделал предложение с большим номером. Таким образом, если выборщик игнорирует *Prepare* или *Accept* запрос (так как он уже получил *Prepare* или *Accept* запрос с большим номером), он может проинформировать об этом инициатора, который в этом случае должен отказаться от своего предложения. Это повышает эффективность алгоритма, но не влияет на его корректность.

3.3 Получение информации о принятом решении

Для того, чтобы понять, что по какому-то решению достигнут консенсус, исполнитель должен убедиться что оно было принято большинством выборщиков. Очевидный алгоритм - каждый выборщик, когда бы он ни получил предложение, которое он решил принять, пересылает принятое решение всем исполнителям. Это позволяет исполнителям узнать о выбранном предложении, как только оно было принято. Также это требует от выборщика коммуницировать с каждым исполнителем, таким образом каждое выбранное предложение будет генерировать количество сообщений, пропорциональное произведению количества выборщиков на количество исполнителей.

Предположение об отсутствии ошибок “византийского” типа в среде передачи сообщений позволяет одному исполнителю доверять информации, полученной от другого исполнителя. Иными словами, исполнитель

может узнать от другого исполнителя, что определённое предложение было принято. Выборщики могут информировать о своих решениях избранного исполнителя, который, в свою очередь, проинформирует всех остальных в тот момент, когда определит, что консенсус был достигнут. Этот подход требует дополнительных обменов сообщениями, и, следовательно, внесет задержки для всех исполнителей. Это также менее надёжно, так как избранный исполнитель может аварийно остановится. Однако, этот подход экономит количество пересылаемых сообщений - оно пропорционально сумме количества выборщиков и количества исполнителей.

В общем случае, выборщики могут информировать о своих решениях некоторое фиксированное множество избранных исполнителей, каждый из которых в дальнейшем проинформировал бы остальных исполнителей о принятом решении. Расширение множества избранных исполнителей повышает надёжность системы, но увеличивает количество пересылаемых сообщений.

В условиях возможных потерь сообщений некоторое значение может быть выбрано, но при этом ни один исполнитель об этом никогда не узнает, так как соответствующие сообщения были потеряны. Исполнитель, таким образом, должен иметь возможность узнать от выборщиков какие решений они приняли, но отказ выборщика может сделать это действие неосуществимым.

Вследствие этого, в случае отказа выборщика, нет способа узнать, что решение принято большинством. Если исполнителю необходимо узнать о конкретном предложении, было ли оно принято, или нет, он может попросить инициатора сделать предложение о принятии этого решения, используя алгоритма, описанного выше.

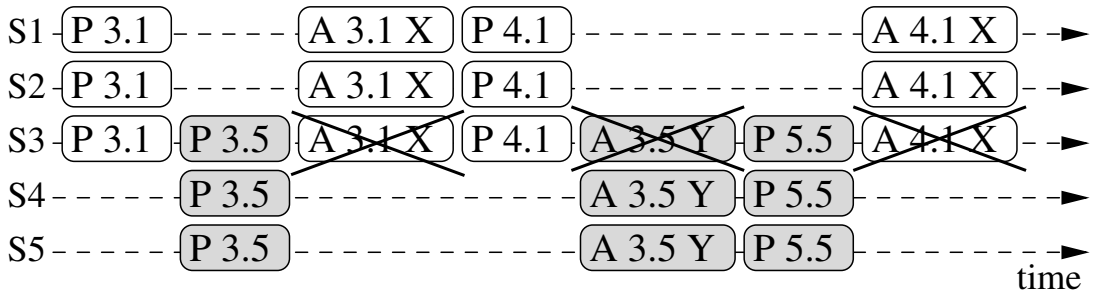


Рис. 3.3: Конкурирующие инициаторы погут создать ситуацию взаимоблокировки.

3.4 Обработка ситуации взаимоблокировки

Существует сценарий, в котором есть два инициатора, которые делают предложения с увеличивающимися номерами, и ни одно из этих предложений не может быть принято. Инициатор **S1** завершает фазу 1 предложения **N1**. После этого, другой инициатор **S5** завершает фазу 1 предложения **N2** (**N2** $>$ **N1**). Инициатор **S1** переходит в фазу 2 и посылает *Accept* запросы, но они игнорируются, так как выборщики уже пообещали не принимать предложения с номерами меньшими **N2**. Не имея консенсуса, инициатор **S1** заново начинает фазу 1 для предложения с номером **N3** (**N3** $>$ **N2**), и успевает завершить её, таким образом препятствуя консенсусу по решению **N2**. Подобная последовательность событий может продолжаться неограниченно долго, создавая ситуацию взаимоблокировок.

Один из способов избежать этой ситуации состоит в том, чтобы добавить случайную задержку перед попыткой заново начать фазу 1 для предложения ввиду отсутствия консенсуса, чтобы позволить завершить свой раунд голосования другим инициаторам.

Другой вариант избежать этих ситуаций состоит в том, чтобы только избранный инициатор должен использоваться для создания предложений. Если избранный инициатор может успешно коммуницировать с

большинством выборщиков, и он использует предложения с номерами большими, чем он когда-либо уже использовал, тогда он преуспеет и в достижении консенсуса по новому предложению. В случае, если избранный инициатор узнает о запросе с большим номером, он может отказаться от предложения, и в конце концов выбрать предложение с большим номером.

Если достаточная часть процессов системы (инициаторы, выборщики, сеть) функциональны, живучесть системы может быть достигнута посредством процедуры выборов избранного инициатора.

3.5 Особенности реализации

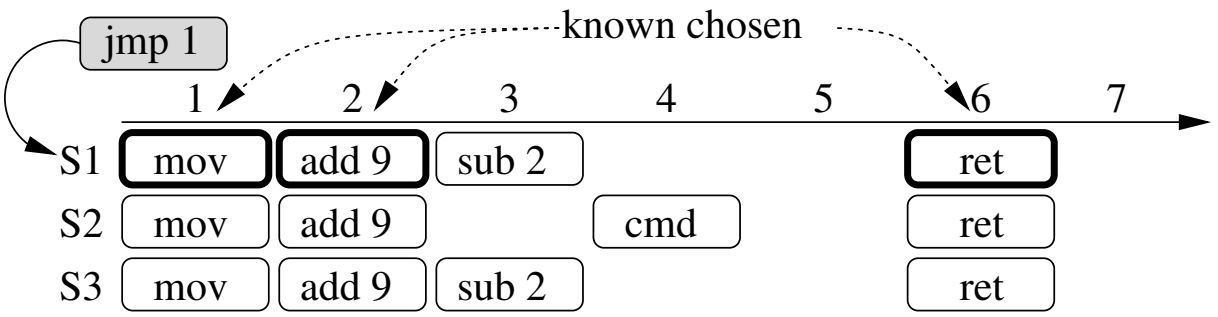
Paxos предполагает, что взаимодействующие процессы объединены в сеть. В алгоритме консенсуса каждый процесс играет роль инициатора, выборщика или исполнителя. Алгоритм выбирает лидера, который играет роль избранного инициатора и избранного исполнителя. В алгоритмах Paxos запросы и ответы посылаются в виде обычных сообщений, а ответы на запросы должны содержать соответствующий номер предложения, чтобы избежать путаницы в случае обработки сразу нескольких запросов. Также, некоторые переменные нужно хранить персистентно, так как в противном случае при аварийном отказе могут возникнуть ситуации, нарушающие инварианты, на основании которых работает Paxos. Например, перед отправкой ответа на запрос выборщик обязан записать выбранное им значение в персистентное хранилище.

3.6 Переход от Single-Paxos к Multi-Paxos

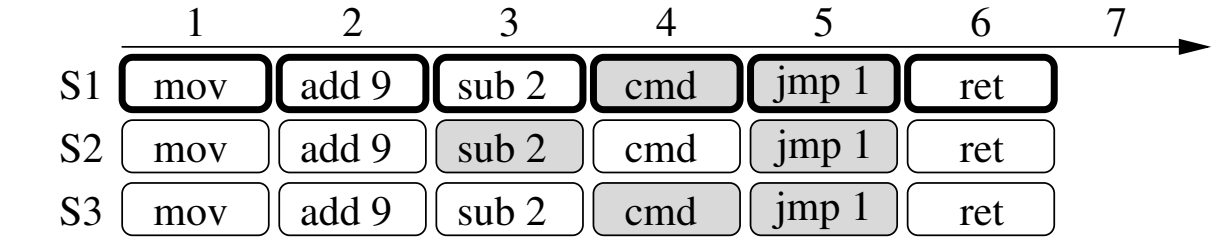
Предметом рассмотрения выше был алгоритм принятия консенсуса только по одному вопросу, так называемый Single-Paxos. Теперь, используя этот алгоритм в качестве базового, будет описан алгоритм для ведения реплицированного журнала в контесте задачи распределенного конечного автомата. Это алгоритм называется Multi-Paxos.

Если Single-Paxos решает задачи того, что какие-то участники должны договорится о неком общем значении, то в Multi-Paxos возникает так же еще один участник - клиент, от которого приходят определенные значения в качестве вариантов для голосования. Также, клиент должен иметь возможность узнать, было ли его решение принято кластером, или нет. Клиент так же может отказать, что может привести к проблеме, что он может оказаться в неведенье относительно того, было ли его решение принято, или нет.

