Министерство образования и науки Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к магистерской диссертации

«Разработка алгоритмов работы программно-конфигурируемой сети с распределенным инфраструктурным слоем»

Автор: Иванов Константин Алексеевич ________________
Направление подготовки (специальность): 01.04.02 Прикладная математика и информатика
Квалификация: Магистр
Руководитель: Ульянцев В.И., канд. техн. наук ________________

К защите допустить
Зав. кафедрой Васильев В.Н., докт. техн. наук, проф. ________________

«___» _________ 20___ г.

Санкт-Петербург, 2018 г.

Студент Иванов К.А. **Группа** M4239 **Кафедра** компьютерных технологий **Факультет** информационных технологий и программирования

Направленность (профиль), специализация Технологии проектирования и разработки программного обеспечения

Квалификационная работа выполнена с оценкой ____

Дата защиты «__» _________ 20___ г.

Секретарь ГЭК *Павлова О.Н.* Принято: «__» _________ 20___ г.

Листов хранения ____

Демонстрационных материалов/Чертежей хранения _____________________________________

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 6

1. Обзор существующих алгоритмов контрольного слоя и алгоритмов консенсуса 13
   - 1.1. Реализации программно-конфигурируемой сети 13
     - 1.1.1. Протокол OpenFlow 13
     - 1.1.2. Существующие платформы контроллера. 14
   - 1.2. Задача согласованной композиции политик 19
   - 1.3. Алгоритмы консенсуса 21
     - 1.3.1. Paxos 23
     - 1.3.2. Прочие алгоритмы консенсуса. 32
   - Выводы по главе 1 34
2. Решение задачи композиции и применения сетевых политик при распределенном слое управления 35
   - 2.1. Введение 35
   - 2.2. Оптимизация Generalized Paxos 35
     - 2.2.1. Постановка задачи для реплицированной машины состояний 35
     - 2.2.2. Хранение конфигурации сети 37
     - 2.2.3. Модификация Generalized Paxos 39
     - 2.2.4. Псевдокод модификации Generalized Paxos 46
     - 2.2.5. Доказательство работоспособности модификации 49
   - 2.3. Задача согласованной композиции политик 55
     - 2.3.1. Постановка задачи и модель сети 55
     - 2.3.2. Используемые команды 56
     - 2.3.3. Реализация установки политик на коммутаторы. 58
     - 2.3.4. Псевдокод контрольного слоя 60
     - 2.3.5. Доказательство выполнения свойств согласованной композиции политик 62
   - Выводы по главе 2 65
3. Реализация алгоритма применения сетевых политик при распределенном слое управления, тестирование и сравнение с аналогами 66
   - 3.1. Введение 66

3.2. Реализация алгоритма применения сетевых политик при распределенном слое управления ... 67
3.2.1. Компоненты приложения... 67
3.2.2. Протокол выбора лидера ... 70
3.3. Тестирование надежности протокола ... 73
3.4. Измерение производительности и сравнение с аналогами... 77
Выводы по главе 3 ... 82
ЗАКЛЮЧЕНИЕ... 83
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ ... 85

ВВЕДЕНИЕ

Компьютерная сеть — это система, обеспечивающая передачу данных между устройствами. Передаваемые данные группируются в так называемые *пакеты*. Отправленные пакеты могут передаваться через множество промежуточных узлов пока не достигнут устройство назначения. Чтобы эффективно передавать данные между устройствами (для чего должно учитываться множество факторов таких, как наличие каналов связи между промежуточными устройствами и их загруженность), требуется построение *конфигурации* сети, регламентирующей для каждого устройства направление дальнейшей передачи пакетов.

Сеть разделяют на два слоя в соответствии с функциями, выполняемыми входящими в них устройствами: *слой данных* отвечает за передачу пакетов по сети, и *слой управления* отвечает за формирование конфигурации. В традиционных сетях функции передачи пакетов и формирования конфигурации выполняются одними и теми же устройствами — маршрутизаторами. При этом задача построения эффективной конфигурации на уровне маршрутизаторов трудновыполнима, поскольку они не имеют информации о состоянии сети в целом.

Программно-конфигурируемые сети (Software defined networks, SDN) — это сети, в которых слой управления отделен от слоя данных. При этом слой управления состоит из одного или нескольких *контроллеров*, в то время как слой данных населен специализированными *коммутаторами*, архитектура такой сети представлена на рисунке 1. В программно-конфигурируемой сети конфигурация строится за счет установки *сетевых политик*. Каждый контроллер управляется так называемым *SDN приложением*, которое отслеживает состояние сети и на его основе определяет, какие сетевые политики устанавливать в сеть.

Сетевые политики бывают разных видов, базовые включают в себя следующие:

а) политики *пересылки* (forwarding), которые отвечают за передачу пакета дальше по сети промежуточными коммутаторами, например «отправлять TCP трафик от адреса 10.0.0.4 на коммутатор 7»;

б) политики мониторинга позволяют собирать статистику о проходящих через коммутатор пакетах, такую как количество пакетов UDP трафика

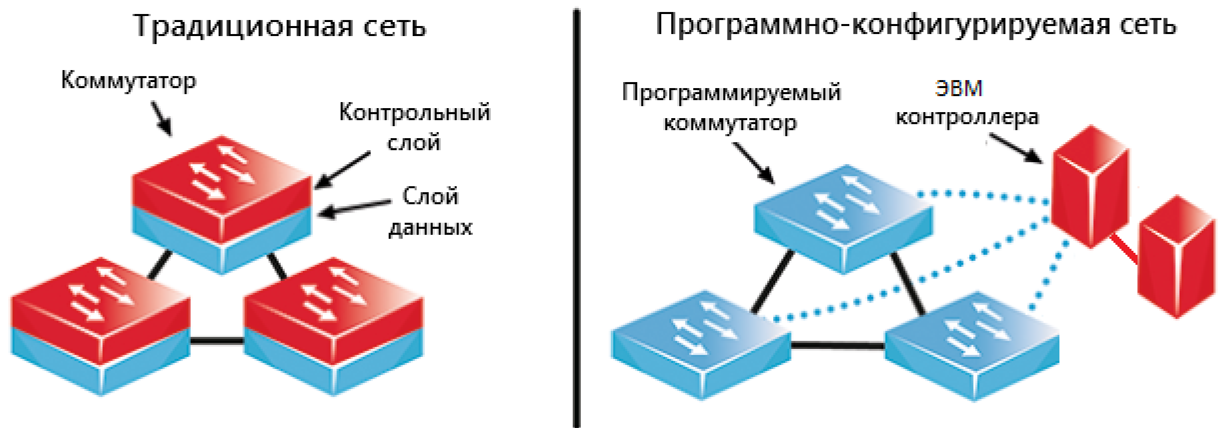


Рисунок 1 – В программно-конфигурируемой сети слой управления отделен от слоя данных. [1]

или количество пакетов, обработанных в соответствии с некоторой политикой пересылки;

в) политики обработки пакета используются для модификации заголовка пакета, например, для изменения адреса отправителя или адреса назначения.

На их основе можно задавать более сложные политики фильтрации трафика, балансировки нагрузки и другие. Каждая сетевая политика затрагивает один или несколько коммутаторов, может влиять на определенный вид трафика (например, TCP трафик с портом назначения 80) и имеет приоритет (в случае, один пакет регулируется несколькими политиками, приоритет позволяет разрешить конфликт «на месте»). Множество сетевых политик, установленных в сети, составляет *конфигурацию сети*.

В данной работе рассматриваются только политики пересылки. Формально, конфигурация в программно-конфигурируемой сети — это некоторый ориентированный граф $C = \langle S, E, M, Pr \rangle$, где множество вершин S есть множество коммутаторов в сети, множество ребер E указывает направление пересылки пакетов, $M(e)$ — множество сетевых пакетов, на которые оказывает влияние ребро e, $Pr(e)$ — приоритет ребра e. Пакеты передаются коммутаторами в соответствии с функцией $forward(C, s, p) = \mathrm{argmax}_{s' \colon (s,s') \in E,\, p \in M((s,s'))} Pr((s, s'))$. Когда коммутатор s получает некоторый пакет p, то в установленной на него конфигурации C он находит вершину $s_{next} = forward(C, s, p)$, и передает пакет p на коммутатор, соответствующий s_{next}. В случае, если не существует подходящего s_{next},

коммутатор отправляет пакет контроллеру, на что тот может сам выбрать s_{next} для данного пакета p и установить сетевую политику, регламентирующую обработку сетевых пакетов, аналогичных p. Сетевая политика — это некоторое множество ребер в данном графе $P \subseteq E$.

Конфигурация C называется противоречивой, если функция $forward(C, s, p)$ не однозначна для каких либо s и p. В частности, по протоколу OpenFlow установка такой конфигурации на коммутатор приводит к неопределённому поведению. Будем говорить, что политика P противоречит существующей конфигурации $C = \langle S, E, M, Pr \rangle$, если $C' = \langle S, E \cup P, M, Pr \rangle$ противоречива. Две политики P_1 и P_2 будем называть конфликтующими, если $\forall C = \langle S, E, M, Pr \rangle : \ C' = \langle S, E \cup P_1 \cup P_2, M, Pr \rangle$ — противоречива.

Одно из основных преимуществ программно-конфигурируемых сетей заключается в предоставлении приложениям глобального вида на сетевую инфраструктуру. В частности, это позволяет использовать более эффективные алгоритмы маршрутизации, ускорить внедрение инноваций в сетевые алгоритмы благодаря небольшому числу контроллеров, управляющих этими алгоритмами, а также сфокусировать сетевое оборудование на более высокопроизводительное выполнение узкого спектра задач по сравнению с традиционными сетями (где, например, коммутатор выполняет одновременно задачи передачи пакетов и маршрутизации).

Многие реализации контрольного слоя программно-конфигурируемой сети подразумевают наличие лишь одного контроллера, в этом случае речь идет о централизованном контрольном слое. Такие сети имеют известные недостатки:

- Плохая масштабируемость — с ростом сети растет требуемая скорость обработки политик контроллером, и в большой сети контроллер может оказаться неспособным справляться с нагрузкой.
- Единая точка отказа — в случае отказа контроллера управление сетью будет невозможно вплоть до восстановления контроллера.

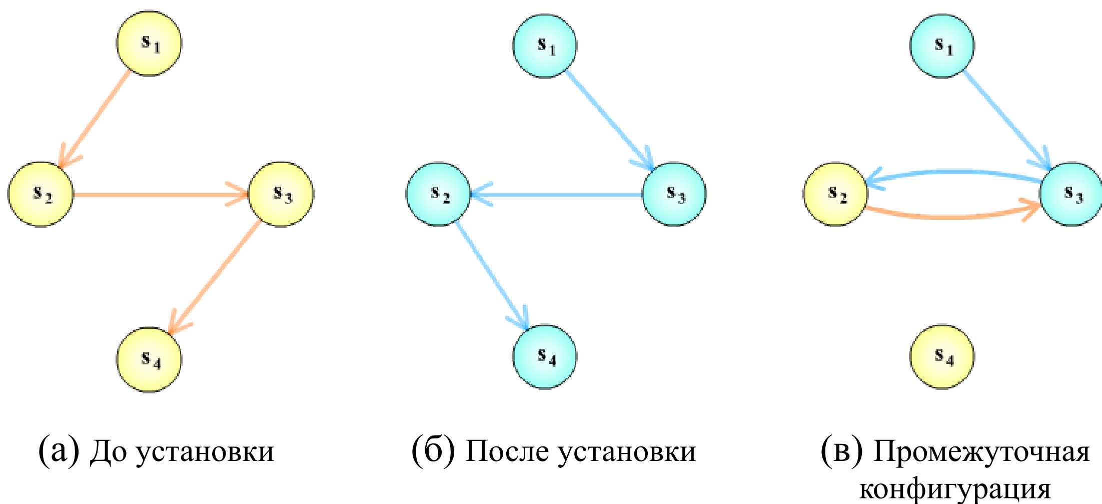
Для решения упомянутых проблем контрольный слой должен состоять из нескольких контролеров, другими словами, быть физически распределенным. При этом он также должен быть логически централизованным: с точки зрения пользователя и SDN приложения, все операции должны выглядеть так, словно сетью управляет лишь один контроллер.

Задача установки сетевой политики на коммутаторы не тривиальна даже при наличии одного контроллера в сети. Рассмотрим, какие гарантии может предоставлять контрольный слой (централизованный или распределенный) в порядке увеличения строгости:

— *Согласованность в конечном счете* (eventual consistency): после завершения установки политики (и уведомления об этом пользователя), если установка других политик не происходит, то в конечном счете все пакеты будут обрабатываться согласно новой конфигурации сети.
— *Строгая согласованность* (strong consistency): после завершения установки политики, пакеты обрабатываются согласно новой конфигурации; кроме того, работа с контрольным слоем выглядит так, как если бы он состоял из одного контроллера.
— *Сохранение конфигурации для пакетов* (per-packet consistency)[2]: при установке политики, каждый пакет обрабатывается согласно либо старой, либо новой конфигурации.
— *Сохранение конфигурации для потоков* (per-flow consistency): все пакеты в потоке регулируются согласно одной конфигурации.

В идеальном случае контрольный слой должен предоставлять гарантию по меньшей мере сохранения конфигурации для пакетов. В противном случае возможно временное возникновение *черных дыр* (black holes) и *циклов* (loops), то есть таких ситуаций, когда сетевые пакеты теряются или пересылаются по одним и тем же коммутаторам, не достигая точки назначения и перегружая каналы сети. Действительно, рассмотрим случай, когда контрольный слой предоставляет гарантию строгой согласованности. На рисунке 2 изображен маршрут передачи некоторого потока пакетов, модифицируемый сетевой политикой. Как видно, если обновление конфигурации устанавливается на коммутаторы в неподходящем порядке, возможна ситуация, когда в процессе обновления пакеты обрабатываются отчасти согласно старой, отчасти согласно новой конфигурации, приводя к появлению цикла.

При реализации распределенного слоя управления, все контроллеры прежде всего должны иметь доступ к общей конфигурации сети, для чего, как правило, используется *реплицированная машина состояний (РМС)* (replicated state machine). Реплицированная машина состояний — это алгоритм, принимающий на вход операции, и рассылающий их на несколько устройств,



(а) До установки (б) После установки (в) Промежуточная конфигурация

Рисунок 2 – Неконсистентная конфигурация сети при гарантии строгой согласованности

каждое из которых выполняет заданные операции; при этом он не теряет работоспособность при отказе одного или нескольких устройств. Процесс рассылки операций на остальные устройства будем называть *репликацией* (replication). Если контроллер одновременно является одним из серверов реплицированной машины состояний конфигурации (далее будем называть их *узлами* или *репликами*), внедрение конфигурации сети происходит быстрее [3].

Кроме того, задача достижения гарантии сохранения конфигурации для пакетов является нетривиальной даже в случае наличия одного контроллера в сети. Данная работа концентрируется на использовании оптимального протокола для хранения конфигурации сети на контроллерах и непосредственно на разработке протокола контрольного слоя, который был предоставлял гарантию сохранения конфигурации для пакетов.

В основе разрабатываемого контроллера лежит предположение о том, что конфликты между устанавливаемыми сетевыми политиками редки. Действительно, рассмотрим типичные SDN приложения, управляющие контрольным слоем.

— *Управление пользовательским трафиком* (user traffic control) указывает способ обработки трафика отдельных приложений, требующих доступа к сети. Политики этой категории, как правило, отвечают за разные виды пакетов.

— *Балансировка нагрузки* (load balancing) позволяет перераспределить трафик при высокой нагрузке на отдельные участки сети, однако применяется редко.
— *Мониторинг* (monitoring) позволяет собирать статистику о пакетах. Политики мониторинга беспрепятственно компонуются с любыми другими политиками.
— Базовая *маршрутизация* (routing) определяет путь перемещения пакетов по сети в случае отсутствия специальных политик.

В то же время, существующие на сегодняшний день протоколы контроллера основаны на централизованных алгоритмах хранения конфигурации, в которых все политики проходят через выделенный узел-координатор, и конфликты политик разрешаются на уровне этого узла. Такие протоколы используют от трех коммуникационных шагов для репликации политики.

Кроме упомянутых проблем, любая задача, поставленная для распределенной сети, должна учитывать модель сети, в которой сообщения, отправляемые одним устройством другому, могут приходить в произвольном порядке и с произвольными задержками, но, по умолчанию, в конечном счете доходят до пункта назначения. Кроме того, выделяют следующие основные модели отказов в сети в порядке увеличения строгости:

— *Отказ устройства* (Node crash) — исполнение действий на устройстве прекращается;
— *Отказ канала связи* (Link crash) — канал передачи сообщений перестает передавать сообщения в одну или в обе стороны;
— *Потери сообщений* (Omission) — доставка сообщений не гарантируется;
— *Византийская ошибка* (Byzantine failure) — устройство ведет себя произвольным образом; можно считать, что оно таким образом взаимодействует с другими устройствами, чтобы по возможности нарушить свойства, предоставляемые алгоритмом.

В данной работе предполагается модель, в которой возможны отказ одного или нескольких устройств и потеря сообщений.

В отличие от аналогичных работ, рассматривающих реализацию контроллера при распределенном слое управления, в данной работе впервые реализуется алгоритм контрольного слоя, предоставляющего гарантию сохранения конфигурации для пакетов. Также в работе задача

репликации сетевых политик впервые решается при помощи алгоритма, специализированного под случай редкого возникновения конфликтов политик. Приведенный в данной работе алгоритм использует Gen-Paxos — оптимизированную версию Generalized Paxos — в качестве протокола реплицированной машины состояний, в следствие чего демонстрирует производительность сравнимую с другими подобными алгоритмами при достижении более строгих гарантий. В частности, для внедрения неконфликтующих между собой политик в сеть, алгоритм требует меньшее число коммуникационных шагов и показывает бо́льшую пропускную способность. Протокол работает корректно и в случае установки конфликтующих политик, хотя оптимизация производительности в этой ситуации выходит за рамки данной работы.

Дальнейшее описание работы построено следующим образом. Глава 1 описывает существующие протоколы SDN контроллеров для сетей с централизованным и распределенным слоями, а также дает введение в протоколы консенсуса, описывает Paxos и существующие его модификации. В главе 2 приводится теоретическое описание алгоритма работы разрабатываемого контроллера, описываются использованные оптимизации протокола Generalized Paxos и приводится доказательство работоспособности полученного алгоритма. Глава 3 описывает архитектуру прототипа контроллера и технические детали. Наконец, в главе 4 описывается методология тестирования и оценки производительности алгоритма, приводятся результаты.

ГЛАВА 1. ОБЗОР СУЩЕСТВУЮЩИХ АЛГОРИТМОВ КОНТРОЛЬНОГО СЛОЯ И АЛГОРИТМОВ КОНСЕНСУСА

Данный раздел описывает наиболее известные на сегодняшний день технологии, связанные с программно-конфигурируемыми сетями. Далее, описывается задача консенсуса и одно из ее решений — алгоритм Paxos, также рассматриваются модификации Paxos. И наконец, рассматриваются существующие решения задачи разрешения конфликтов политик с использованием транзакционного подхода.

1.1. Реализации программно-конфигурируемой сети

1.1.1. Протокол OpenFlow

OpenFlow — это один из наиболее распространенных протоколов, описывающих взаимодействие коммутатора и контроллера в программно-конфигурируемой сети [4]. На момент написания работы существуют спецификации версий вплоть до версии 1.5.1, однако в работе будет использоваться будет протокол версии 1.0, поскольку его вполне достаточно для реализации прототипа.

Протокол OpenFlow содержит описание функций добавления, модификация и удаления правил обработки пакетов на коммутаторах. OpenFlow условно группирует пакеты в так называемые *потоки* (flows). Каждый поток содержит пакеты с одними и теми же адресами отправки и назначения, портами отправки и назначения и некоторыми другими совпадающими параметрами; другими словами, при неизменной конфигурации сети правила обработки одного пакета из потока распространяются на весь поток.

Коммутатор хранит данные правила в виде *таблиц потоков пакетов* (flow tables). В каждой строке такой таблицы хранится:

— Маска, указывающая, на какие пакеты влияет данное правило (например, TCP трафик, приходящий на 10.0.0.5);

— Действие, определяющее как пакеты должны обрабатываться. Примеры действий: перенаправление на конкретный коммутатор, перенаправление всем соседним коммутаторам, отбрасывание пакета. Доступна также возможность изменения полей пакета, например, MAC адреса, IP адреса или порта пункта назначения.

— Приоритет политики — если заголовок пакета подходит под несколько масок, выбирается правило с наибольшим приоритетом.

— Статистическая информация о потоке (количество пакетов, объем трафика, время обработки последнего пришедшего пакета).

Взаимодействие контроллера с коммутатором, начинающееся с сообщения ***PacketIn***, называют реактивным. Контролер также может проактивно запрашивать и изменять состояние коммутатора, если того требует SDN приложение.

Функциональности, описываемой протоколом OpenFlow, достаточно для того, чтобы на коммутаторах реализовать концентратор, обычный сетевой коммутатор, маршрутизатор, брандмауэр, балансировщик нагрузки, а также более сложные модули [4].

1.1.2. Существующие платформы контроллера.

Контроллеры для централизованного слоя управления. Существует множество платформ контроллера — программ, выполняющих роль SDN контроллера — реализующих протокол OpenFlow. Среди наиболее известных можно выделить следующие:

— NOX [5] — одна из первых платформ для написания приложений контроллера, на C++.

— POX — улучшенная платформа от производителей NOX, написана на Python.

— Beacon [6] — высокопроизводительная [7] платформа на Java. Использует технологию OSGi, которая позволяет подключать и отключать модули к системе без перезапуска виртуальной машины Java (JVM).

— Floodlight [8] — одна из наиболее активно разрабатываемых и поддерживаемых платформ на сегодняшний день, реализована на Java. Предоставляет практически всю функциональность через REST API и содержит множество полезных утилит. Аналогично Beacon, предоставляет возможность подключать и отключать модули «на лету», при этом не полагается на OSGi.

— Ryu [9] — еще одна платформа, написанная на Python. Она предоставляет богатый набор библиотек и примеров приложений, а также подробную документацию.

— Opendaylight [10] — платформа на Java с открытым исходным кодом, поддерживает микросервисную архитектуру. Также поддерживает OSGi.

Упомянутые фреймворки предоставляют возможность работы с централизованным контрольным слоем.

Контроллеры для распределенного слоя управления.

Onix — это платформа для реализации слоя управления программно-конфигурируемой сети. Она предоставляет обобщенный интерфейс, позволяющий оперировать сетью как единым целым, а также примитивы для манипуляции распределенным состоянием (например, распределенной хеш-таблицей) [11].

Помимо простого и обобщенного интерфейса, Onix концентрируется на масштабируемости, поддерживая возможность наличия множества контроллеров в сети. Для этого контроллеры отслеживают состояние сети и хранят его в так называемой *базе информации о сети* (Network Information Base, NIB). NIB может быть *секционирована* (partitioned), при этом контролер будет хранить только часть NIB. Если каждый коммутатор связан непосредственно лишь с одним контроллером, то каждый контроллер может хранить только ту часть NIB, которая относится к дочерним коммутаторам. Контроллеры также предоставляют доступ к своим частям NIB, позволяя агрегировать информацию о сети с разных контролеров при необходимости.

В случае распределенного хранения NIB, Onix предоставляет два вида гарантий согласованности по выбору.

— Сильная согласованность — с точки зрения пользователя и SDN контроллера установка политики выглядит так, как если бы сетью управлял один контроллер. В Onix строгая согласованность достигается через механизм распределенных блокировок. Для применения сетевой политики контроллер берет глобальную блокировку, рассылает политику другим контролерам, затем контроллеры устанавливают политику на коммутаторы, и блокировка отпускается.

— Согласованность в конечном счете — в отсутствие новых сетевых политик, устанавливаемая политика рано или поздно будет установлена. Используется блокировка уровня контроллера. При использовании такой гарантии могут возникать конфликты между вводимыми изменениями.

Onix подразумевает, что приложение контроллера, используемое поверх Onix, задает логику обнаружения и разрешения конфликтов.

Onix использует Zookeper [12] для координации контроллеров, а именно: для решения задачи репликации политик, обнаружения падения узлов. Для хранения NIB используется отдельная распределенная база данных в виду ограничений на размер объектов в Zookeper и ради удобства обращения к данным напрямую.

Таким образом, Onix является удобной платформой для разработки собственных приложений для контроллера при распределенном слое управления. Однако, пользователь платформы вынужден выбирать между низкой производительностью (при использовании сильной согласованности) и необходимостью указания способа разрешения конфликтов сетевых политик в случае их возникновения (в случае выбора согласованности в конечном счете).

Упомянутый ранее Zookeeper — это алгоритм, основанный на протоколе ZAB [13], использующий схему *управления копированием* (primary-backup): контроллер-лидер выполняет запрошенные операции и уведомляет остальные контроллеры о каждом изменении хранимого состояния. В случае отказа лидера, остальные контроллеры выбирают нового лидера, тот восстанавливает состояние и алгоритм продолжает работу в штатном режиме. Zookeeper требует использования FIFO каналов (где отправляемые сообщения приходят в том порядке, в каком были отправлены), при этом предоставляя гарантию применения операций в FIFO порядке (то есть, операции исполняются в том порядке, в каком были предложены).

ONOS расшифровывается как *Открытая Сетевая Операционная Система* (Open Network Operating System) и является очередной платформой слоя управления SDN [14]. Создатели ONOS фокусировались на достижении производительности, масштабируемости и высоких показателях доступности.

В первом прототипе ONOS хранит вид на сеть в виде графа. Для хранения графа используется распределенное хранилище графов Titan с использованием *хранилища типа ключ-значение* (key-value store) Casandra [15]. Поскольку Casandra предоставляет гарантию согласованности в конечном счете, то и состояние сети, поддерживаемое ONOS, является согласованным в конечном счете.

Данный прототип демонстрировал плохие показатели производительности. По этой причине была разработана вторая версия прототипа, где используется реализация распределенного хранилища графа на основе Blueprints [16] поверх RAMCloud [17]. RAMCloud — это распределенное хранилище типа ключ-значение, хранящее данные целиком в памяти устройств, благодаря чему достигаются низкие задержки при удаленных чтении и записи (порядка десятков микросекунд). Для обеспечения корректности оно требует в любой момент времени работоспособности хотя бы кворума узлов.

Каждому коммутатору ONOS сопоставляет один контроллер, который считается *управляющим* (master) данным коммутатором. При изменении конфигурации сети, контроллер ответственен за обновление подконтрольных ему коммутаторов, а также должен участвовать в обновлении глобальной информации о сети. При падении контроллера, остальные контроллеры выбирают мастера для каждого из коммутаторов, не имеющего мастера. Поскольку для каждого коммутатора должен быть выбран ровно один мастер, выбор производится на основе алгоритма консенсуса; в частности, для этой цели ONOS использует сервис ZooKeeper.

Итого, рассмотренные платформы контроллера либо подразумевают централизованное управление сетью, либо предоставляют недостаточную поддержку согласованности, требуя от оператора нахождения компромисса между высокой производительностью и автоматическим предотвращением конфликтов сетевых политик.