Более того, клиентов может быть несколько, это вынуждает использовать реплицированный журнал для устранения возможных противоречий - если бы для разных клиентов производились просто параллельные, независимые голосования, это могло бы привести к тому, что порядок получившихся результатов голосований был бы несогласованным для всего кластера. Иначе говоря, для части серверов сначало было произведено принятие A, затем принятие B, а для остальной части - наоборот.



(a) Реплецированный журнал до процедуры выбора индекса.



(b) Реплецированный журнал после процедуры выбора индекса.

Рис. 3.4: Алгоритм выбора индекса для записи предложения в реплецированный журнал.

Сценарий работы реплицированного конечного автомата выглядит следующим образом:

1. *Клиент посылает команду для выполнения на сервер;*

2. *Используя Paxos, сервер реплицирует запрос клиента на все машины, все команды копируются в фиксированных для всех серверов порядке;*

3. *Сервер дожидается принятия решения по предыдущим записям в журнале, затем применяет их, таким образом выполняя предыдущие команды клиентов;*

4. *Сервер применяет выбранную команду и возвращает результат клиенту;*

Для того, чтобы хранить команды в фиксированном глобальном порядке, каждое предложение должно иметь новую переменную - индекс в реплицированном журнале. Одной из задач алгоритма является обеспечение уникальности этого значения.

Для этого, предположим, у нас есть инициатор A. Ему известно из его локального журнала, по каким предложениям уже были приняты решения, а значит, их номера позиций точно не могут быть использованы в качестве индекса. Также, инициатору следует исключить номера, по которым он уже запустил утверждение предложения, но которые еще не завершились. Таким образом, следует найти первый свободный индекс и попытаться его утвердить - выполнить *Prepare* предложения с этим индексом как одним из параметров (позицией в журнале).

В ответ на этот запрос инициатор может получить ответ что запись кем-либо уже используется, значит надо повторить эту процедуру с номером на 1 большим. Стоит отметить, что в этом ответе также следует передавать информацию об уже существующем по этому индексу предложении, а значит, можно внести это предложение в журнал.

Если же инициатор получает в ответе сообщение о том, что запись не используется, он может утвердить ее, и начать новый цикл обработки запроса клиента.

Таким образом, алгоритм выбора индекса предложения выглядит таким образом:

1. *Найти свободный индекс в журнале.*
2. *Выполнить* ***Prepare*** *запрос Paxos-алгоритма с этим индексом.*
3. *Проверить результат* ***Prepare*** *запроса:*
   (a) *Если возвращено другое значение, то переключиться на это новое значение, заполнить свой журнал, после этого вернуться на шаг 1.*
   (b) *Если возвращено наше значение: вызвать Accept-запрос, получить новую команду клиента и вернуться на шаг 1.*

Также, следует дополнительно отметить, что Accept должен повторно выполнятся в фоновом режиме для всех участников, от которых не был получен ответ.

4. Алгоритм Raft

Raft - алгоритм принятия консенсуса для управления реплицированным журналом. Он эквивалентнен Multi-Paxos, и задуман быть столь же эффективным, как и Multi-Paxos. Однако, его архитектура отличается от Multi-Paxos; она задумана быть понятнее и относительно легко подходит для использования на практике. Для простоты, Raft разделяет ключевые элементы задачи принятия консенсуса на отдельные стадии, такие как: выборы лидера, репликация журнала и обеспечение безопасности - это уменьшает сложность алгоритма, уменьшая число стадий, которые должны быть рассмотрены. Результаты исследования**??** показывают, что архитектура Raft проще для понимания, чем Paxos. Raft также включает в себя механизм для изменения состава распределенной системы, который необходим для практического использования.

Алгоритм принятия консенсуса, как известно, позволяет множеству машин работать в качестве связанной, согласованной группы, которая может пережить отказ некоторых из её членов. Из-за этого, такие алгоритмы играют ключевую роль в создании надежных, крупномасштабных, распределенных систем. Семейство алгоритмов Paxos превалирует среди обсуждаемых алгоритмов принятия консенсуса в течении последнего десятилетия: большинство реализаций алгоритма принятия консенсуса были сделаны на основе Paxos или под его влиянием, например, Parallels Cloud Server для управления своими серверами с метаданными

использует именно этот алгоритм**??**.

К сожалению, алгоритмы семейства Paxos довольно трудно понять, несмотря на многочисленные попытки сделать их яснее. Кроме того, для применимости на практике архитектура требует модификации. При разработке алгоритма Raft были применены специальные методы для улучшения понятности, в том числе декомпозиция - Raft отдельно выделяет выборы лидера, репликацию журнала и обеспечение безопасности; так же было применено уменьшение пространства состояния (по отношению к Paxos, Raft снижает степень недетерминизма и серверам проще оставаться согласованными по отношению друг к другу).

Raft во многом похож на существующие алгоритмы консенсуса, но он имеет несколько нововведений:

- **сильное лидерство**: в отличии от других алгоритмов консенсуса, Raft использует "сильную" форму лидерства. Например, записи в реплицируемом журнале распространяются только от лидера на другие серверы. Это упрощает управление репликацией журнала и делает Raft проще для понимания;

- **процедура избрания лидера**: Raft использует рандомизированные таймеры во время процедуры избрания лидера. Это нововведение требует незначительного изменения механизма периодических контрольных сообщений, который необходим практически для любого алгоритма консенсуса; он позволяет разрешать конфликты просто и быстро;

4.1 Недостатки Paxos с точки зрения Raft

За последние десять лет, семейство алгоритмов Paxos стало почти синонимом алгоритма принятия консенсуса: этот протокол наиболее часто учат на курсах и большинство реализаций консенсуса используют его в качестве отправной точки. Paxos сначала определяет протокол, способный достичь соглашения в отношении единственного решения, например такого, как выбор одной записи для репликации, его называют Single-Paxos. Затем объединяются несколько экземпляров этого алгоритма, чтобы обеспечивать ряд решений, таких как управление реплецированным журналом Multi-Paxos. Paxos обеспечивает безопасность и отказоустойчивость, и он поддерживает изменения в составе кластера. Его корректность была доказана, и он является эффективным в нормальном случае.

К сожалению, Paxos имеет два существенных недостатка. Первый недостаток в том, что Paxos исключительно трудно понять. Полное объяснение, неясно и запутанно, даже для опытных исследователей.

Предположительно, непрозрачность Paxos происходит от использования Single-Paxos в качестве своей основы. Single-Paxos разделен на два этапа, которые не имеют простых интуитивных объяснений и не могут быть поняты независимо друг от друга. Из-за этого трудно понять почему протокол Single-Paxos работает. Композиция нескольких экземпляров Single-Paxos для создания Multi-Paxos добавляет значительную сложность и неясность. Создатели Raft считают, что целом проблема создания алгоритма принятия консенсуса для нескольких решений (т.е., управление журналом вместо обработки только одной записи) должна решаться другими методами, которые являются более прямым и очевидным.

Вторая проблема с Paxos, что он не очень хорошую приспособлен для практического применения. Одной из причин является то, что нет хорошо документированного алгоритма для Multi-Paxos. Описание Paxos в основном содержит Single-Paxos; есть только описание подходов к реализации Multi-Paxos, но многие детали отсутствуют. Такие системы, как Chubby[4] реализовали Paxos-подобный алгоритм, но не были опубликованы детали.

Кроме того, архитектура Paxos не очень хорошо продуманна для использования в реальных системах; это еще одно следствие построения Multi-Paxos на основе Single-Paxos. Например, мало пользы записи отдельных записей в журнал независимо друг от друга, так как в случае сбоя возможно нарушение согласованности. Это требует дополнительных усилий для построения алгоритма для управлением журналом. Проще и эффективнее разработать систему вокруг журнала, где новые записи добавляются последовательно в фиксированном порядке. Еще одна проблема в том, что Paxos использует подход каждый сервер взаимодействует с каждым. Это имеет смысл в упрощенном мире, где будет производиться только одно решение, но маловероятно, что реальные системы могут использовать это решение, так как его сложность становится $\mathcal{O}(n^2)$, где n - число серверов. Есть способы избежать этого с помощью избрания отдельного сервера в качестве лидера, что добавляет сложность к реализации.

В результате, практические реализации имеют мало общего с Paxos. Каждая реализация начинается с Paxos, обнаруживает трудности в ее реализации, а затем значительно изменяет архитектуру. Это отнимает много времени и приводит к ошибкам, и трудности Paxos усугубляют проблему. Paxos может быть хорошим для доказательства теорем о правильно-

сти алгоритма принятия консенсуса, но реальные реализации настолько отличаются от Paxos, что доказательства имеют мало смысла. Характерен следующий комментарий от авторов проекта Chubby: “Огромная пропасть лежит между описанием алгоритма Paxos и реально применяемым алгоритмом консенсуса...Окончательная реализация будет основана на непроверенном протоколе”

Из-за этих проблем, с точки зрения создателей алгоритма Raft, Paxos не является хорошим выбором для построения реальных систем или для обучения. Учитывая важность консенсуса в крупномасштабных программных систем, был разработан альтернативный алгоритм консенсуса с улучшенными свойствами.

4.2 Raft как алгоритм принятия консенсуса

У создателей Raft было несколько целей во время проектирования алгоритма Raft: требовалось обеспечивать полную описание для построения реальных системы, чтобы значительно снижает объем работы, необходимой со стороны разработчиков, использующих этот алгоритм; он должен быть безопасным при любых условиях, и доступен в типичных условиях эксплуатации; он должен быть эффективным при выполнении обычных операций. Но наша самая важная цель - понятность. Алгоритм должен быть доступен для понимания и расширения, которые неизбежны в реальных реализациях.

Один из подходов заключается в сокращении пространства состояний путем уменьшения числа состояний. В частности, в реплицированном журнале не должно быть “дырок”, и Raft ограничивает ситуации, в которых разные журналы могут стать несовместимыми друг с другом.

Хотя в большинстве случаев авторы алгоритма пытались устранить недетерминизм, есть некоторые ситуации, когда на самом деле он улучшает понятность. В частности, рандомизированные таймеры вносят недетерминизм, но они, как правило, чтобы уменьшают пространство состояний путем обработки всех возможных вариантов в некотором обобщеном виде. В частности, этот подход использовался для упрощения процедуры выборов лидера.

Raft реализует принятия консенсуса с помощью избрания единственного лидера, предоставляя ему полную ответственность за управление реплицируемого журнала. Лидер принимает записей в журнал от клиентов, копирует их на другие сервера, и говорит остальным серверам, когда можно безопасно использовать записи журнала в своих реплицируемых конечных автоматах. Идея наличия специального лидера упрощает управление реплицированным журналом. Лидер может не остановится или отключиться от других серверов, в этом случае будет избран новый лидер.

Благодаря наличию лидера Raft разделяет проблема консенсуса на три относительно независимых подзадачи, которые обсуждаются ниже:

- процедура выборов лидера: новый лидер должен быть выбран, когда существующий лидер отказывает
- репликация журнала: лидер должен получать записи в журнал от клиентов и реплицировать их в кластере, вынуждая другие журналы быть согласованными с его собственным журналом
- безопасность: если для какого-либо сервера в его конечном автомате была применена запись из журнала, то никакой другой сервер не может применить другую команду для того же индекса журнала.

Ниже будет описанно, как Raft обеспечивает это условие

4.3 Основые состояния

Кластер, в котором применяется Raft, обычно содержит несколько серверов; Для удобства рассмотрим кластер из пяти серверов, такой размер позволяет системе пережить два отказа. В любой момент времени каждый сервер находится в одном из трех состояний: лидер, ведомый, или кандидат. В нормальном режиме работы существует ровно один лидер и все другие сервера являются ведомыми. Ведомые пассивны: они самостоятельно не порождают никаких запросов, а просто отвечают на запросы от лидера и кандидатов. Лидер обрабатывает все клиентские запросы (если клиент обращается к ведомому, он перенаправляет его на лидера). Третье состояние, кандидат, используется, чтобы выбрать нового лидера, как описано в разделе . Рисунок 2 показывает состояния и их переходы; переходы будут рассмотрены ниже.

Raft делит время на эпохи произвольной длины, как показано на рисунке 3. Эпохи нумеруются последовательно возростающими целыми числами. Каждая эпоха начинается с выборов, в которых один или более кандидатов пытаются стать лидером, как описано в разделе. Если кандидат побеждает на выборах, то он становится лидером для оставшейся части эпохи. В некоторых редких ситуациях выборы заканчиваются ничьей среди нескольких кандидатов. В этом случае эпоха принудительно заканчивается без лидера; новая эпоха (с новыми выборами) начинается непосредственно после окончания текущей. Raft гарантирует, что существует не более одного лидера в отдельно взятой эпохе.

Различные сервера могут наблюдать переходы между эпохами в раз-

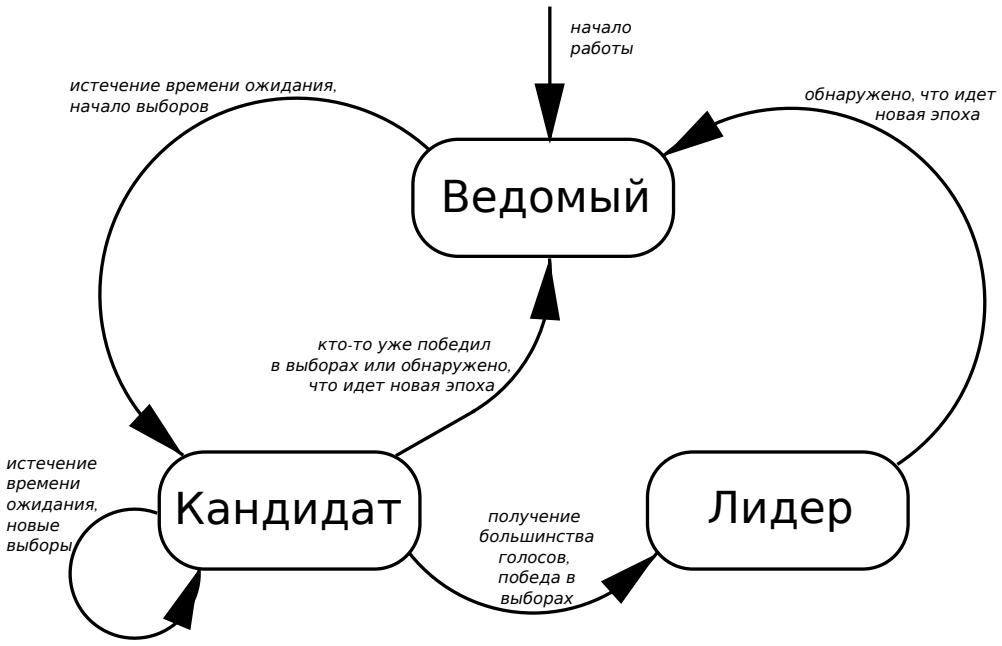


Рис. 4.1: Возможные состояния серверов. Ведомые только отвечают на запросы других серверов. Если ведомый не получает запросов в течении некторого времени, он становится кандидатом и начинает процедуру выбора лидера. Кандидат, которые получает большинство голосов становится новым лидером.

личное время, а в некоторых ситуациях сервер может по какой-либо видеть не наблюдать отдельные выборы или даже целые эпохи. Эпохи действуют в качестве логических часов в Raft; они позволяют обнаруживать сервера с устаревшей информацией, например такой, как “устаревшее” лидерство. Каждый сервер хранит номер эпохи, который монотонно возрастает с течением времени. При любом взаимодействии двух серверов, они обмениваются своими номерами эпохи, если номера эпох отличаются, то эти номера увеличиваются до большего значения. Если кандидат или лидер обнаруживает, что его номер эпохи устарел, он сразу же возвращается в состояние ведомого. Если сервер получает запрос с устаревшим номером сессии, он отвергает запрос.

В Raft сервера взаимодействуют с помощью механизма удалённого вызова процедур (*Remote Procedure Call*), и базовый алгоритм Raft требует только два типа RPC. *RequestVote* вызывается кандидатами во вре-

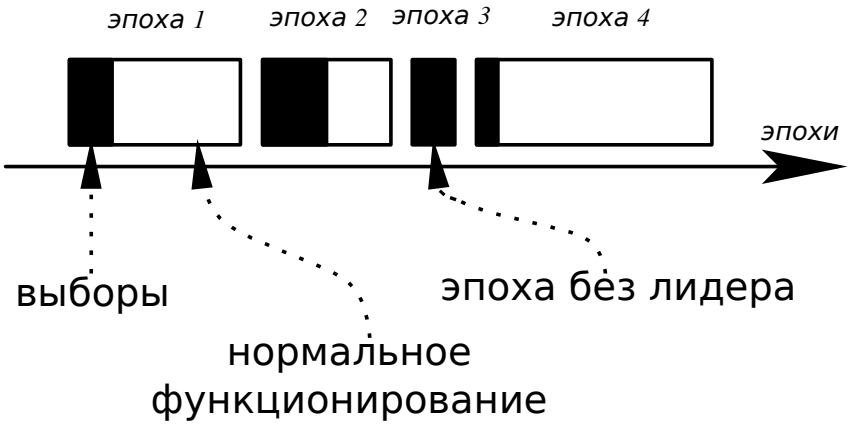


Рис. 4.2: Время делится на эпохи, и каждая эпоха начинается с процедуры выборов лидера. После успешных выборов, лидер управляет кластером до конца эпохи. Некоторые выборы заканчиваються неудачно, и в этом случае эпоха заканчивается без выбора лидера. Переходы между эпохами можно наблюдать в различные моменты времена на различных серверах.

мя выборов; *AppendEntries* вызывается лидером для копирования записи в реплицированный журнал и обеспечивает механизм периодических контрольных сообщений. Сервера повторяют попытку вызова удалённой процедуры при отсутствии успешного ответа на вызов процедуры. Также, для лучшей производительности, сервера выполняют вызовы удалённых процедур параллельно.