Протоколы контроллера, основанные на хранении конфигурации с более сильными гарантиями.

Программно-транзакционная сеть. Ранее был предложен подход на так называемых *программно-транзакционных сетях* (Software Transactional Network, STN) [18, 19]. При данном подходе запрос на установку политики может завершится успешно и вернуть ***ack*** либо обнаружить что правило не совместимо с существующей конфигурацией и вернуть ***nack***, не внеся никаких изменений в конфигурацию.

В работах, описывающих STN, предлагается разрешение конфликтов политик непосредственно на слое данных. Для этого коммутаторы должны

поддерживать атомарную операцию *read-modify-write*. Если контроллеры посылают конфликтующие правила на коммутатор, то он отвечает отказом на установку правил, конфликтующих с уже установленными. Также работы вводят методику *тегирования* (tagging) пакетов. Каждой композиции политик ставится в соответствие некоторый тег, и сетевые пакеты помечаются тегом, указывающим, в соответствии с какой политикой обрабатывать данный пакет. Данная техника позволяет достичь гарантии сохранения конфигурации для пакетов. При этом в следствие произвольного порядка установки политик на коммутаторы возможно возникновение «черных дыр», если коммутатор получает пакет с неизвестным тегом. Для борьбы с этой проблемой используется метод двухфазного обновления [2]: на первой фазе политика устанавливается на *внутренние порты* (internal ports) — внутренние порты используются для общения коммутаторов между собой; на второй фазе политика устанавливается на *входные порты* (ingress ports), через которые пакеты попадают в сеть. Таким образом, пакеты с новым тегом могут появится не раньше начала второй фазы, и все коммутаторы будут иметь политику по их обработке.

BFT-Light концентрируется на достижении строгой согласованности [20], для реализации данного протокола используется Floodlight контроллер с BFT-SMaRt алгоритмом в качестве реплицированной машины состояний. BFT-SMaRt основан на алгоритме PBFT [21] и предоставляет гарантию работы при возможных *незлоумышленных* (nonmalicious) византийских ошибках, то есть способен работать при внесении каналом связи случайных изменений в передаваемые сообщения. Алгоритм BFT-SMaRt требует три коммуникационных шага для репликации команды.

Ravana [22] также использует для хранения конфигурации алгоритм, предоставляющий гарантию строгой согласованности — Zookeeper. В работе, описывающей данный контроллер, рассматриваются возможные проблемы, возникающие при отказе контроллера, в частности: отказ контроллера в процессе получения уведомления от коммутатора о появлении нового потока пакетов, отказ контроллера в процессе репликации сетевой политики, отказ контроллера в процессе установки политики на коммутатор; затем приводится

протокол, позволяющий их разрешить. В архитектуре Ravana имеется выделенный главный контроллер, который взаимодействует с коммутаторами, в то время как остальные контроллеры используются лишь в случаях отказа главного контроллера.

1.2. Задача согласованной композиции политик

В работе Канини, Кузнецова и др.[23] описываются некоторые свойства, которые могут требоваться от распределенной системы, а также ставится задача *согласованной композиции политик, СКП* (consistent policy composition), что позволяет формально судить о гарантиях, предоставляемых контрольным слоем.

Говоря неформально, СКП интерфейс принимает конкурентные запросы на установку политик и применяет политики таким образом, что запросы влияют на пакеты слоя данных, как если бы запросы были осуществлены последовательно. Абстракция предоставляет транзакционный интерфейс, в котором каждый из запросов может быть успешно осуществлен, после чего новые пакеты следуют согласно новой конфигурации, или отменен, если политика не может быть скомпонована с уже установленной конфигурацией, в таком случае политика не затрагивает ни один пакет. С точки зрения прогресса, если несколько конкурентно устанавливаемых политик конфликтуют между собой, но не с существующей конфигурацией, то хотя бы одна из политик будет применена. Формально, каждый контроллер p_i принимает запросы $apply_i(\pi)$, где π — это политика, и возвращает ack_i в случае приема политики или $nack_i$ в случае отказа.

Для заданного выполнения алгоритма *история* — это последовательность внешне-наблюдаемых событий: *появление пакетов в сети* (inject), *пересылка пакетов* (forwarding) коммутаторами, осуществление запросов на установку политики и получение ответа на запрос. Пусть ev — событие появления некоторого пакета в сети. Будем обозначать как $\rho_{ev,H}$ последовательность коммутаторов, которые данный пакет посетил в истории H, и называть эту последовательность следом пакета.

Известно, что в распределенных системах отсутствует понятие общего времени [24], поэтому нельзя говорить о полной упорядоченности событий в истории. Отношение частичного порядка на событиях в данной истории H будет обозначаться $<_H$. Будем говорить, что запрос req *предшествует* запросу

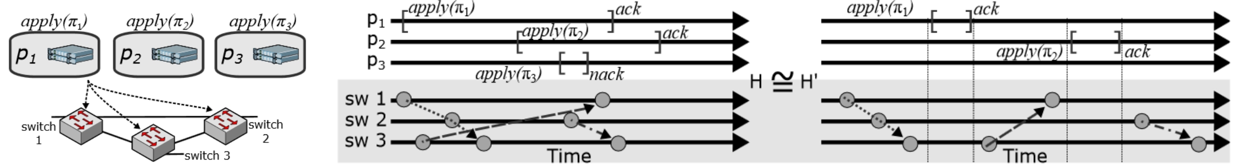


Рисунок 3 – Пример линеаризации истории. [23]

req' в истории H, и записывать $req <_H req'$, если ответ на запрос req появляется в истории H прежде инициации запроса req'. Если ни один из запросов не предшествует другому, будем говорить, что запросы *конкурентны*. Аналогично будем говорить, что событие появления пакета в сети ev *предшествует* запросу req, и обозначать $ev <_H req$, если ev появляется в истории H раньше инициации req; симметричным образом определяется $req <_H ev$. Два события появления пакета ev и ev' связаны отношением $ev <_H ev'$, если ev появляется в истории H раньше ev'. Событие появления пакета ev конкурирует с запросом req, если $ev \not<_H req$ и $req \not<_H ev$. История H называется *последовательной*, если никакие два запроса не конкурируют и никакое событие появления пакета не конкурирует с запросом.

Пусть $H|_{p_i}$ обозначает локальную историю контроллера p_i, то есть множество событий H, содержащее лишь события p_i. Будем предполагать, что каждый контроллер *корректно построен* (well-formed): каждая локальная история $H|_{p_i}$ последовательна, то есть никакой контроллер не принимает новый запрос до выдачи ответа на предыдущий. Запрос, вызванный p_i *завершен*, если за ним следует соответствующий ответ (ack_i или $nack_i$); в противном случае он называется *незавершенным*. История H *завершена*, если каждый запрос завершен в H. *Завершение* истории H — это такая завершенная история история H', которая отличается от H лишь тем, что каждый незавершенный запрос в H завершен при помощи ack (это может быть нужно, если запрос уже повлиял на пакеты) или $nack$, вставленного где-либо после инициации запроса.

Две истории H и H' эквивалентны, если H и H' содержат одинаковое множество событий, для каждого p_i выполнено $H|_{p_i} = H'|_{p_i}$, и для каждого события появления пакета ev в H и H' верно $\rho_{ev,H} = \rho_{ev,H'}$.

Последовательно согласованная история. Последовательная завершенная история H называется *законной* (legal), если выполняются два свойства:

а) Политика применена в H тогда и только тогда, когда она не конфликтует с множеством политик, ранее примененных в H;
б) Для каждого события появления пакета в сети ev в H, след $\rho_{ev,H}$ согласуется с композицией всех принятых политик, которые предшествуют ev в H.

Определение 1. Будем говорить, что полная история H *последовательно согласована* (sequentially composable) или *линеаризуема* (linearizable), если существует законная последовательная история S такая, что

а) H и S эквивалентны;
б) $<_H \subseteq <_S$.

Говоря неформально, определение подразумевает, что трафик в H обрабатывается так, как если бы все запросы были применены атомарно и каждый появившийся в сети пакет обрабатывался мгновенно. Свойство законности в данном случае лишь требует, чтобы примененные запросы оказывали влияние на трафик. Более того, эквивалентная последовательная история S должна учитывать порядок, в котором происходят неконкурирующие запросы и появления пакетов в H. На рисунке 3 слева представлена некоторая история, и справа — ее линеаризованный эквивалент.

Определение 2. Будем говорить, что алгоритм решает задачу согласованной композиции политик, если для каждой его истории H существует завершение H' такое, что выполняется:

а) Согласованность: H' последовательно согласованно.
б) Завершение: в конечном счете, каждый корректный контроллер p_i, который принимает запрос $apply_i(\pi)$, возвращает ack_i или $nack_i$ в H.

Таким образом, предоставляется семантика «все-или-ничего»: политика, независимо от поведения контроллера, который ее устанавливает, либо в конечном счете возымеет эффект, либо не повлияет ни на один пакет. Рассмотренные определения позволят формально привести свойства, требуемые от контрольного слоя в нашей работе.

1.3. Алгоритмы консенсуса

Задача *консенсуса* ставится следующим образом: пусть есть множество процессов P, каждый процесс предлагает некоторое *значение*. Затем процессы выбирают одно из предложенных значений. Алгоритм, реализующий консенсус, должен удовлетворять следующим свойствам:

— *Согласованность* (Consistency) — все корректные процессы должны выбрать одно и то же значение;
— *Валидность* (Validity) — выбранное значение является значением, предложенным одним из процессов;
— *Завершение* (Termination) (также иногда называется *живучестью* (liveness)) — алгоритм в конечном счете завершается.

Было показано, что задача консенсуса в распределенной системе не разрешима при возможности отказа даже одного процесса [25]. Более того, CAP-теорема [26] утверждает, что для любого распределенного алгоритма невозможно иметь следующие три свойства одновременно:

— Устойчивость к разделению сети — расщеплению сети на изолированные подсети;
— Согласованность — (здесь в несколько ином значении) история операций является линеаризуемой, то есть операции осуществляются так, как если бы были атомарны.
— Доступность — на любой запрос к распределенной системе в конечном счете приходит корректный ответ.

Таким образом, при создании алгоритма для работы в распределенной системе придется отказаться от одного из этих свойств. В данной работе пренебрегается доступностью в пользу двух других свойств.

Прежде чем приступать к описанию существующих алгоритмов консенсуса, введем еще несколько определений.

Определение 3. Будем говорить, что алгоритм требует m коммуникационных шагов, если существует цепочка сообщений $M_1, M_2, \dots M_m$, в которой M_{i+1} может быть отправлено только после M_i, и данная цепочка - минимальная такая цепочка по длине.

Кворумы. При задании протоколов голосования или репликации часто используют так называемые *кворумы* [27].

Определение 4. Пусть имеется множество S. *Системой кворумов* называют множество $Q \subseteq 2^S$ со следующими свойствами:

а) $Q \neq \emptyset$;
б) $\forall A,\, B \in Q:\; A \cap B \neq \emptyset$.

Элементы множества Q называют *кворумами*.

Среди наиболее известных систем кворумов — кворум большинства. Он задается следующим образом: $Q = \{A \subseteq S \mid |A| > |S|/2\}$. Если $|S| = N$, то при такой системе кворумов все кворумы имеют размер $\lceil N/2 \rceil$.

1.3.1. Paxos

Classic Paxos. Paxos [28, 29] представляет собой протокол, реализующий задачу консенсуса при возможности потери сообщений. Paxos жертвует доступностью, будучи устойчивым к разделениям сети и сохраняя согласованность.

В Paxos процессам назначаются условные роли:

— *Инициатор* (proposer) производит запрос к распределенной системе и ожидает ответа на запрос. Ожидание может быть активным, то есть, инициатор должен периодически предлагать значение на случай потери сообщения или конфликтов.

— *Лидер* (leader) координирует действие протокола, принимая значения от инициатора и реплицируя их между избирателями.

— *Избиратели* (acceptors) служат в качестве отказо-устойчивой памяти алгоритма. Когда значение записано кворумом избирателей, оно считается зафиксированным.

— *Исполнители* (learners) отслеживают фиксируемые значения. При появлении нового такого значения исполнитель может, например, отправить ответ клиенту.

В системе возможно наличие нескольких лидеров, то есть процессы считают разные процессы лидерами. При этом выполнение протокола остается корректным, но живучесть не гарантируется.

В *классическом* (classic) Paxos каждый запуск протокола предназначен для достижения консенсуса относительно одного значения. Протокол выполняется в несколько *раундов* (ballots, rounds). Каждый раунд имеет идентификатор; на идентификаторах должна быть задана операция сравнения. Каждый раунд состоит из следующих фаз:

Фаза 1a: Подготовка. Лидер начинает новый раунд с идентификатором больше, чем у предыдущих раундов. Он посылает ***Prepare*** сообщение с номером нового раунда всем избирателям.

Фаза 1b: Обещание. Исполняется избирателем, если номер раунда в ***Prepare*** больше любого ранее слышанного номера раунда. Избиратель отвечает сообщением ***Promise***, тем самым давая обещание в будущем игнорировать сообщения с меньшим номером раунда. Возможно применить оптимизацию: если избиратель ранее получал сообщение с бóльшим номером раунда, то он может ответить отказом, сообщая наибольший известный номер раунда; однако для корректности протокола это не требуется.