4.4 Процедура выборов лидера

Raft использует механизм периодических контрольных сообщений для управления состоянием кластера, в том числе, для проведения выборов лидера, когда они нужны. При запуске сервера начинают свою работу в состоянии ведомых. Сервер остается в состоянии ведомого до тех пор, пока он получает корректные RPC от лидера или кандидатов. Лидер переодически отправляет контрольные сообщения путем вызова *AppendEntries* для всех ведомых(в том числе и в ситуации, когда нет новых записей для добавления в журнал нет). Если ведомый не получа-

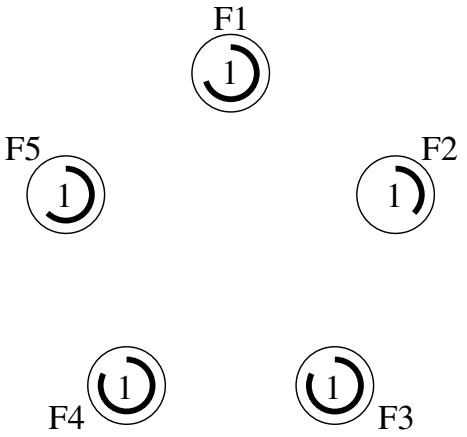
ет сообщения от лидера в течение определенного периода времени под названием “период ожидания выборов”, то он предполагает, что лидера больше нет, и начинает выборы, чтобы избрать нового лидера.

Для того, чтобы начать выборы, ведомый увеличивает свой номер эпохи и переходит в состояние кандидата. Затем голосует за себя, взводит внутренний таймер и параллельно вызывает *RequestVote* для каждого из других серверов в кластере. Кандидат продолжает оставаться в таком состоянии до тех пор, пока не случится одно из этих событий:

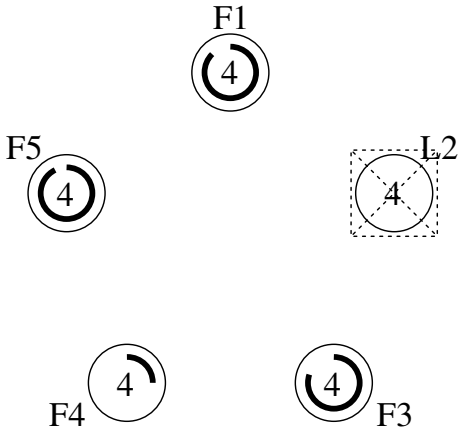
- кандидат победает на выборах
- другой кандидат побеждает и устанавливает себя в качестве лидера
- ничья; эпоха заканчивается без лидера

4.4.1 Победа в выборах

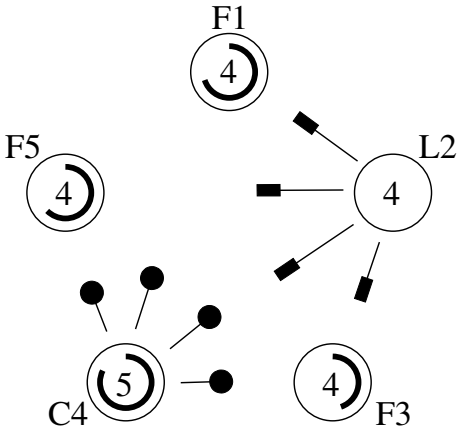
Кандидат победает на выборах, если он получает в той же эпохе голоса “за” от большинства серверов всего кластера. Каждый сервер будет голосовать как минимум за одного кандидата в отдельно взятой эпохе, по принципу *FIFO*. Необходимость большинства голосов “за” для победы гарантирует, что не более, чем один кандидат может победить на выборах для конкретной эпохи. После того, как кандидат побеждает на выборах, он становится лидером и начинает отправляет периодические контрольных сообщения всем другим серверам, чтобы подтвердить своё лидерство и предотвратить новые выборы.



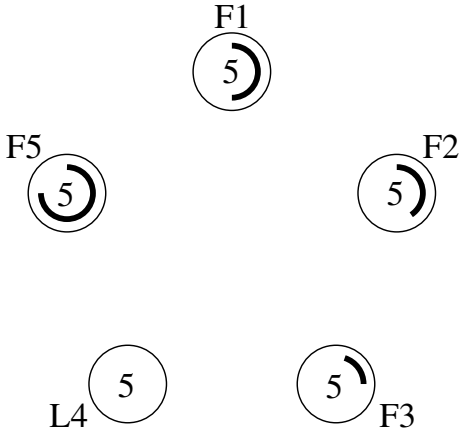
(a) Начальное состояние, все сервера являются ведомыми. Так как у *F2* рандомизированный таймер сработает раньше других, *F2* раньше остальных перейдет в состояние кандидата и начнет выборы, что с большой вероятностью гарантирует ему победу



(b) Кластер находился в эпохе номер 4 с лидером *L2*, однако, по какой-то причине *L2* отказывает. В отсутствии контрольного сообщения от лидера, таймера ведомых отчитывают время до начала голосования. У *F4* таймер сработает первым, он начнет процедуру голосования



(c) Сервер *C4* становится кандидатом, увеличивает номер своей эпохи на единицу, и отправляет всем серверам сообщение с посьбой проголосовать за него. В это же время сервер *L2* восстанавливается после сбоя, и посылает контрольное сообщение всем серверам для подтверждения своего лидерства



(d) Сервер *C4*, получив большинство голосов (от *F1*, *F3*, *F5*), становится лидером *L4*. *L4*, получив сообщение от *L2*, отвергает его, так как его номер эпохи больше. *L2*, получив от *L4* сообщение, становится ведомым *F2* с эпохой 5, так как обнаруживает, что его номер эпохи устарел

Рис. 4.3: Возможный сценарий поведения кластера в алгоритме Raft.

4.4.2 Победа другого кандидата

В ожидании голосов, кандидат может получить *AppendEntries* от другого сервера, который является лидером. Если номер эпохи лидера (которые передается как один из параметров RPC) не меньше, чем у кандидата, то кандидат признает лидера законным и возвращается в состояние ведомого. Если номер эпохи лидера в RPC меньше текущей эпохи кандидата, то кандидат отвергает RPC и продолжается оставаться в состоянии кандидата

4.4.3 Ничья

Третий возможный исход, состоит в том, что кандидат ни побеждает, ни проигрывает выборы: если много ведомых попытаются стать кандидатами в приблизительно одно и то же время, голоса могут быть разделены таким образом, что ни один кандидат не получит большинство. Когда это происходит, у кандидов начинают истекать внутренние таймера (перед выборами каждый кандидат выставляет свой собственный таймер и запускает его). Затем, каждый кандидат с вышедшим таймером попробует начать новые выборы, увеличивая номер эпохи и начиная новый раунд вызовом *RequestVote* для всех остальных серверов. Без дополнительных мер “ничья” может повторяться до бесконечности.

Raft использует рандомизированные таймера, чтобы обеспечить, что ситуации “ничьих” будут редки, а если они и случатся, то будут разрешены быстро. Чтобы предотвратить такие ситуации, интервалы срабатывания таймеров для кандидатов и ведомых выбираются случайным образом из фиксированного диапазона (например, 150-300 миллисекунд). Это приводит к тому, что в большинстве случаев только один ведомый

начинает выборы; он выигрывает выборы и посылает контрольное сообщение до того, как у любых других серверов сработает таймер. Тот же механизм используется для обработки “ничьих” во время выборов. Каждый кандидат перезапускает его рандомизированный таймер в начале выборов, и ждет истечения таймера, чтобы начать новые выборы; это уменьшает вероятность еще одной “ничьей” в новых выборах.

4.5 Репликация журнала

После того, как лидер был выбран, он начинает обслуживать запросы клиентов. Каждый запрос клиента содержит команду, которая должна быть выполнена на распределенном конечном автомате. Лидер добавляет команду в свой журнал как новую запись, затем параллельно для каждого сервера вызывает *AppendEntries* чтобы реплицировать эту запись. Когда запись была благополучно реплицированна (эта процедура подробно описывается ниже), лидер применяет команду из новой записи в его конечном автомате и возвращает результат этого вычисления клиенту. Если же какие-то ведомые отказывают, работают медленно, или сетевые пакеты для них теряются, лидер вновь вызывает *AppendEntries* (даже если лидер уже ответил клиенту), пока все ведомые в конечном счете не сохранят эту запись в своих журналах.

Организация журналов показанна на рисунке 5. Каждая запись журнала хранит команду конечного автомата вместе с номером эпохи, когда запись была полученна лидером. Номера эпох в записях журнала используются для обнаружения несоответствий между журналами. Каждая запись журнала также имеет целочисленный индекс, являющийся позицией в журнале.

Лидер решает, когда можно выполнять очередную запись журнала в распределенном конечном автомате; такие записи называются *фиксированными*. Raft гарантирует, что фиксированные записи не будут потеряны и в конечном итоге будут выполнены на всех доступных конечных автоматах. Запись журнала фиксируется лидером, создавшим запись, причем происходит это в тот момент, когда запись успешно реплицируется на большинстве серверов кластера. Эта операция также фиксирует все предыдущие записи в журнале лидера, в том числе записи, созданные предыдущими лидерами. Лидер отслеживает наибольший индекс фиксированной записи, и включает этот индекс в вызовы *AppendEntries* (в том числе и при посылке периодических контрольных сообщений), поэтому другие серверы в конечном итоге получают актуальное значение индекса последней записи. После того, как ведомый узнает, что запись журнала была зафиксированна, он применяет эту запись на своем локальном распределенном конечном автомате (записи обрабатываются в порядке, соотвествующем порядку в журнале).

Raft гарантирует следующие свойства:

- если две записи в нескольких логах имеют одинаковый индекс и номер эпохи, они содержат одну и ту же команду
- если две записи в нескольких логах имеют одинаковый индекс и номер эпохи, то все предыдущие команды в этих журналах инденичны

Первое свойство следует из того, что лидер создает не более чем одну запись с фиксированным индексом и фиксированным номером эпохи, а все записи никогда не изменяют своего положения в журнале. Второе

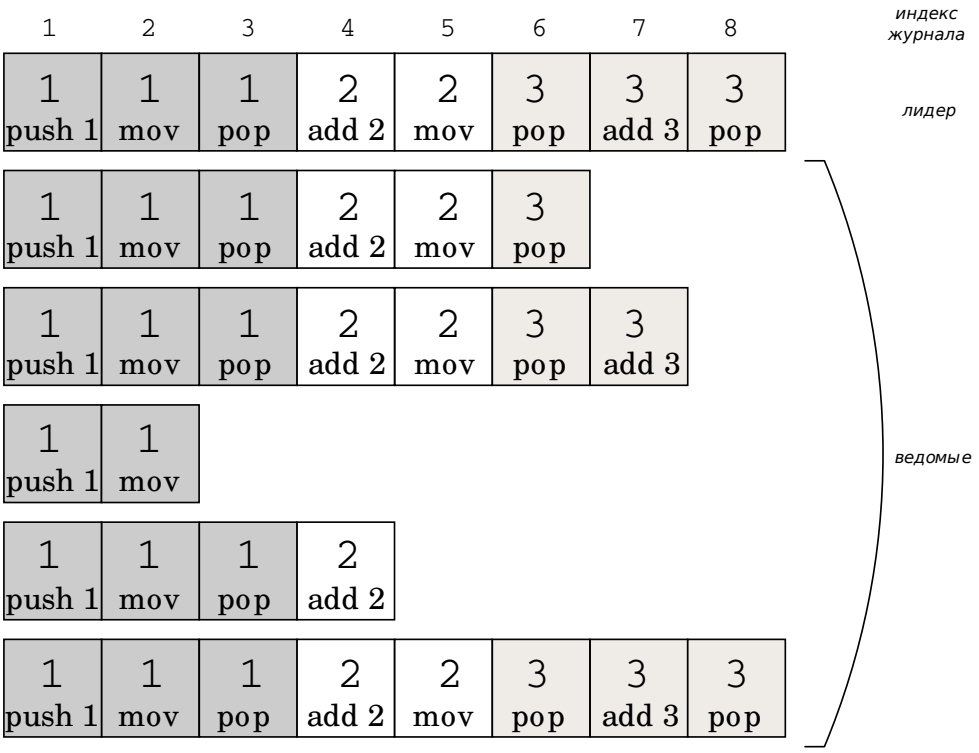


Рис. 4.4: Журналы состоят из записей, которые нумеруются последовательно. Каждая запись содержит номер эпохи, в которой он был создан и команду для конечного автомата.

свойство гарантируется простой проверкой согласованности в реализации процедуры *AppendEntries*. При отправке записи с помощью вызова *AppendEntries*, лидер добавляет индекс и номер эпохи записи, которая непосредственно предшествует новой записи. Если ведомый не находит запись в своем журнале с таким же индексом и эпохой, то он отказывается принимать новую запись. В результате, всякий раз, когда *AppendEntries* успешно завершается, лидер знает, что журнал ведомого идентичен его собственному журналу.

Во время ожидаемой нормальной работы, журналы лидера и ведомых остаются согласованными, поэтому в *AppendEntries* проверка согласованности никогда не закончится неудачно. Однако, в нетривиальных ситуациях, например таких, как отказ лидера, журнал может оказаться в несогласованном состоянии (старый лидер мог не успеть реплицировать

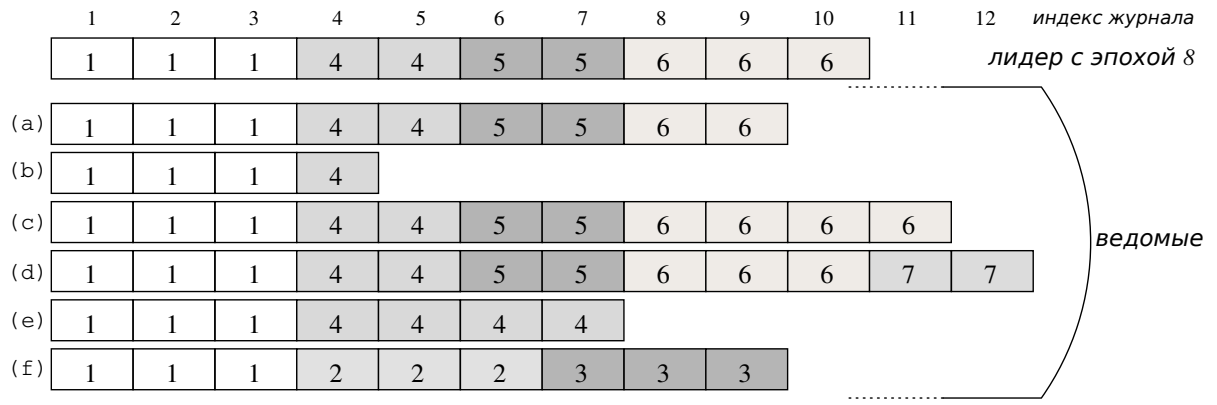


Рис. 4.5: Когда лидер (вверху) побеждает в выборах, то возможны какие-либо из сценариев (a-f) в журналах ведомых. Каждый блок представляет собой одну запись журнала, число внутри блока - номер эпохи соотвествующей записи. Ведомый может не иметь некоторых записей (a-b), может иметь дополнительные незафиксированные записи (c-d), или оба типа дефектов (e-f). Например, сценарий (f) может произойти, если сервер был лидером в эпохе 2 и добавлял несколько записей в свой журнале, но отказал прежде, чем успел зафиксировать какие-либо из них; затем перезапустился, стал лидером на эпоху 3 и добавил несколько записей в свой журнале; прежде, чем какая-либо из записей в эпохе 3 или 2 была зафиксированна, затем лидер снова отказал и не работал в течение нескольких эпох.

все записи в своем журнале). Эти несоответствия могут накапливатся в следствии серии отказов лидеров и ведомых. Рисунок 6 иллюстрирует ситуации, в которых журналы ведомых могут отличаться от нового лидера. Ведомый может не иметь записи, которые присутствуют у лидера (так как лидер не успел реплицировать все записи), он может иметь дополнительные записи, которые не присутствуют у лидера (так как старый лидер начал репликацию записи, но отказал во время выполнения этой операции), так же могут быть записи, которые отсуствуют в обоих серверах. Посторонних записей в журнале может быть несколько.

В Raft лидер обеспечивает непротиворечивость, заставляя журналы ведомых дублировать его журнал. Это означает, что конфликтующие записи в журналах ведомых будут перезаписаны с элементами из журнала лидера.

Чтобы согласовать журнал, лидер должен найти последнюю запись

в журнале, где два журнала совпадали, стереть все записи в журнале ведомого после этой точки, и отправить ведомому все записи лидера после этой точки. Все эти действия происходят во время проверки согласованности, выполняемой *AppendEntries*. Лидер хранит переменную *next_index* для каждого ведомого, она хранит индекс записи журнала лидера, которая будет отправлена этому ведомому следующей. Когда сервер впервые становится лидером, он инициализирует все значения *next_index* номером своей последней записи в своем журнале плюс один. Если журнал ведомого противоречит лидеру, проверка согласованности в процедуре *AppendEntries* откажет при следующем вызове. После отказа, лидер уменьшает *next_index* и повторяет *AppendEntries*. В конце концов значение *AppendEntries* достигнет точки, где лидер и ведомый имеют идентичные журналы. Когда это произойдет, проверка согласованности *AppendEntries* успешно выполнится, затем процедура удалит конфликтующие записи в журнале ведомого, затем добавит записи из журнала лидера (если таковые имеются). После успешного выполения процедуры *AppendEntries* журнал ведомого будет согласован с лидером, как минимум до конца текущей эпохи

С этим механизмом лидеру не нужно предпринимать никаких специальных действий для восстановления журнала, когда он приходит к власти. Ему достаточно начать обычную работу, и журналы автоматически будут согласованны при отказе проверки согласованности процедуры *AppendEntries*. Лидер никогда не перезаписывает или удаляет записи в своем журнале.

Этот механизм позволяет Raft может получать, реплицировать, и фиксировать новые записи, пока большинство из серверов работают.

5. Результаты исследований

На основе описаных алгоритмов Raft и Paxos, был написан симулятор, реализующий реплицированный конечный автомат (с распределенным журналом), описанный в главе 1. Создание симулятора было начато с моделирования узлов и и связей между ними, с добавлением возможности отключать/приостанавливать определенные узлы и имитировать проблемы в связи между ними: есть возможность дублировать сообщения, пропускать их или переупорядочивать. Также, симулятор поддерживает выполняет моделирование событий в реальном времени, т.е. с помощью механизма таймеров, предаставляемой ОС.

Основными компонентами симулятора являются:

- **События:** создание событий происходит в определенном узле, в определенный момент времени, который может привести к изменению состояния узла или генерации нескольких событий, которые происходят в последующие моменты времени, они могут возникнуть на другом узле (ввиду передачи сообщений между узлами);
- **Список событий для обработки:** в системе существует очередь с приоритетами для событий, которые должны быть обработаны симулятором;
- **Узел с состоянием:** представляет собой локальные данные для реализации алгоритма, связанного с каждого узлом и клиентом,

взаимодействующим с ним;

- **Часы:** монотонно возрастающая функция, где события являются мгновенными. Цель часов состоит исключительно в том, чтобы обеспечивать частичного порядка для событий в списке событий.