Фаза 2a: Предложение. Если лидер получил сообщение ***Promise*** от кворума избирателей, то начинается вторая фаза. Лидер должен выбрать значение. Если избиратели ранее получали предложения, то они должны передать их в сообщениях ***Promise***; если хотя бы один избиратель сообщил о предложенном значении, лидер должен выбрать значение, сообщенное в раунде с наибольшим номером (такое значение будет одно и то же для всех избирателей). Если избиратели не сообщали о предложениях, лидер выбирает любое значение из предложенных. Затем лидер рассылает сообщение ***AcceptRequest***, содержащее выбранное значение, всем избирателям.

Фаза 2b: Принятие. Выполняется избирателем, если только он не давал обещание игнорировать сообщения с номер раунда меньше или равным номеру раунда в ***AcceptRequest***. Избиратель сохраняет значение, полученное в ***AcceptRequest***, и рассылает сообщение ***Accepted*** инициатору и исполнителям. При получении ***Accepted*** от кворума избирателей, исполнитель считает значение зафиксированным и уведомляет заинтересованные процессы о новом значении.

Раунд завершается неудачно, если в фазе **1b** лидер, или в фазе **2b** хотя бы один из исполнителей, не получили сообщения хотя бы от кворума избирателей. В таком случае инициируется очередной раунд алгоритма. В случаях, когда требуется решить задачу консенсуса относительно нескольких значений, запускается несколько экземпляров алгоритма, возможно параллельно. Псевдокод данного алгоритма приведен в листинге 1.

Листинг 1 – Псевдокод Classic Paxos.

```
propose(value):
    Make leader remember value
```

```
Periodically execute
leader.phase1a():
    bal <- bal + 1
    broadcast Prepare(bal) to acceptors

On Prepare(bal)
acceptor.phase1b:
    Ignore further messages from ballots below bal
    send Promised(bal, value) to leader

On Promised(bal, value)
leader.phase2a:
    if value != None:
        toPropose <- value
    On quorum messages do:
        if toPropose = None:
            toPropose <- getAnyProposed()
        broadcast AcceptRequest(bal, toPropose) to acceptors

On AcceptRequest(bal, value)
acceptor.phase2b:
    chosen <- value
    broadcast Accepted(value) to learners

On Accepted(value)
learner.learn(value):
    When received value from quorum of acceptors:
        Fix(value)
```

Допустим, все процессы корректны, сообщения доставляются без задержек и существует единственный признанный процессами лидер (в таких случаях будем говорить о *запуске в корректной среде* ()). В таких условиях, алгоритм требует $\Omega(N^2)$ сообщений и три коммуникационных шага для завершения; действительно, сообщения из первой фазы не зависят от предлагаемых значений и могут быть выполнены заранее, остальные сообщения составляют цепочку **инициатор** $\to$ **лидер** $\to$ **избиратель** $\to$ **исполнитель**. Существует техника *централизованного уведомления исполнителей*: фаза ***2b*** разделяется так, что избиратели отправляют сообщение одному исполнителю, а этот исполнитель рассылает сообщение остальным

исполнителям; в таком случае требуется порядка $5 \cdot N$ сообщений и четыре коммуникационных шага.

Отметим здесь, что когда мы будем говорить, что протокол требует *порядка* $f(N)$ сообщений, будем подразумевать, что требуется $f(N) + O(m)$ сообщений, где N — число участвующих в протоколе процессов, m — число фаз алгоритма. Дать точную оценку может быть затруднительно, поскольку на каждой фазе алгоритма процесс, рассылающий сообщения, может отправлять или не отправлять сообщение в том числе и самому себе; в первом случае можно сэкономить на отправке одного сообщения. Если в одной из фаз алгоритма более чем один процесс рассылает сообщения, погрешность может быть больше, чем $O(m)$, в таком случае мы будем иначе формулировать оценку.

Fast Paxos В отличие от классического Paxos, в Быстром Paxos предлагаемые значения отправляются напрямую избирателям, благодаря чему алгоритм выполняется в два коммуникационных шага [30]. Также, в нем каждый раунд имеет свое семейство кворумов. В этом алгоритме раунд состоит из следующих фаз:

Фазы 1a и 1b — такие же как в классическом Paxos.

Фаза 2a (быстрая): Предложение. Выполняется при получении сообщений ***Promise*** от кворума избирателей. Координатор должен уведомить избирателей в случае, если какое-то значение было или могло быть зафиксировано на предыдущих раундах. Если такого значения нет, то координатор посылает сообщение ***AcceptRequest*** со специальным значением Any. Если координатор получил ***Promise*** сообщения со значением v от множества избирателей A, и не существует такого кворума Q, что $A \cap Q = \emptyset$. В таком случае никакое другое значение кроме v не могло быть зафиксировано на предыдущих раундах, и лидер отправляет сообщение ***AcceptRequest*** со значением v.

Возможна ситуация, когда после получения ***Promise*** сообщений от кворума избирателей, есть две разные политики, которые были или могли быть зафиксированы. Действительно, пусть S и R — кворумы быстрого раунда, Q — кворум классического раунда. Лидер получает сообщения от избирателей $S \cap Q$ с предложением о команде C_1, и сообщения от избирателей $S \cap Q$ с предложением

о команде C_2, конфликтующей с C_1; этой информации недостаточно для того, чтобы решить, была зафиксирована политика C_1, политика C_2 или никакая из них. Такой случай вынужденным образом запрещается благодаря введению дополнительного ограничения на кворум, называемого *требование к кворумам быстрого раунда*: если Q - кворум, S и R — кворумы быстрого раунда, то $Q \cap S \cap R \neq \emptyset$. Пример системы кворумов с такими свойствами — кворумы классического раунда имеют размер $> N/2$, кворумы быстрого раунда имеют размер $> 3N/4$.

Фаза 2b (быстрая): Принятие. Отличается от классической версии тем, что если лидер прислал ***AcceptRequest*** содержащий ***Any***, то избиратель дожидается сообщения от инициатора и самостоятельно выбирает одно из предложенных значений; в противном случае избиратель берет значение, указанное лидером. Затем выбранное значение отправляется лидеру и исполнителям.

В данной версии алгоритма каждый избиратель может выбрать свое собственное значение из предложенных. Если инициаторы предлагают несколько разных значений в течение одного раунда, возможна ситуация, когда не будет кворума избирателей, принявших одно значение. В таком случае, чтобы обеспечить живучесть протокола, как правило прибегают к использованию раунда восстановления. Среди известных техник восстановления: координированное восстановление, которое сводится к использованию раунда классического Paxos или другого централизованного алгоритма, и не-координированное восстановление, которое требует меньшее количество шагов, однако по-прежнему не исключает возможность конфликтов [31]. Псевдокод алгоритма приведен в листинге 2.

Листинг 2 – Псевдокод Fast Paxos.

```
propose(value):
    broadcast Propose(value) to acceptors

Periodically execute
leader.phase1a():
    bal <- bal + 1
    broadcast Prepare(bal) to acceptors
```

```
On Prepare(bal)
acceptor.phase1b:
    Ignore further messages from ballots below bal
    send Promised(bal, chosen) to leader

On Promised(bal, value)
leader.phase2a:
    If exist different values for ballot bal:
        initRecoveryRound()
    On quorum of messages do:
        If exist quorum of acceptors
          who could sent Promised(bal, v):
            toPropose <- v
        Else:
            toPropose <- Any

        broadcast AcceptRequest(bal, toPropose) to acceptors

On AcceptRequest(bal, value)
acceptor.phase2b:
    chosen <- value = Any ? getProposed() : value
    broadcast Accepted(chosen) to learners

On Accepted(value)
learner.learn(value):
    When received value from quorum of acceptors:
        Fix(value)
```

Допустим, алгоритм запущен в корректной среде. По указанным ранее соображениям, первая фаза может быть запущена заранее, и имеются следующие цепочки сообщений: **инициатор** $\rightarrow$ **избиратель** $\rightarrow$ **исполнитель**; **лидер** $\rightarrow$ **избиратель** $\rightarrow$ **исполнитель**. Таким образом, алгоритм требует два коммуникационных шага. Общее количество сообщений снова $\Omega(N^2)$, что можно уменьшить до порядка $5 \cdot N$ при централизованном уведомлении исполнителей.

Generalized Paxos В практических приложениях рассмотренные версии Paxos реализуют таким образом, что каждый процесс хранит линейно-упорядоченную историю обновления состояния *(log entires)*. При этом значение, относительно которого достигается консенсус — это обновленная версия состояния.

Generalized Paxos [32] — это модификация Классического и Быстрого Paxos, в которой модель *линейно упорядоченной* (total order) истории ослабляется до *частично упорядоченной* (partial order) истории. Такое изменение модели дает выигрыш в тех случаях, когда инициаторы одновременно предлагают несколько обновлений, коммутирующих между собой (то есть порядок их применения не важен); при такой ситуации в Generalized Paxos конфликта не происходит, в отличие от двух других версий Paxos, и раунд может завершится успешно. Поскольку линейно упорядоченное множество является частным случаем частично упорядоченного множества, Классический и Быстрый Paxos фактически являются частным случаем Generalized Paxos.

Роль реплицируемого состояния играют так называемые *CStructs*. На $CStructs$ должно быть задано отношение частичного порядка с операцией $\prec$. Напомним, что на частично-упорядоченном множестве определены следующие операции:

а) Функция наименьшей верхней границы (*least upper bound*):
   $x \sqcup y = z$ такой, что
   1) $x \prec z$
   2) $y \prec z$
   3) $x \prec z', y \prec z' \Rightarrow z \prec z'$

   В определении выше z не обязательно определено для всех пар x, y. В тех же случаях, когда оно определено, z должно быть элементом рассматриваемого множества.
б) Противоположная ей функция наибольшей нижней границы (*greatest lower bound*) $\sqcap$.

Для удобства будем также пользоваться префиксной формой этих двух операторов:

— $\sqcup\{C_1, C_2, \ldots, C_n\} = C_1 \sqcup C_2 \sqcup \ldots \sqcup C_n$

Также множество $CStructs$ должно быть **нижней полурешеткой** (meet-semilattice), то есть операция наибольшей нижней границы ($\sqcap$) должна быть всегда определена. Алгоритм Generalized Paxos строится таким образом, чтобы каждое вычисление наименьшей нижней границы ($\sqcup$) также было определено.

Кроме того, на $CStruct$ должна быть определена операция конкатенации, добавляющая команду к cstruct:

— $v \cdot C$

Наконец, определим некоторые отношения на $CStruct$.

Определение 5. cstruct C называется *конструируемым* (*constructible*) из заданного множества команд $Cmds$, если $C = \sqcap Cmds'$ для какого-то $Cmds' : Cmds \subseteq Cmds'$.

Определение 6. cstruct w называется *расширяющим* (*extending*) cstruct v (обозначается $w \succ v$), если $\exists u \in CStructs : u \cdot v = w$

По сравнению с Быстрым Paxos, фазы алгоритма Generalized Paxos модифицируются следующим образом.

Фаза 2a (классический раунд). Лидер, для каждого кворума избиратель Q, вычисляет $\sqcap \{acceptor.cstruct \mid acceptor \in Q\}$, и складывает их в некоторое множество Γ, изначально пустое. Затем лидер вычисляет $\sqcup\Gamma$. Если Γ оказывается пустым, то $\sqcup\Gamma = \perp$ (нейтральный элемент). Таким образом, $\sqcup\Gamma$ будет содержать такие команды, каждая из которых была принята всеми избирателями некоторого кворума. К полученному Γ лидер применяет предложенные команды.

Отдельного внимания стоит вопрос о том, когда отправлять собранное $\sqcup\Gamma$, поскольку 1b сообщения приходят с произвольной задержкой. Это происходит, как только приходят ***2a*** сообщения от кворума избирателей и таким образом $\Gamma \neq \emptyset$. Затем, если приходит новое ***1b*** сообщение и $\sqcup\Gamma$ изменяется, посылать еще одно $2a$ с новым $\sqcup\Gamma$ и тем же номером раунда. Полученное значение $\sqcup\Gamma$ лидер рассылает всем избирателям.

Фаза 2b (классически раунд) Каждый избиратель, при условии что $leader.bal > acceptor.bal$ или $leader.bal = acceptor.bal$ и $leader.cstruct \succ acceptor.cstruct$, записывает полученный cstruct и отправляет его всем исполнителям. Каждый исполнитель, по получении сообщений ***2b*** от кворума избирателей Q, записывает glb от полученных значений.

Фаза 2a (быстрый раунд). Аналогична фазе 2a классического раунда, кроме того, что лидер не применяет новых команд к $\sqcup\Gamma$, а отсылает ее избирателям как есть.

Фаза 2b (быстрый раунд). Каждый избиратель, при тех же условиях что в фазе 2b классического раунда, добавляет к полученному cstruct предложенные инициатором команды, записывает его и отправляет всем исполнителям.

В быстром раунде Generalized Paxos исполнители действуют также как в классическом раунде. Псевдокод алгоритма приведен в листинге 3.

Листинг 3 – Псевдокод Generalized Paxos.

```
propose(value):
    broadcast Propose(value) to acceptors

Periodically execute
leader.phase1a():
    bal <— bal + 1
    broadcast Prepare(bal) to acceptors

On Prepare(bal)
acceptor.phase1b:
    Ignore further messages from ballots below bal
    send Promised(bal, chosen) to leader

combine(values):
    G <— for any minimal quorum of values
        leastUpperBound(values)
    return greatestLowerBound(G)

On Promised(bal, value)
leader.phase2aStart:
    If exist conflicting values for ballot bal:
        initRecoveryRound()
    On quorum of messages do:
        combined <— for received values: combine(values)
        if isClassicRound:
            for p in pendingProposed:
                combined <— p . combined
        broadcast AcceptRequest(bal, combined) to acceptors

On AcceptRequest(bal, value)
acceptor.phase2b:
    if isFastRound:
        for p in pendingProposed:
            value <— p . value
```

```
    chosen <— value
    broadcast Accepted(chosen) to learners

On Accepted(value)
learner.learn:
    When received value from quorum of acceptors:
        For received values:
            Fix(combine(values))
```

1.3.2. Прочие алгоритмы консенсуса.