- **Недетерминированность:** генератор случайных чисел, требующийся алгоритму,также используется для моделирования задержки и потерь пакетов, отказов узлов.

Основной цикл симулятора перебирает события из списка событий. Каждое событие обрабатывается симулятором, с учетом текущего состояния связанного узлом, результат обработки возвращается в виде новых событий и нового состояния задействанного узла; Все новые события добавляются в список событий.

Реплицированный конечный автомат симулировал изменение нескольких целочисленных регистров, команды со стороны клиента позволяют задавать значения этих регистров, увеличивать иили уменьшать их на заданную величину.

5.1 Исследование поведения алгоритмов

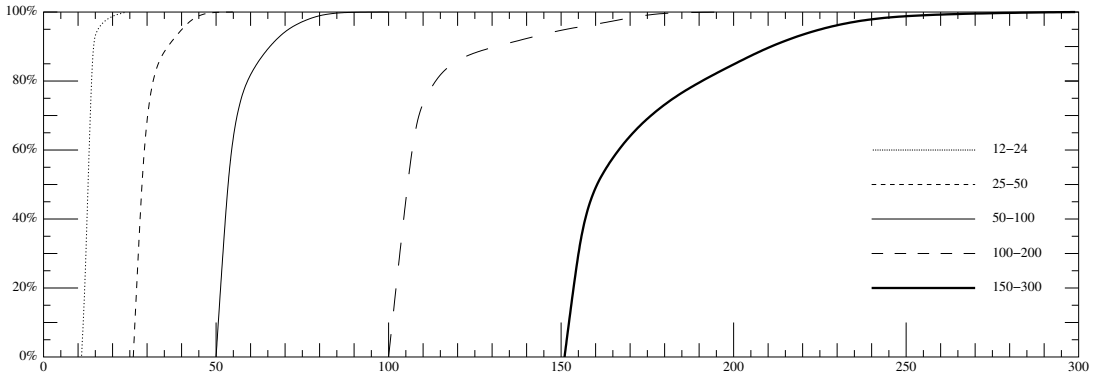
Всего рассматривалось три реализации:

- **Raft**, полностью соотвествующий описанию, данному выше.
- **Multi-Paxos**, где каждый узел является инициатором.
- **Multi-Paxos-With-Leader**, с избранным инициатором.

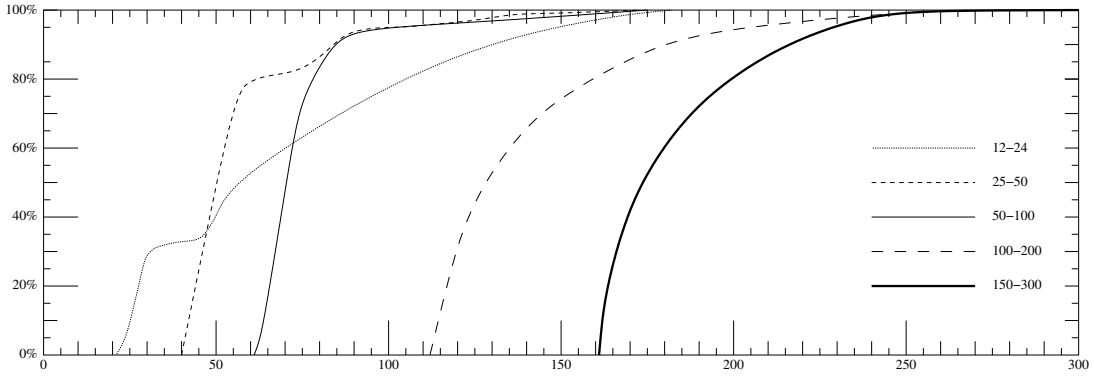
Для всех этих алгоритмов существуют как общие ситуации, как например, отказ определенного узла, так и характерные только для определенных алгоритмов. Там, где это было возможно, алгоритмы сравнивались в идентичных ситуациях, как например в замере производительности было одинаковое число узлов и число запросов от клиента в секунду.

5.2 Ситуация отказа лидера

Одна из первых рассматриваемых ситуаций для алгоритмов Raft и Multi-Paxos-With-Leader - отказ избранного лидера. В случае его отказа, в оставшихся узлах начинает отчитываться внутренний таймер, по срабатыванию которого узел инициирует голосование. Сама ситуация разбивается на две части: период до голосования, который продолжается до тех пор, пока не сработает один из таймеров оставших после отказа лидера узлов. Как видно из графиков 11 и 12, механизм выборов лидеров отрабатывает быстро и сравним по порядку с выбранном диапазоном срабатывания таймеров.



(a) Функция распределения времени (в миллисекундах) срабатывания таймеров оставшихся после остановки лидела узлов, при разных диапазонах срабатывания рандомизированных таймеров. Как видно из графика, процент сработавших узлов экспоненциально растет, за первую треть срабатывает более 80% узлов.



(b) Функция распределения времени окончания голосования, при разных диапазонах срабатывания рандомизированных таймеров. Как видно из графика, через 250 миллисекунд для после начала голосования, победитель будет определен с вероянтостью не меньше 99%.

Рис. 5.1: Функция распределения в случае отказа лидера. Размер кластера - 100 узлов.

5.3 Отказ обычного узла

Для всех трех алгоритмов возможна ситуация выхода из строя одного узлов. В случае Raft и Multi-Paxos-With-Leader выбирается узел, который не является лидером. Для проведения эксперемента, стороны клиента генерировался непрерывный набор команд, на каждую команду давалось 100 миллисекунд на обработку, в случае неудачной обработки симулятор увеличивал временную задержку междку командами кли-

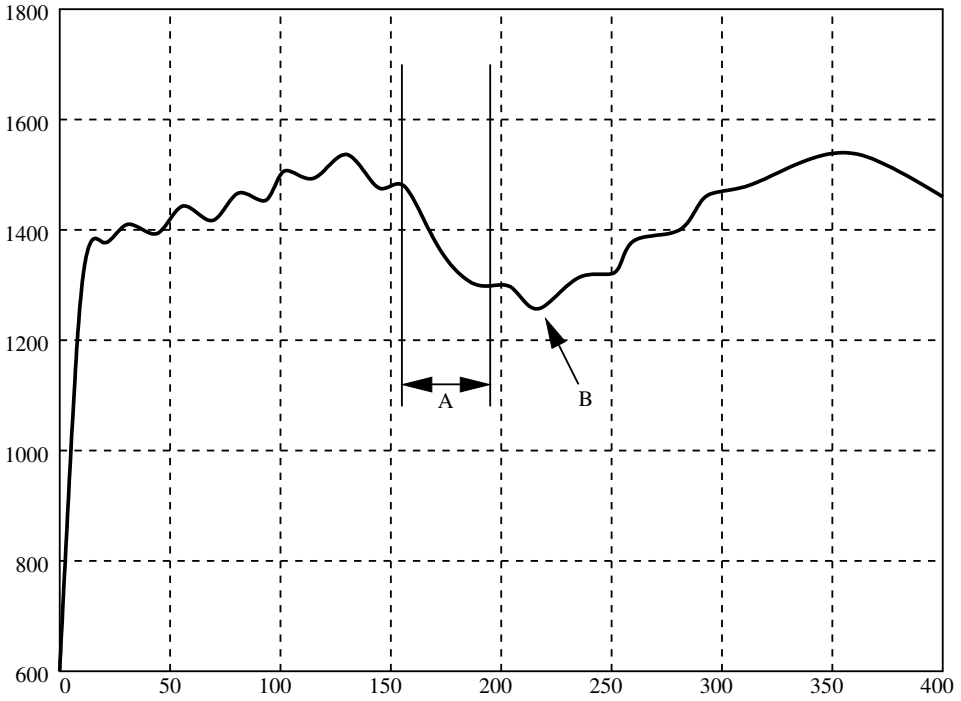


Рис. 5.2: Производительность кластера с отказом одного обычного узла, с алгоритмом Raft, где A - период отключения узла, видно уменьшение производительности. Пик B, после включения отказавшей ноды, вызван тем, что узел обновлял свой журнал, который устарел за время отключения. По вертикальной оси - время начала эксперемента, в секундах, по горизонтали - число операций в секунду.

ента, а если кластер успевал обработать команду, симулятор уменьшал задержку. Таким образом, в каждый момент времени измерялось максимальное число операций в секунду, обрабатываемое кластером из 10 узлов. В некоторый момент времени отключался один из узлов, имитируя отказ, а спустя некоторое время включался обратно.

Как видно из графиков, отказ обычного узла не вызывает сильного падения производительности для всех трех алгоритмов. Также, в Raft, после включения узла, наблюдается незначительное падение производительности, вызванное реплицацией журнала на этот узел.

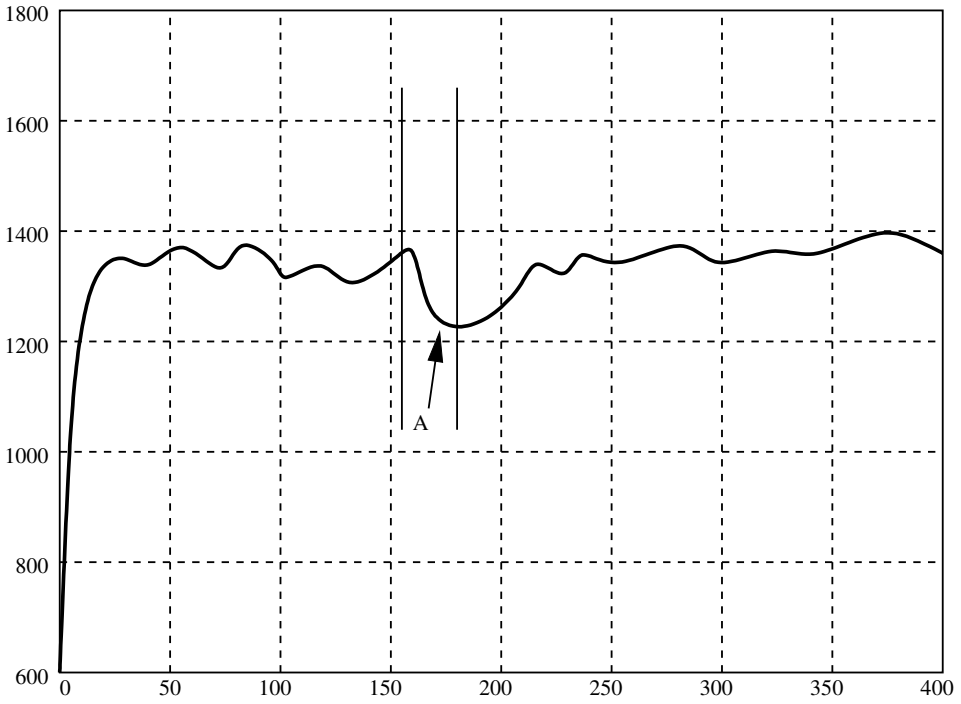


Рис. 5.3: Производительность кластера с отказом одного обычного узла, с алгоритмом Multi-Paxos-With-Leader, где A - период отключения узла. По вертикальной оси - время начала эксперемента, в секундах, по горизонтали - число операций в секунду.

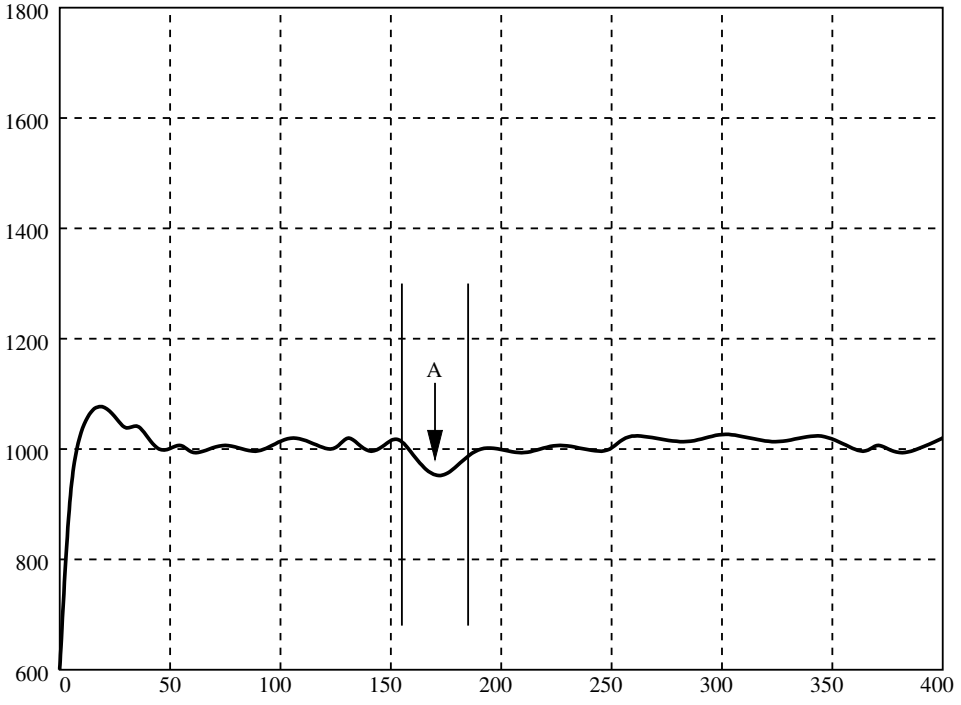


Рис. 5.4: Производительность кластера с отказом одного обычного узла, с алгоритмом Multi-Paxos, где A - период отключения узла. По вертикальной оси - время начала эксперемента, в секундах, по горизонтали - число операций в секунду.

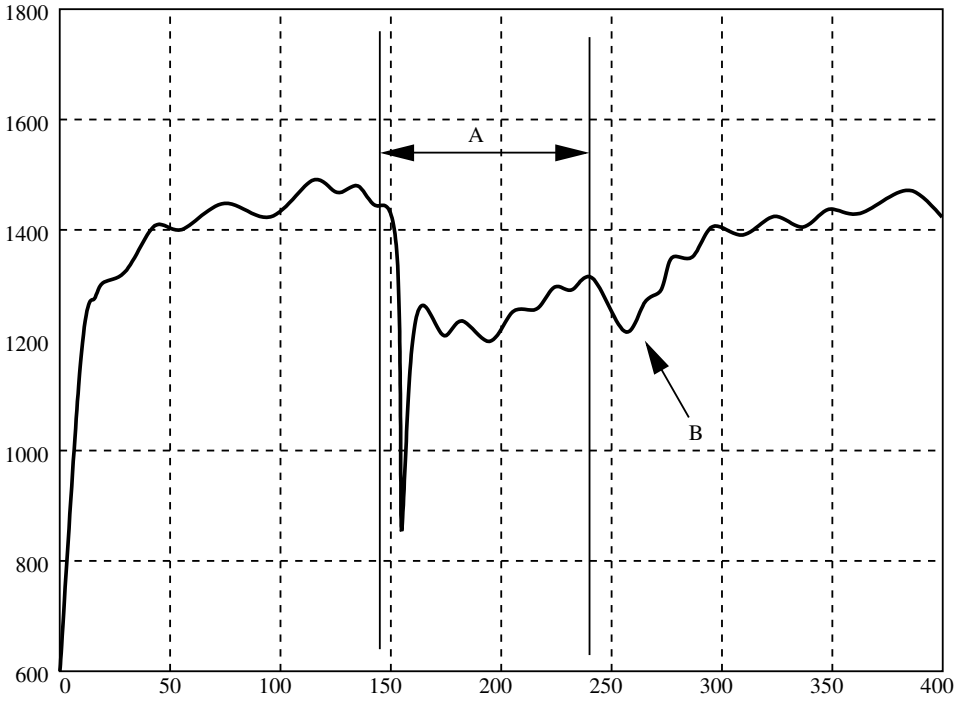


Рис. 5.5: Производительность кластера с отказом лидера, с алгоритмом Raft, где A - период отключения лидера. По вертикальной оси - время начала эксперемента, в секундах, по горизонтали - число операций в секунду. Пик в начале периода вызван отсуствием лидера и необходимостью голосования. Пик B, после включения отказавшей ноды, вызван тем, что узел обновлял свой журнал, который устарел за время отключения. По горизонтали - время начала эксперемента, в секундах, по вертикальной оси - число операций в секунду.

5.4 Отказ узла-лидера

Ситуация отключения лидера возможна только для Raft и Multi-Paxos-With-Leader. Для проведения эксперемента, стороны клиента генерировался непрерывный набор команд, на каждую команду давалось 100 миллисекунд на обработку, в случае неудачной обработки симулятор увеличивал временную задержку междку командами клиента, а если кластер успевал обработать команду, симулятор уменьшал задержку. Таким образом, в каждый момент времени измерялось максимальное число операций в секунду, обрабатываемое кластером из 10 узлов. В некоторый момент времени отключался лидерв, имитируя отказ, а спустя некоторое время включался обратно.

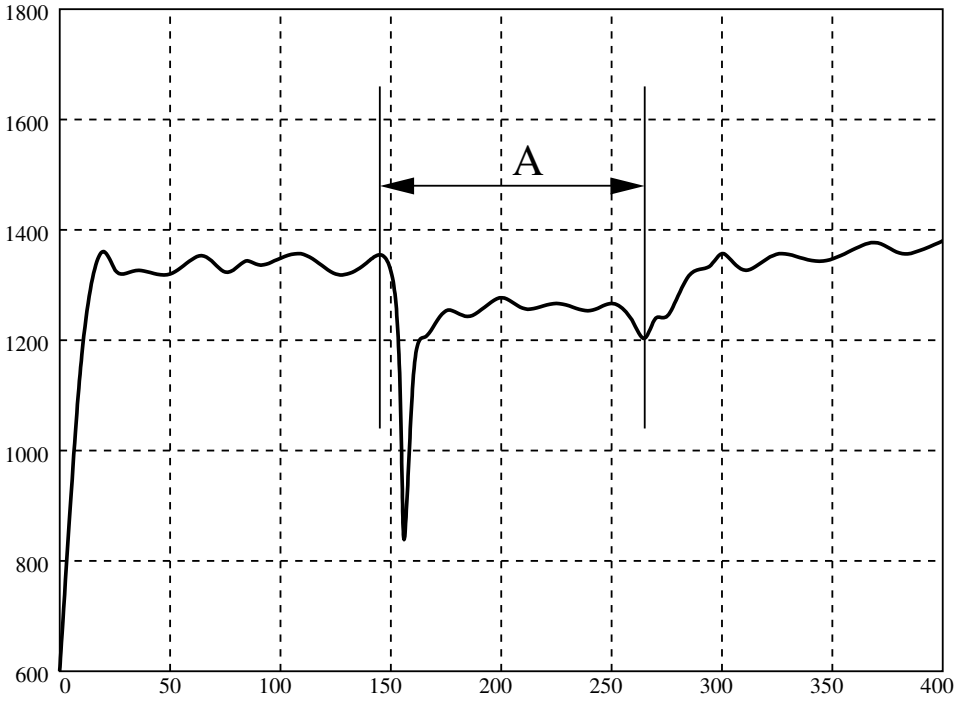


Рис. 5.6: Производительность кластера с отказом лидера, с алгоритмом Multi-Paxos-With-Leader, где A - период отключения лидера. По горизонтали - время начала эксперемента, в секундах, по вертикальной оси - число операций в секунду.