Работа Фонсесы [33] описывает способы репликации состояния: так называемые *пассивный* и *активный*, которые похожи на классический и быстрый Paxos соответственно. Однако, в этой работе не рассматривается возможность конфликта политик.

Multicoordinated Paxos использует несколько координаторов на каждом раунде. Таким образом, падение одного из координаторов не останавливает репликацию команд [31].

FGLL модифицирует Generalized Paxos с использованием аккуратно подобранных кворумов чтения и записи. В результате на восстановление после конфликта требуется лишь один дополнительный коммуникационный шаг, а также используются кворумы такие же по размеру, как в Классическом Paxos [34].

Mencius [35], реализующий реплицированную машину состояний, требует лишь двух коммуникационных шагов для репликации значения и использует различные узлы в качестве лидера, тем самым усредняя нагрузку между узлами. Однако в силу заложенного в его основу механизма работы, репликация приостанавливается в случае отказа одного из узлов до тех пор, пока узел не будет восстановлен или заменен. Регуляция распределенной конфигурации сети требует быстрого применения политик, в противном случае возможно переполнение буферов коммутаторов и последующая потеря пакетов, поэтому Mencius плохо применим к данной задаче. Fast Mencius исправляет данную проблему алгоритма Mencius [36].

EPaxos [37]. В данном алгоритме не используется выделенный лидер, и каждый узел может может непосредственно принимать решения по принятию команд. Это достигается благодаря передачи ограничений на порядок применения команд между узлами. Такой подход позволяет

равномерно распределить нагрузку между узлами. Данный алгоритм достигает следующих преимуществ над Generalized Paxos: меньший размер кворума, используемого в быстрых раундах (размер кворума в EPaxos на один меньше), разрешение конфликта требует лишь один дополнительный коммуникационный шаг. Рассмотренные алгоритмы предоставляют

преимущества над алгоритмом Generalized Paxos, которые могут быть полезны для контроллера программно-конфигурируемой сети. Улучшения Generalized Paxos, рассматриваемые в данной работе, имеют другое направление, нежели улучшения, рассмотренные выше, и потенциально применимы к этим работам. Рассмотрение данного вопроса оставлено для дальнейшей работы.

Алгоритм Caesar [38] описывает способ оптимизации Быстрого Paxos для случаев применения конфликтующих значений. В отсутствие конфликтующих предложений, Caesar требует немного большую задержку при применении политики, поскольку требует взаимодействия с большим число узлов (на один) по сравнению, например, с FGLL. Однако в задаче применения сетевых политик, вероятность возникновения конфликтующих политик невелика.

M^{2}Paxos [39] является модификацией Generalized Paxos и Multicoordinated Paxos, оптимизированной для случая, когда предложения конфликтующих команд производятся часто лишь на протяжении короткого промежутка времени. При применении команды, один из лидеров принимает на себя эксклюзивное владение принимаемой командой и всем множеством команд, конфликтующих с ней. Данная техника позволяет использовать такие же кворумы как в Классическом Paxos и избежать частой передачи графа зависимостей между объектами, сохраняя задержку в два коммуникационных шага как в Быстром Paxos. Однако, в программно-конфигурируемых сетях конфликты команд как правило редки, и ожидаемые издержки данной оптимизации превышают получаемый выигрыш.

Данная работа использует алгоритм Paxos и его модификации для решения задачи консенсуса относительно конфигурации сети в присутствии регулярных обновлений. В частности, за основу берется алгоритм Generalized Paxos.

Выводы по главе 1

В данной главе были рассмотрены существующие протоколы решения задачи консенсуса, в частности Generalized Paxos. Он производит репликацию команд всего в два коммуникационных шага в случае, если предлагаемые политики не конфликтуют, что меньше, чем у других алгоритмов. Из его недостатков:

- Бо́льшие кворумы;
- В случае конфликта политик требуется раунд восстановления, так что для репликации потребуется даже большее число коммуникационных шагов и сообщений, чем у других алгоритмов;
- На фазе «избиратель $\to$ исполнитель» требуется $\Omega(N)$ сообщений, где N — число узлов;
- Сообщения содержат не отдельные команды, а целые cstruct. Существует методика создания контрольных точек, однако и в случае ее использования сообщения содержат часть cstruct, а на время создания контрольной точки исполнение алгоритма приостанавливается.

Также, обзор существующих контроллеров с распределенным контрольным слоем показал, что контроллеры, предоставляющие по крайней мере гарантию строгой согласованности, используют централизованные алгоритмы распределенного хранилища конфигурации сети, что не оптимально для случая SDN сети, где конфликты политик редки. Также не было найдено алгоритмов, реализующих гарантию сохранения конфигурации для пакетов.

В данной работе предпринимается попытка реализовать протокол контроллера, предоставляющего гарантию сохранения конфигурации для пакетов, основанного на алгоритме Generalized Paxos, а также преодолеть некоторые из недостатков Generalized Paxos с целью достижения лучшей производительности, чем у аналогичных контроллеров.

ГЛАВА 2. РЕШЕНИЕ ЗАДАЧИ КОМПОЗИЦИИ И ПРИМЕНЕНИЯ СЕТЕВЫХ ПОЛИТИК ПРИ РАСПРЕДЕЛЕННОМ СЛОЕ УПРАВЛЕНИЯ

2.1. Введение

В данной главе приведено описание протокола распределенного контрольного слоя. В частности приводится способ использования алгоритма Generalized Paxos для задачи репликации сетевой конфигурации. Также рассматриваются некоторые улучшения в Generalized Paxos, позволяющие добиться использования меньшего числа сообщений для репликации политики в случаях корректной среды, при этом не теряя согласованности в общем случае, а также предоставляя гарантию наличия прогресса в системе при возможных отказах процессов до тех пор, пока существует хотя бы кворум корректных процессов. Новый алгоритм мы будем называть Gen-Paxos. Наконец, рассматривается процесс установки и удаления политики, позволяющий обеспечить гарантию последовательной согласованности (см. раздел 1.2), и, в том числе, гарантию сохранения конфигурации для пакетов.

Мы предполагаем следующую архитектуру программно-конфигурируемой сети: имеется несколько контроллеров и множество коммутаторов. SDN приложения выполняются на отдельных машинах, и при необходимости установить политику связываются с одним из корректных контролеров; контролеры определяют можно ли применить политику, и если это возможно, то устанавливают ее на коммутаторы.

2.2. Оптимизация Generalized Paxos

2.2.1. Постановка задачи для реплицированной машины состояний

Процессы Процессам назначаются роли:

- инициатор (proposer);
- лидер (leader);
- избиратель (acceptor);
- исполнитель (learner).

Инициаторы *предлагают* новые команды. Лидер и избиратели участвуют в консенсусе относительно приема или отказа команд и последующей их репликации, и уведомляют исполнителей о новом cstruct. В конечном счете, исполнитель должен получить cstruct, содержащий либо саму предложенную политику, либо отказ от нее.

Свойства, требуемые от протокола. Сформулируем свойства, которые требуются от разрабатываемого протокола реплицированной машины состояний.

Определение 7. При предложении некоторой политики P будем говорить, что производится *запрос на установку политики* P, и обозначать $req(P)$. Моментом инициации запроса $req(P)$ будем называть момент, когда какой-либо инициатор впервые предлагает P, и обозначать $reqInit(P)$. Моментом завершения запроса $req(P)$ будем называть момент, когда какой-либо исполнитель впервые узнает о новой принятой команде $+P$ или $-P$, и обозначать такой момент $reqDone(P)$.

Определение 8. Будем говорить, что запрос $req(P_1)$ *произошел до* запроса $req(P_2)$, если $reqDone(P_1)$ произошло до $reqInit(P_2)$. Будем называть два запроса *конкурентными*, если ни один из них не произошел до другого.

Определение 9. Cstruct C называется собираемым из заданного множества команд $Cmds$, если $C = \sqcap Cmds'$ для какого-то $Cmds' \subseteq Cmds$. Данное отношение симметрично отношению *конструируемости*, заданном в главе 1, и введено для удобства изложения.

Под $learner.fixed$ будем понимать последнюю версию cstruct, известную исполнителю $learner$. Применение политики P будем обозначать $+P$, отказ — $-P$.

а) Валидность: для каждого исполнителя $learner$, $learner.fixed$ собирается из множества $\bigsqcup\{\{+P, -P\} \mid P \in PP\}$, где PP — множество предложенных команд.

б) Устойчивость: для каждого исполнителя $learner$, если в какой-то момент $learner.fixed = v$, то во все последующие моменты времени $learner.fixed \succ v$.

в) Согласованность: $TotalFixed = \sqcup\{learner.fixed \mid learner \in learners\}$ всегда определено, то есть разные исполнители содержат непротиворечивые друг другу cstructs.

г) Живучесть: Для любых инициатора p, исполнителя l и кворума избирателей Q, если p предложил политику P, тогда если в конечном счете множество процессов $\{p,\ l,\ Q\}$ синхронно и все процессы в нем корректны, то $l.fixed$ в конечном итоге содержит $+P$ или $-P$.

д) Нетривиальность: Пусть инициатор p предложил политику P, $l.fixed = C_0$ на момент предложения для некоторого исполнителя l. Тогда для любых исполнителя l и кворума избирателей Q, если выполняется $isCStructSound(P \cdot C_0)$ и конкурентно никакой инициатор не устанавливает политику, конфликтующую с P, вплоть до получения команды с P каким-либо из исполнителей, тогда начиная с момента времени, когда процессы p, l, Q будут синхронны и корректны, в конечном счете конфигурация $l.fixed$ будет содержать команду $+P$.

Отметим, что по сравнению со свойствами, описываемыми для Generalized Paxos, здесь добавляется еще одно. Последнее свойство появляется в силу введения команд-отказов, без него алгоритм мог бы без нарушения остальных свойств отказываться от всех предлагаемых политик.

Модель сети. В сети возможны отказы процессов (node crush) и потери сообщений (omission), отправленных одним процессом другому.

Далее введем несколько определений.

Определение 10. Назовем множество процессов P синхронным, если любое сообщение, отправленное от p_1 к p_2 ($p_1, p_2 \in P$), будет получено в течение заранее заданного времени $T_{delivery}$.

Определение 11. Будем называть *сеть* синхронной, если все входящие в нее процессы синхронны.

В данной работе мы будем основываться на допущении о том, что большую часть времени сеть, или хотя бы кворум процессов, является синхронным и все процессы корректны. Следовательно, начиная с произвольного момента времени сеть в конечном счете будет синхронна.

2.2.2. Хранение конфигурации сети

В данной работе в качестве реплицированной машины состояний, хранящей конфигурацию сети, используется алгоритм на основе Generalized Paxos. Напомним, что в работе, описывающей Generalized Paxos, используются термины:

- Команда — аналог значения, выбор которого происходит в классическом Paxos. В нашем случае команде соответствует политика.
- cstruct — хранит некоторую композицию команд, в данном случае соответствует конфигурации сети

Виды политик. Рассмотрим представление политик пересылки. На данный момент будем предполагать, что каждая команда отвечает за добавление некоторой политики; далее понятие команды будет расширено.

В целом, политики пересылки — это множество ребер e_i в графе конфигурации, каждое из которых характеризуется четырьмя параметрами: $In(e_i)$ — начальная вершина ребра, $Out(e_i)$ — конечная вершина ребра, $M(e_i)$ — множество сетевых пакетов, на которые оказывает влияние ребро, $Pr(e_i)$ — приоритет ребра. Ребра, из которых состоит политика P будем называть *элементами* политики P.

Далее зададим понятие конфликта на политиках без того, чтобы ссылаться на конфигурацию сети.

Определение 12. Пусть e — элемент некоторой политики. Будем называть декартово произведение множеств $\{In(e)\} \times M(e) \times \{Pr(e)\}$ *доменом* элемента политики, и обозначать $D(e)$.

Определение 13. Доменом политики назовем объединение доменов ее элементов: $D(P) = \bigcup_{e \in P} D(e)$

Чтобы политики имели смысл, введем ограничение на входящие в нее элементы:

$$isPolicySound(P) = \nexists e_1, e_2 \in P : D(e_1) \cap D(e_2) \neq \emptyset \wedge Out(e_1) \neq Out(e_2) \quad (1)$$

Далее будем считать, что любая рассматриваемая политика удовлетворяет этому свойству. Если некоторая операция производит политику не имеющую смысла при некотором наборе аргументов, то мы считаем, что эта операция не определена для этого набора аргументов.

Сразу выделим некоторые свойства на $isPolicySound$:

а) $A \subseteq B, isPolicySound(B) \Rightarrow isPolicySound(A)$ — тривиально

б) $\forall a, b \in A : isPolicySound(\{a, b\}) \Rightarrow isPolicySound(A)$ — по свойствам квантора $\exists$.

Cstruct представляется в виде множества политик. Для cstructs вводится отдельная функция $isCStructSound$ по аналогии с $isPolicySound$:

$$isCStructSound(C) \ = \ \nexists P_1, P_2 \in C: \ P_1 \neq P_2 \ \wedge \ D(P_1) \cap D(P_2) \neq \emptyset \qquad (2)$$

Здесь мы предъявляем несколько более сильное требование чем для политик, а именно, если домены каких-то политик в C пересекаются, то даже если функция пересылки $forward(C, s, p)$ остаётся однозначной для любых коммутатора s и пакета p, конфигурация C объявляется противоречивой. Несложно показать, что для $isCStructSound$ выполняются те же свойства, что для $isPolicySound$.

Определим необходимые для cstruct операции:

а) $C1 \sqcup C2 \ = \ C1 \cup C2$

б) $v \cdot C \ = \ \{v\} \sqcup C$

в) $C1 \sqcap C2 \ = \ C1 \cap C2$

В общем случае неконфликтующие политики не обязательно коммутируют, однако для рассматриваемого алгоритма мы будем требовать, чтобы команды в cstruct коммутировали. Наконец, расширим понятие ”команды” — команда может указывать на применение политики P или на отказ от ее применения, что будет обозначаться как $+P$ и $-P$ соответственно.

Функция $isCStructSound$ расширяется на отказы от политик следующим образом: $isCStructSound(-P \cdot C) \ = \ +P \ \notin \ C \ \wedge \ isCStructSound(C)$. Так, непротиворечивая конфигурация не может содержать одновременно прием и отказ от одной и той же политики. Формулировки прочих операций остаются неизменными.

2.2.3. Модификация Generalized Paxos

Алгоритм Generalized Paxos имеет некоторые сложности, из-за которых реплицированная машина состояний на его основе может оказаться менее эффективной, чем РМС на основе прочих алгоритмов. Один из них заключается в том, что сообщения содержат не отдельные команды, а целые cstructs. Таким образом, один раунд алгоритма может требовать значительное количество трафика. В оригинальной статье о Generalized Paxos описывается техника оптимизации: периодически на избирателях создаются контрольные точки, и в сообщениях передаются только политики, не содержащиеся в последней

контрольной точке. Однако в этом случае сообщения все равно получаются более тяжеловесными, чем в прочих алгоритмах; также создание контрольной точки требует приостановки репликации команд [37].

Для преодоления этого недостатка была разработана модифицированная версия алгоритма — Gen-Paxos, отличающаяся от Generalized Paxos в следующих пунктах:

а) Мы требуем, чтобы любые две операции в cstruct коммутировали, то есть порядок применения команд был неважен.

б) Отказ от фаз ***1a*** и ***1b***, на которых лидер определяет, какие политики могли быть зафиксированы с тем, чтобы затем сообщить избирателям, какие политики можно безопасно применить.

   Взамен, вместо cstruct избиратели хранят принятые команды на одном из двух уровней.

   1) *Корневой уровень* содержит cstruct, который содержит команды, точно входящие в конфигурацию. Данный cstruct может только расти с течением времени, и все изменения в нем продиктованы лидером, когда тот уверен, что команда получена кворумом избирателей, или в том, что политика не будет применена в обход лидера.

   2) *Нестабильный уровень* содержит список политик, примененных в ходе Быстрой версии алгоритма. Политика удаляется с нестабильного уровня, когда она уже содержится в корневом уровне, или когда она противоречит конфигурации корневого уровня — в последнем случае гарантируется, что политика не была применена никаким кворумом избирателей.

   Большинство обращений чтения к локальному хранилищу избирателя возвращают сумму корневого и нестабильного уровня, так что с точки зрения других процессов, разделение на уровни отсутствует. Будем считать, что хранилище содержит cstruct, равный $u_1 \cdot u_2 \cdot \ldots \cdot u_n \cdot core$, где $core$ — команды корневого уровня, u_i — команды нестабильного уровня. Однако в случае записи в хранилище, данное разграничение помогает показать, что если какие-то политики и удаляются из хранилища, то они не могли быть ранее зафиксированы и их удаление допустимо.

Будем говорить, что избиратель *принял* команду, если эта команда содержится в корневом или нестабильном уровне локального хранилища избирателя.

Отметим, не только избиратели, но и исполнители хранят такие двухуровневые хранилища, поскольку они должны поддерживать, по-возможности, актуальную копию cstruct, хранимого каждым из избирателей.

Кроме того, поскольку лидер не используется в качестве посредника при рассылке команд избирателям, отпадает необходимость в требовании к кворумам быстрого раунда, упоминаемого в разделе 1.3, поэтому специальные кворумы, ранее использованные для быстрых раундов, в данном алгоритме использоваться не будут. Взамен, на всех стадиях алгоритма будут применяться стандартные кворумы.

в) При получении предлагаемой политики от инициатора, избиратель исполняет фазу ***2a*** немедленно. На фазах ***2a*** и ***2b*** передаются не cstructs целиком, а лишь одна политика.

В связи с данным изменением инициатор вынужден играть более активную роль. В алгоритме Generalized Paxos, в случае отказа лидера или исполнителя в процессе обработки сообщения, политика может потеряться, но она будет содержаться в cstruct, предложенном на следующем раунде, и рано или поздно будет применена. Для того, чтобы политики не терялись с учетом рассматриваемой модификации, инициатор должен периодически предлагать политику вплоть до момента, когда команда с данной политикой окажется в зафиксированном cstruct. Более того, будем предполагать, что даже в случае отказа инициатора, однажды предложенная политика будет предлагаться повторно вплоть до применения через другого инициатора, либо через данного инициатора после его восстановления; в следующих разделах будет подробно описана архитектура, позволяющая этого добиться.

г) В качестве раунда восстановления всегда используется координированное восстановление через «классическую» часть Generalized Paxos.

д) Фазы быстрой части алгоритма ассоциируются с последним раундом восстановления.

Для избежания путаницы введем понятие *эпохи*. Эпоха, условно, начинается после завершения некоторого раунда восстановления, и продолжается вплоть до следующего раунда восстановления. Номером эпохи назовем номер раунда восстановления, после которого эта эпоха началась.

Когда избиратель дает обещание не принимать сообщения с меньших номеров раундов, чем bal, то обещание распространяется на всю эпоху — сообщения быстрой части алгоритма, отправленные в эпохах с номерами меньше, чем bal, также будут игнорироваться. Данное условие позволяет убедится в том, что при расширении корневого уровня локального хранилища, происходящем во время раунда восстановления, не удалятся нестабильные команды, которые могли быть или были зафиксированы в ходе быстрой части алгоритма.

Каждый процесс будет хранить два идентификатора эпохи: $recBal$ обозначает номер последнего начавшегося раунда восстановления и является аналогом переменной bal в Paxos, он будет использоваться в раундах восстановления; $epoch$ указывает на номер последнего завершенного раунда восстановления и будет использоваться в быстрых раундах.

Фазы алгоритма. Далее подробно рассмотрены фазы алгоритма Gen-Paxos.

Будем предполагать, что каждый узел выступает одновременно в роли инициатора (для некоторых предлагаемых политик), избирателя, исполнителя, и один из контроллеров будет лидером (по меньшей мере часть времени); таким образом, любой инициатор и лидер одновременно являются избирателями и исполнителями.

Фаза 2a (быстрая) Инициатор отсылает новую команду всем избирателям в сообщении ***AcceptRequest***. К сообщению так же прилагается номер текущей эпохи. Затем он периодически повторяет исполнение этой фазы вплоть до фиксации команды с данной политикой, о чем он узнает в момент фиксации команды в качестве исполнителя. Инициатор узнает о новых эпохах, когда соответствующие раунды восстановления завершаются.

Фаза 2b (быстрая) Избиратель получает сообщение ***AcceptRequest*** с политикой P, принадлежащее эпохе $epoch$. Возможно два случая.

а) Если $epoch$ меньше, чем номер последнего раунда восстановления $recBal$, известного избирателю, то предложение считается устаревшим. Избиратель отвечает узлу-отправителю сообщением ***AlreadyPromisedFast***, содержащим номер текущей эпохи $epoch$. Отметим, что $recBal$ может не равняться $epoch$, если параллельно исполняется раунд восстановления, и в таком случае нет смысла уведомлять инициатора вплоть до завершения восстановления. Взамен, если $recBal \neq epoch$, избиратель напоминает лидеру о происходящем раунде восстановления, отсылая сообщение ***RemindRecovery*** с номером раунда $recBal$.

б) В противном случае, избиратель записывает политику, добавляя ее в нестабильный уровень, либо как $+P$, если возможно без внесения конфликта, либо как $-P$ (будем говорить, что избиратель *принимает решение о политике*). Затем он отправляет сообщение ***Accepted*** с принятой командой всем исполнителям и лидеру.

Также избиратель запоминает, что эпоха $epoch$ уже началась, и раунд восстановления с номером $epoch$ уже имел место.

Если команда с данной политикой уже содержалась в локальном хранилище избирателя, то ***Accepted*** также отправляется исполнителю, поскольку повторное предложение политики говорит о том, что она могла быть не зафиксирована до сих пор. Исполнитель, при получении сообщения ***Accepted***, запоминает выбранную команду данного избирателя, сохраняя ее в одно из собственных двухуровневых хранилищ (в общем счете, каждый исполнитель поддерживает по одному хранилищу на каждого избирателя). Полученная команда добавляется в нестабильное множество. Когда команда набирает кворум голосов, исполнитель добавляет ее в отдельный cstruct, предназначенный для зафиксированных политик.

Обнаружение конфликтов Лидер получает ***Accepted*** с целью анализа работы протокола и обнаружения конфликтов. В отличие от исходного алгоритма Generalized Paxos, лидер получает сообщения после того, как избиратели приняли решение о политике, и таким образом не влияет

непосредственно на принимаемые решения. Анализ проводится по получении сообщений ***Accepted***, содержащих команду с некоторой политикой P, от быстрого кворума избирателей. По аналогии с Generalized Paxos, требуется обнаруживать ситуацию, когда из множества предложенных конфликтующих политик разные избиратели принимают разные политики. В отличие от Generalized Paxos, если среди полученных сообщений ***Accepted*** присутствуют хотя бы одна команда $+P$ и хотя бы одна $-P$, то объявляется конфликт. Столь строгое условие используется для того, чтобы гарантировать обнаружение ситуации, когда ни $+P$, ни $-P$ не могут быть применены. В случае, если обнаружен конфликт, лидер предлагает политики, его вызвавшие, для раунда восстановления (выступая в роли инициатора). Затем он выполняет фазу 1a «классической» части алгоритма.

Фаза 1a (классическая). Остается неизменной. Лидер инициирует новый раунд, выбирая идентификатор раунда больший, чем у предыдущих раундов, и отсылает сообщение ***Prepare*** с идентификатором.

Данная фаза исполняется с периодом $T_{ballotDelay}$ до тех пор, пока лидер в качестве исполнителя не получит сообщение $Accepted$ от кворума избирателей. Если в синхронной сети гарантируется доставка сообщений в течение времени $T_{delivery}$, то должно выполняться $T_{ballotDelay} > T_{delivery}$. Кроме того, если лидер получает сообщение ***RemindRecovery*** о номере раунда bal, который он не начинал, это значит, что предыдущий лидер отказал и раунд восстановления мог быть не завершен — в таком случае инициируется фаза 1a с номером раунда $bal + 1$.

Фаза 1b (классическая). Остается неизменной. Избиратель получает сообщение ***Prepare*** с номером раунда $recBal$. Если ранее он слышал о раунде с большим номером $recBal'$, то отправляет сообщение ***Promised*** с тем номером раунда лидеру (на что лидер начинает новый раунд $recBal' + 1$). В противном случае, избиратель запоминает номер раунда $recBal$ и отправляет лидеру сообщение ***Promise*** с хранимым cstruct.

Фаза 2a (классическая). Остается неизменной. Лидер получает сообщение ***Promise***, содержащее C. Для каждого кворума избирателей

Q, от которых были получены cstructs $a.cstruct$, лидер вычисляет $\sqcap\{a.cstruct \mid a \in Q\}$, и складывает их в Γ. Если $\Gamma \neq \emptyset$, то лидер вычисляет $\sqcup\Gamma$, добавляет предложенные при обнаружении конфликта политики к $\sqcup\Gamma$ и отправляет сообщение ***AcceptRequest*** с получившимся cstruct. При этом считаем, что до получения сообщений от избирателей $\Gamma = \emptyset$, то есть впервые Γ станет непустым после получения сообщений от кворума избирателей.

Фаза 2b (классическая). Данная фаза претерпевает изменения. Избиратель получает сообщение ***AcceptRequest***, содержащее номер раунда $recBal$ и cstruct. Пусть последний слышанный избирателем раунд — $acceptor.recBal$, хранимый cstruct — $acceptor.cstruct$. Напомним, что $local.cstruct$ является двухуровневым хранилищем. Следственно, изменение возникает в следующем: при условии, что $acceptor.recBal < recBal$, или $acceptor.recBal = recBal$ и $local.cstruct.core \prec cstruct$, избиратель записывает полученный cstruct в корневой уровень хранилища acceptor.cstruct. Затем избиратель отправляет сообщение ***Accepted***, содержащий C. После завершения этой фазы, избиратель может участвовать в быстрых фазах новой эпохи, поэтому он запоминает новый идентификатор эпохи $acceptor.epoch = recBal$.

Исполнитель получает ***Accepted*** сообщение, содержащее C. В хранилище, соответствующем избирателю-отправителю, исполнитель записывает C в cstruct, лежащий на корневом уровне. Затем по аналогии с фазой 2a вычисляется $\sqcup\Gamma$, и cstruct, соответствующий зафиксированной конфигурации, пополняется всеми политиками из $\sqcup\Gamma$.

Полученный алгоритм сильно адаптирован под случай, когда все неконфликтующие политики коммутируют, а конфликты политик редки. Пока предлагаемые параллельно политики не конфликтуют между собой, алгоритм требует лишь два коммуникационных шага и порядка $3 \cdot N$ сообщений для их применения, что меньше чем у Generalized Paxos. Доказательство свойств, требуемых от данного алгоритма, будет рассмотрено далее.