5.5 Производительность алгоритмов

В этом исследовании клиент генерировал непрерывный набор команд, на команду давалось 100 миллисекунд на обработку, в случае неудачной обработки симулятор увеличивал временную задержку между командами клиента, если кластер успевал обработать команду, симулятор уменьшал задержку. Таким образом в каждый момент времени измерялась производительность кластера. Как видно из графика, производительность алгоритмов Raft и Multi-Paxos-With-Leader приблизительно одинакова, а Multi-Paxos работает медленнее. Это вызванно тем, что число RPC-вызовов в последнем случае пропорцианально квадрату от числа узлов, каждый узел является инициатором, каждый инициатор рассылает сообщения каждому выборщику.

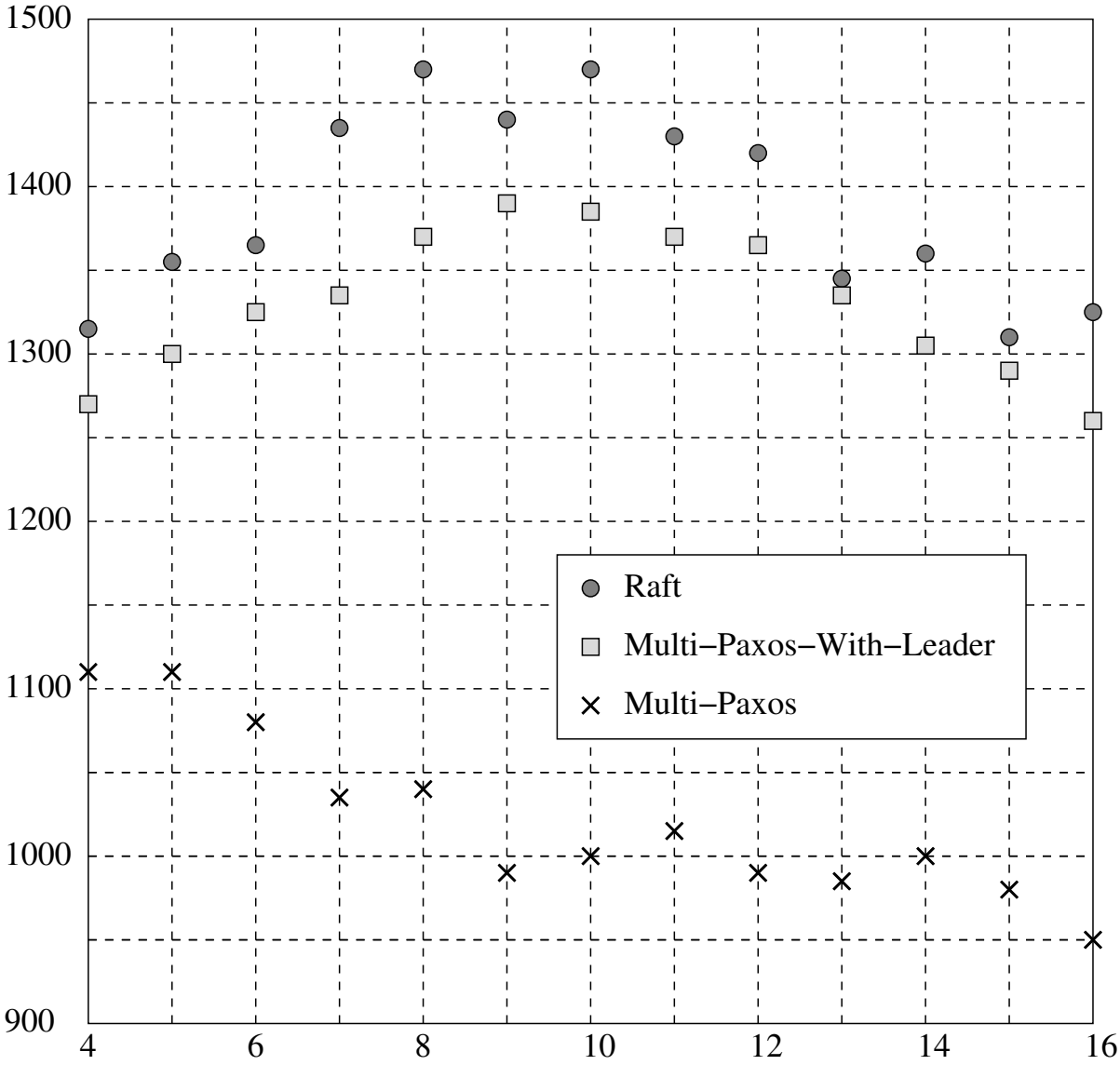


Рис. 5.7: Производительность кластера с различными алгоритмами консенсуса. По вертикальной оси - число операций в секунду, по горизонтали - размер кластера.

6. Заключение

1. Дано определение задачи решения консенсуса, и его роли в построении кластеров.
2. Разработан и реализован симулятор распределенной системы на основе реплицированного конечного автомата.
3. Приведены алгоритмы Raft и Paxos с их полным описанием, показон вывод алгоритма Multi-Paxos из Single-Paxos.
4. Показанно, что введение механизма избранного лидера улучшает производительно стандартного Multi-Paxos алгоритма, его производительность становится сравнимой с алгоритмом Raft.
5. Показанно, каковы временные порядки восстановления кластера при отказе выбранного лидера для алгоритмов Multi-Paxos (с лидером-инициатором), Raft.
6. Показанно, что отказ одного узла в кластере, который не является лидером (если это алгоритм с возможность проведения голосования, в противном случае просто любой узел), незначительно влияет на производительность системы вцелом.
7. Продемонстрированно, что в случае, когда каждый узел является и инициатором и выборщиком в Multi-Paxos, число сообщений пропорционально квадрату размеру кластера, что негативно сказывается на производительности кластера вцелом.

Список литературы

[1] M. J. Fischer, *The Consensus Problem in Unreliable Distributed Systems.* Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory, 1983

[2] B. W. Lampson, *How to build a highly available system using consensus.*. Distributed Algorithms, Eds. Springer-Verlag, 1996

[3] J. Gray, L. Lamport, *Consensus on Transaction Commit.* Microsoft Research, 2004

[4] J. Turek, D. Shasha, *The many faces of consensus in distributed systems.* J. Computer 25, 2, 1992

[5] M. Fisher, N. Lynch, M. Paterson, *Impossibility of Distributed Consensus with One Faulty Process.* J. ACM 32, 2, 1985

[6] S. Gilbert, N. Lynch, A. Nancy, *Perspectives on the CAP Theorem.* J. Computer 45, 2, 2012

[7] L. Lamport, M. Fischer, *Byzantine generals and transaction commit protocols.* Microsoft Research, 1982

[8] L. Lamport, *Time, clocks, and the ordering of events in a distributed system.* J. ACM 21, 7, 1978

[9] L. Lamport, *The part-time parliament.* J. ACM 16, 2, 1998

[10] К. Коротаев, *Система распределенного, масштабируемого и высоконадежного хранения данных для виртуальных машин.* Конференция HighLoad, 2012

[11] L. Lamport, *Paxos made simple.* J. ACM 32, 4, 2001

[12] L. Lamport, N. Lynch, *Generalized Consensus and Paxos.* Microsoft Research , 2005

[13] L. Lamport, *Fast Paxos*. DDistributed Computing 19, 2, 2005

[14] C. Tushar, G. Robert, R Joshua, *Paxos Made Live – An Engineering Perspective*. PODC '07: 26th ACM Symposium on Principles of Distributed Computing, 2007

[15] M Burrows, *The Chubby Lock Service for Loosely-Coupled Distributed Systems*. OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, 2006.

[16] R. Renesse, *Paxos made moderately complex*. Cornell University, 2012.

[17] F. B. Schneider, *Implementing fault-tolerant services using the state machine approach: a tutorial*. J. ACM 22, 4, 1990

[18] D. Ongaro, J. Ousterhout, *In Search of an Understandable Consensus Algorithm*. Stanford University, 2013

[19] H. Howard, M. Schwarzkopf, A. Madhavapeddy, J. Crowcroft, *Raft Refloated: Do We Have Consensus?*. SIGOPS Operating Systems Review, January 2015

[20] H.Howard, *ARC: Analysis of Raft Consensus*. University of Cambridge, Computer Laboratory, UCAM-CL-TR-857, 2014.