Большое количество сообщений. Опишем еще одну модификацию алгоритма, связанную с используемым числом сообщений. Для выполнения одного раунда требуется $O(N^2)$ сообщений, где N — число процессов.

Как правило, алгоритмы распределенного хранилища используют TCP соединение для коммуникации узлов (среди алгоритмов, рассмотренных в главе 1, все, о которых можно судить по описанию, используют протокол TCP). В данном алгоритме также будет использоваться протокол TCP для обеспечения работоспособности при потерях сообщений. Таким образом, использование бродкаст-сообщений невозможно, и используются соединения типа точка-точка. На последней фазе каждый избиратель рассылает сообщение всем исполнителям, что приводит к использованию $O(N^2)$ сообщений.

Стоит заметить, что в некоторых случаях имеется лишь один исполнитель, непосредственно заинтересованный в устанавливаемой команде. В таком случае возможно изменить фазу ***2b*** быстрой части алгоритма таким образом, чтобы избиратели отправляли сообщение одному исполнителю. Как и ранее, избиратели также отправляют сообщение лидеру, благодаря чему лидер может разослать примененные команды тем контроллерам, которые не получили их ранее, на отдельной фазе ***2b’***.

В случае, когда в команде заинтересовано l исполнителей, требуется $\Theta(N \cdot l)$ сообщений на фазе ***2b*** и $\Theta(N - l)$ сообщений на фазе ***2b’***. Для случая $l = 1$ требуется порядка $3 \cdot N$ сообщений для установки одной политики. При этом заинтересованные исполнители получают политику после двух коммуникационных шагов, в то время как остальные — после трех шагов.

При $l \gg 1$ пользователь может быть заинтересован в линейном числе сообщений не смотря на то, что большинство исполнителей получат команду лишь после трех коммуникационных шагов. В целях достижения компромисса, данная оптимизация включается при $l > l_0$, заданного пользователем. В случае $l = 1$ рассылать политику вторичным исполнителям может непосредственно заинтересованный исполнитель — это позволит уровнять нагрузку между узлами.

2.2.4. Псевдокод модификации Generalized Paxos

Далее приводится псевдокод алгоритма с учетом всех модификаций. Операция $(+)$ на cstruct будет обозначать $(\cdot)$ или $(\sqcup)$ в зависимости от операндов. Для начала рассмотрим двухуровневое хранилище, оно представлено на листинге 4.

Листинг 4 – Некоторые операции над двухуровневым хранилищем.

```
class CStructStore:
    fun total():
        core + unstable

    On core change(newCore):
        unstable <- filter(unstable, agreesWith(newCore))
```

Поскольку поле *core* обновляется редко, результат функции *total* можно кешировать; при добавлении нестабильной политики оно пересчитывается тривиальным образом.

Непосредственно псевдокод алгоритма представлен далее. На листингах 5, 6, 7, 8, 9 представлены фазы быстрой части алгоритма, листинги 10, 11, 12, 13, 14 содержат фазы классической части алгоритма.

Листинг 5 – Фаза 2a (быстрая), выполняется инициатором.

```
ProposeFast(policy):
    Execute periodically
    Until variable 'fixed' contains policy
        broadcast AcceptRequest(this.epoch, policy) to acceptors
```

Листинг 6 – Фаза 2b (быстрая), выполняется избирателем.

```
On AcceptRequest(epoch, policy):
    if (epoch < curRecBal):
        if (curRecBal != curEpoch):  # if in recovery
            send RemindRecovery(curRecBal) to leader
    else:
        curEpoch <- epoch
        curRecBal <- epoch
        cmd <- store.addPolicyUnstable(policy)  // as ack or nack
        broadcast Accepted(cmd) to targetLearners(cmd) + leader
```

Листинг 7 – Обнаружение более новой эпохи, выполняется всеми узлами.

```
On AlreadyPromisedFast(epoch):
    curEpoch <- max(curEpoch, epoch)
```

Листинг 8 – Фаза выучивания команды (быстрая), выполняется исполнителем.

```
On Accepted(acceptorId, cmd):
    received[acceptorId].addUnstable(cmd)
    if exists Q such that
      (this.received[acc].total() contains cmd for each acc in Q):
        fixed.add(cmd)
```

Листинг 9 – Фаза обнаружения конфликтов, выполняется лидером.

```
On Accepted(acceptorId, cmd):
    broadcast Accepted(cmd) to learners - targetLearners(cmd)
    If previously heard of cmd0 such that
      (cmd0 conflicts with cmd):
        pendingProposals[curEpoch].add({cmd, cmd0})
        pendingProposals[curEpoch + 1].add({cmd, cmd0})
        InitBallot()
```

Листинг 10 – Фаза 1a (классическая), выполняется лидером.

```
InitBallot():
    startedBal <- startedBal + 1
    broadcast Prepare(startedBal) to acceptors

Until got Accepted from curRecBal ballot
Periodically do
    InitBallot()

On RemindRecovery(recBal) or AlreadyProposed(recBal):
    if startedBal < curRecBal:
        startedBal <- curRecBal
        InitBallot()
```

Листинг 11 – Фаза 1b (классическая), выполняется избирателем.

```
On Prepare(recBal):
    if curRecBal > recBal:
        send AlreadyPromised(curRecBal) to leader
    else:
        curRecBal <- recBal
        send Promise(currentProcessId, recBal, cstruct) to leader
```

Листинг 12 – Фаза 2a (классическая), выполняется лидером.

```
On Promise(acceptorId, recBal, cstruct):
    G = {}
    for all quorums Q such that received Promise
      from every acceptor in Q:
        votes <- for each acc in Q take cstruct from Promise of acc
        G.add(greatestLowerBound(votes))
    fixed[recBal] <- G.leastUpperBound()

On each change of fixed[recBal]:
```

```
    C <- fixed[recBal]
    for p in pendingProposals[recBal]:
        C.addPolicy(p)   // as accepted or rejected
    broadcast AcceptRequest(recBal, C) to acceptors
```

Листинг 13 – Фаза 2b (классическая), выполняется избирателем.

```
On AcceptRequest(recBal, cstruct):
    if (recBal > curRecBal or
        (recBal = curRecBal and cstruct extends store.core)):
        curRecBal <- recBal,
        store.core <- cstruct
        curEpoch <- recBal
        broadcast AcceptRequest(currentProcessId, recBal,
          store.total()) to learners
    else:
        send AlreadyPromised(recBal) to leader
```

Листинг 14 – Фаза выучивания команды (классическая), выполняется исполнителем.

```
On Accepted(acceptorId, recBal, cstruct):
    if cstruct extends stores[acceptorId]:
        stores[acceptorId] <- cstruct
        for Q in allMinQuorums(stores):
            fixed <- fixed + leastUpperBound(Q.totals())
```

2.2.5. Доказательство работоспособности модификации

Далее приведено доказательство перечисленных ранее свойств. Для начала, введем определение и лемму, далее будет рассмотрена основная теорема.

Определение 14. Будем называть команду *зафиксированной* если она была принята (то есть хранится в локальных двухуровневых хранилищах) хотя бы кворумом избирателей.

Зафиксированным cstruct будем называть такой cstruct, который состоит в точности из все зафиксированных команд.

Лемма 1. Зафиксированный cstruct монотонно растет с течением времени.

Доказательство. Состояние локального хранилища избирателя изменяется в двух случаях:

— На фазе 2b быстрой части алгоритма команды добавляются в нестабильный уровень хранилища, отчего зафиксированный cstruct может только увеличиться.
— На фазе 2b классической части алгоритма корневой уровень замещается новым cstruct C_{new}, при этом часть политик на нестабильном уровне может быть удалена. Покажем, что C_{new} включает в себя зафиксированный cstruct на момент выполнения данной фазы.

  Допустим, это не так, и cstruct C_new, применяемый избирателями на фазе 2b классической части алгоритма не содержит некоторую зафиксированную политику cmd. Пусть текущий раунд восстановления имеет номер $recBal$. По определению, cmd была принята некоторым кворумом избирателей Q. Рассмотрим случаи того, откуда могла придти команда cmd:

  а) Если избиратели из Q приняли команду cmd до фазы 1b раунда восстановления $recBal$, то построенный на фазе 2a лидером cstruct $\sqcup\Gamma$ будет содержать cmd. Следовательно, cmd будет содержатся в C_{new}, противоречие.
  б) Рассмотрим случаи, когда избиратель принял cmd после фазы 1b раунда восстановления $recBal$:
     – Ни один избиратель не мог принять cmd в ходе выполнения фазы 2b быстрой части алгоритма по причине данного им обещания не принимать сообщения с эпох с номерами меньше $recBal$, а эпоха с номером $recBal$ начнется для данного избирателя только после завершения им фазы 2b раунда восстановления $recBal$.
     – Предположим cmd была принята избирателем из Q как часть полученного в сообщении $Accept$ cstruct в ходе выполнения фазы 2b раунда восстановления. На фазе 1b избиратель дал обещание не принимать сообщения с раундов с номером меньше, чем $recBal$. Если же при исполнении фазы 2b раунда $recBal$ избиратель принимает некоторый cstruct, содержащий cmd, то корневой уровень будет содержать cmd, противоречие.

Теорема 2. Свойства, сформулированные в разделе 2.2.1, выполняются для Gen-Paxos.

Доказательство. Покажем выполнение каждого из свойств.

Валидность. Покажем по индукции, что лидер, избиратели и исполнители в любой момент времени хранят cstructs, собираемые из множества $\bigsqcup\{\{+P, -P\} \mid P \in PP\}$, где PP — множество предложенных команд.

а) База: все cstructs у лидера, избирателей и исполнителей инициализируется $\bot$ (не содержит никаких команд) — собирается из любого множества команд.

б) Переход: по построению, cstructs которыми оперируют лидер, избиратели и исполнители, строятся одним из следующих способов:
   - $cmd{\cdot}C$, где $cmd = +P$ или $cmd = -P$, P — политика, предложенная инициатором, C — некоторый ранее полученный cstruct и удовлетворяет условию по индукционному предположению;
   - $\sqcap\{C_i\}$, где C_i — ранее полученные некоторым способом cstructs и удовлетворяют условию по индукционному предположению;
   - $\sqcup\{C_i\}$ — аналогично предыдущему пункту.

   Свойство «собираемости» замкнуто относительно операций $(\cdot)$, $\sqcup$, $\sqcap$, поэтому все используемые в алгоритме cstructs собираются из предложенных команд.

Устойчивость. Инвариант алгоритма заключался в том, что если в какой-то момент времени зафиксированным cstruct является v, то в любой последующий момент времени для зафиксированного cstruct w верно $v \prec w$. Тем же свойством будет обладать и $learner.fixed$ по построению.

Согласованность. Требуется показать, что, каждый раз, когда в ходе работы алгоритма вычисляется некоторое выражение на cstruct, это выражение определено.

- Операция $\sqcap$ всегда определена.
- Операция вычисления cstruct, содержащегося в двухуровневом хранилище, всегда согласована, поскольку операции с хранилищем непосредственно поддерживают такой инвариант.
- Когда лидер использует операцию $(\cdot)$ на фазе 2a, он пытается вычислить $+P \cdot C$; если результат не имеет смысла, то вычисляется $-P \cdot C$, которое всегда определено.

— **Фаза 2a.** Покажем, что $\sqcup\Gamma$, вычисляемое на фазе 2a, определено. Действительно, для любых двух кворумов Q и R существует $a \in Q \cap R$. Поскольку $A \prec B$, $isCStructSound(B \sqcup C) \Rightarrow isCStructSound(A \sqcup C)$ (свойство (а) для $isCStructSound$), и a.cstruct $\sqcup$ a.cstruct, очевидно, определено, то $\sqcap\{a.cstruct \mid a \in Q\} \sqcup \sqcap\{a.cstruct \mid a \in R\}$ также определено. Поскольку Γ состоит из элементов вида $\sqcap\{a.cstruct \mid a \in Q_i\}$, где Q_i — кворумы, то $\forall e1, e2 \in \Gamma : isCStructSound(e1 \sqcup e2)$. Из чего по свойству (б) для $isCStructSound$ следует, что $\sqcup\Gamma$ определено.

— **Фаза 2b.** $\sqcup\Gamma$, вычисляемое исполнителем, определено по тем же соображениям.

Живучесть. Требуется показать, что если была предложена политика P, то, когда лидер, кворум избирателей Q и исполнителя процессы будут синхронны в конечном счете будет зафиксировано $+P$ или $-P$.

Действительно, рассмотрим период времени, когда лидер, исполнитель l и кворум избирателей Q стали синхронны.

Мы предполагаем, что политика P периодически предлагается одним из инициаторов. Фазы алгоритма построены таким образом, чтобы получаемые сообщения обрабатывались независимо от того, было ли такое же сообщение получено ранее или нет — в любом случае процесс внесет некоторые изменения в локальное состояние и отправит некоторое сообщение другим процессам, если требуется.

Предположим, для каждого избирателя a из Q, $a.epoch = a.recBal$. В быстром раунде в конечном счете инициатор отправит сообщение с номером эпохи, большим $epoch$, избирателям из Q; если инициатор отправил сообщение с меньшим номером эпохи, избиратели уведомят его об эпохе $epoch$, и инициатор повторит предложение с актуальным номером эпохи. Невозможна ситуация, когда политика P не применяется из-за того, что исполнитель не может получить кворум голосов (сообщений ***Accepted***) от избирателей. Если все избиратели из Q приняли одну и ту же команду ($+P$ или $-P$), то в конечном счете исполнитель получит сообщения от кворума избирателей Q с одной и той же командой, содержащей P, и зафиксирует ее. Если разные избиратели приняли разные команды, то в конечном счете лидер обнаружит конфликт и инициирует раунд восстановления, предложив команду P.

Отметим, что поскольку в течение раунда восстановления избиратели не принимают сообщений быстрых раундов, то новые конфликты возникнуть не могут, и новый раунд восстановления может начаться только после интервала времени, достаточного для завершения текущего раунда (в силу предположения о синхронности процессов). То есть, избиратели не дают обещаний не принимать сообщения от текущего раунда восстановления раньше, чем он закончится. Лидер в конечном счете получит сообщения от кворума избирателей Q на фазе 2а, добавит $+P$ или $-P$ к вычисленному $\sqcup\Gamma$, и разошлет всем избирателям полученный cstruct. Также и исполнитель в конечном счете получит сообщения ***Accepted*** с cstruct, содержащим $+P$ или $-P$, от избирателей Q.

Теперь покажем, что для каждого избирателя a из Q в конечном счете $a.epoch = a.recBal$. Действительно, лидер периодически инициирует новый раунд bal вплоть до получения $+P$ или $-P$ в качестве исполнителя на текущем раунде bal, что произойдет не раньше, чем некоторый кворум R узнает о завершении раунда bal (то есть, $\forall a \in R : a.epoch = a.recBal = bal$). Далее, пока $+P$ или $-P$ не будет выбрано, инициатор будет предлагать P на быстрых раундах, и существует такой избиратель $a \in Q \cap R$, который сообщит инициатору и новой эпохе, а инициатор сообщит всем избирателям из Q.

Также возможна ситуация, в которой произошел отказ лидера до момента, когда требуемые процессы стали синхронны, и лидер не мог периодически предлагать новый раунд. В этом случае в конечном счете будет избран новый лидер (свойство, которое мы будем требовать от используемого алгоритма выбора лидера; подробнее о данном алгоритме описано в следующей главе). Поскольку инициатор продолжает предлагать P, если для одного из избирателей $a \in Q$ выполнено $a.epoch \neq a.recBal$, то он отправит сообщение ***RemindRecovery*** новому лидеру, и тот инициирует новый раунд восстановления.

Нетривиальность. Покажем, что если инициатор предложил политику P, на момент предложения $l.fixed = C_0$ для некоторого исполнителя $l.fixed$, и $isCStructSound(+P \cdot C_0)$, тогда если в конечном счете было принято $-P$, то существует конкурентно производимый запрос на установку политики P' такой, что P и P' конфликтуют.

Действительно, поскольку запрос на установку P вернул $-P$, то существует кворум избирателем Q, каждый из которых принял хотя бы одну команду $+P'_a$, конфликтующую с $+P$. Рассмотрим запрос на установку одной из соответствующих политик P'_a (далее пусть $P'_a = P'$).

Допустим, $req(P')$ произошел до $req(P)$. По условию, на момент $reqDone(P')$, существует по меньшей мере один исполнитель l, для которого $l.fixed$ содержит $+P'$ (по определению запроса на установку); тогда $TotalFixed$ также содержит $+P'_a$ на момент $reqDone(P')$ — по определению $TotalFixed$. Следовательно, $TotalFixed$ содержит $+P'$ и на момент $reqInit(P)$, что приводит к противоречию с условием. Очевидно, $req(P')$ не могло произойти после $req(P)$, следовательно запросы конкурентны.

Сформулируем еще одно свойство, специфичное для нашей модификации, согласно которому раунды восстановления не должны использоваться, если конкурентно не были предложены конфликтующие политики.

— Пусть начиная с некоторого момента времени лидер le и кворум избирателей Q синхронны и корректны. Тогда если никакие два запроса на установку политик P_1, P_2 где P_1 и P_2 — конфликтуют, не конкурентны, то в конечном счете раунды восстановления перестанут использоваться.

Доказательство. Поскольку лидер le и кворум избирателей Q синхронны и корректны, то по рассуждениям, приведенным в доказательстве свойства живучести, в конечном счете завершится некоторый раунд восстановления $recBal$ такой, что раунды восстановления с номером большим $recBal$ ещё не инициировались, и каждый избиратель из Q узнает о новой эпохе: $\forall a \in Q : a.recBal = a.epoch = recBal$. Далее, поскольку никакие два конкурентно исполняемых запроса не конфликтуют, то избиратели будут принимать одинаковые решения относительно каждого запроса, и конфликта не возникнет. Новый раунд восстановления не может быть инициирован, поскольку обнаружение конфликтов не происходит, лидеру стало известно о завершении раунда $recBal$ в течение времени $T_{delivery} < T_{ballotDelay}$, и никакие сообщения ***RemindRecovery*** с номерами раунда, большими $recBal$, не приходят.

2.3. Задача согласованной композиции политик

Далее ставится задача описания контрольного слоя, решающего задачу согласованной композиции политик, введенной в разделе 1.2. Протокол OpenFlow предоставляет два режима работы контроллеров — в первом, каждому коммутатору сопоставляется некоторый управляющий контроллер, и только он может устанавливать политики; во втором, все контроллеры равноправны, и каждый контроллер может устанавливать политики на коммутаторы. Мы будем пользоваться второй схемой работы, поскольку она облегчает разрешение ситуации, когда один или несколько контроллеров отказывают. Для хранения конфигурации сети будет использоваться описанный ранее алгоритм Gen-Paxos, однако далее мы абстрагируемся от конкретного алгоритма и будем ссылаться на протокол, предоставляющий возможность хранения конфигурации, как на некоторую реплицированную машину состояний, реализующую свойства, которые мы требовали от Gen-Paxos.

2.3.1. Постановка задачи и модель сети

Требования к контрольному слою. Контрольный слой предоставляет интерфейс, позволяющий запросить установку политики P. В случае, если политика не конфликтует с уже установленной конфигурацией, то она устанавливается на коммутаторы, после чего процесс, запросивший установку политики, получает в ответ $ack(P)$; в случае конфликта, ни один пакет в сети не затрагивается новой политикой, а процесс, сделавший запрос на установку, получает $nack(P)$. Кроме того, контрольный слой должен предоставлять гарантию сохранения конфигурации для пакетов — каждый пакет обрабатывается согласно одной и той же политики.

Более формально, мы требуем свойство согласованной композиции политик, описанное в разделе 1.2 — для любой истории исполнения алгоритма H существует завершение H' такое, что

а) Согласованность: H' последовательно согласованно.

б) Завершение: в конечном счете, каждый корректный контроллер p_i, который принимает запрос $apply_i(P)$, возвращает ack_i или $nack_i$ в H. Аналогично, любой контроллер, принимающий запрос $remove_i(P)$, в конечном счете возвращает ack_i в H.

в) Нетривиальность: если корректный контроллер p_i выполняет запрос на установку $req(P)$, и для любого запроса на установку P' такого, что

$req(P') \not<_H req(P)$, верно что P и P' не конфликтуют, то запрос $req(P)$ завершится возвратом ack.

Модель сети. Рассмотрим сеть, имеющую некоторое число контроллеров, коммутаторов и хостов. Будем считать, что коммутатор может обмениваться сообщениями с произвольным контроллером. Обмен сообщениями коммутаторов с хостами выходит за рамки рассматриваемой проблемы. Поскольку в данной работе рассматривается работа контроллеров, будем также называть контроллеры *процессами*.

Как описывалось в разделе 2.2.1, в сети возможны падения контроллеров и потери сообщений, а также произвольные задержки доставки сообщений. Случаи падения коммутатора и потери сообщений между контроллером и коммутатором можно разрешать независимо от случаев ошибок взаимодействия коммутаторов в нижележащем протоколе [40] и в данной работе они не рассматриваются.

2.3.2. Используемые команды.

Для начала опишем команды, которые будут предлагаться инициаторами реплицированной машины состояний. Команду установки политики P будем обозначать $Ins(P)$. Команду удаления политики будем обозначать $Rem(P)$, эта команда не конфликтует ни с какими другими командами. При запросе к контрольному слою на установку или удаление, данные команды применяются реплицированной машиной состояний, затем только принятые команды, не конфликтующие с текущей конфигурацией, устанавливаются непосредственно на коммутаторы. Команда $Ins(P)$ может быть принята или отвергнута, в то время как $Rem(P)$ всегда принимается, поскольку она не порождает конфликты. Команды $Confirm(P)$ и $ConfirmRemove(P)$ служат для подтверждения завершения установки и также не конфликтуют ни с какой другой командой.

Кроме того, команды установки политики $Ins(P)$ изменяются таким образом, чтобы содержать *порядковое число* политики. Интуитивно, политики с большим порядковым числом могут применяться поверх удаленных конфликтующих политик лишь с меньшим порядковым числом. Команду на установку политики P и порядковым числом i будет обозначать $Ins^i(P)$. На этапе предложения политики P, инициатор берет известный ему как исполнителю cstruct C и находит такой минимальный номер j, что

$\forall + Ins^i(P_0) \in C : D(P_0) \cap D(P) \neq \emptyset \wedge +Rem(P_0) \in C \Rightarrow j > i$, и предлагает $Ins^j(P)$.

Формально, функция определения противоречивости конфигурации выглядит следующим образом:

$$\begin{aligned} &isCStructSound(+Ins^i(P_1) \cdot C) \\ &\qquad = isCStructSound(C) \wedge \\ &\qquad (\nexists + Ins^j(P_2) \in C : P_1 \neq P_2 \wedge D(P_1) \cap D(P_2) \neq \emptyset \wedge i = j) \end{aligned} \tag{3}$$

$$isCStructSound(-Ins^i(P) \cdot C) = +Ins^i(P) \notin C \wedge isCStructSound(C)$$
$$isCStructSound(Rem(P) \cdot C) = +isCStructSound(C)$$
$$isCStructSound(Confirm(P) \cdot C) = +isCStructSound(C)$$
$$isCStructSound(ConfirmRemove(P) \cdot C) = +isCStructSound(C)$$

Несложно показать, что при такой постановке непротиворечивости cstruct, свойства для cstruct, описанные в разделе 2.2.2, выполняются, и неконфликтующие команды коммутируют, таким образом использование алгоритма Gen-Paxos является законным.

Заметим, что добавление порядковых номеров не приводит к появлению необоснованных конфликтов. Более формально:

— Пусть инициатор предлагает политику $Ins^i(P)$. Если для любой политики P', конфликтующей с P, $req(Ins^j(P'))$ произойдет после $req(Ins^i(P))$, или $req(Ins^i(P'))$ и $req(Rem(P'))$ произошли до $req(Ins^i(P))$, то команда $Ins^i(P)$ будет принята.

Доказательство. Очевидно, в случае, если $req(Ins^j(P'))$ происходит после $req(Ins^j(P))$, то $Ins^j(P')$ не может вызвать конфликт при применении $Ins^i(P)$.

Рассмотрим второй случай. В силу того, как инициатор выбирает i, $Ins^i(P)$ не конфликтует с известной инициатору конфигурацией на момент предложения. По условию, для любой политики P', конфликтующей с P, запросы $req(Ins^i(P))$ и $req(Ins^j(P'))$ неконкурентны. Следовательно, по свойству нетривиальности используемой реплицированной машины состояний, команда $Ins^i(P)$ будет принята, то есть $+Ins^i(P)$ будет применена.

2.3.3. Реализация установки политик на коммутаторы.

Рассмотрим проблему установки политик на коммутаторы. Как упоминалось в главе 1, для предотвращения циклов и черных дыр на слое данных, контрольный слой должен предоставлять гарантию сохранения конфигурации для пакетов, то есть каждый пакет должен обрабатываться согласно одной и той же конфигурации даже в случае параллельной установки политик.

Будем считать, что все устанавливаемые политики уникальны. Для этого к каждой политике добавляется идентификатор вида $(controllerId, localId)$, где $controllerId$ — идентификатор инициатора, предложившего политику, $localId$ — локальный идентификатор политики для данного контроллера. На фазе, где политика рассылается всем контроллерам, каждый контроллер запоминает политику, и на следующих фазах передается лишь идентификатор.

Протокол OpenFlow предоставляет возможность атомарной установки политики на коммутатор даже в случае, если политика состоит из нескольких элементов (правил). Поэтому далее без ограничения общности будем считать, что каждая политика состоит из одного элемента.

Для установки воспользуемся подходом, предложенным в работе Канини, Кузнецова и др.[23], который заключается в присвоении *тегов* (tags) конфигурациям сети. Будем обозначать тег, соответствующий конфигурации C, как $\tau(C)$. При появлении пакета в сети, коммутатор назначает пакету тег, соответствующий конфигурации, по которой должен обрабатываться данный пакет. Далее каждый коммутатор обрабатывает пакет согласно конфигурации, соответствующей этому тегу.

Метод простой установки, когда контроллеры, получив новую политику, немедленно устанавливают ее на коммутатор, потенциально порождает черные дыры: один из промежуточных коммутаторов на пути пакета может не успеть принять конфигурацию, соответствующую тегу пакета, и пакет будет отброшен. Для решения этой проблемы используется техника двухфазного обновления.

Пусть коммутатор хранит биекции $installedTag$ и $removedTag$ из множества политик в множество пакетов; данные функции модифицируется при каждой установке новой политики на коммутатор. Их семантика следующая: для приходящего на коммутатор пакета p с тегом τ конфигурация $config(\tau)$, в соответствии с которой он обрабатывается, состоит из всех политик P

таких, что $installedTag(P) \leqslant \tau$, и, если $removedTag(P)$ определено, то $\tau < removedTag(P)$.

Кроме того, каждый коммутатор хранит тег $injectTag$, которым он помечает все входящие в сеть пакеты.

Для установки политики P некоторый выделенный контролер выполняет следующие шаги:

а) На первой фазе, контроллер устанавливает новую политику P. Каждый коммутатор имеет счетчик установленных политик $installedNum$. Если P ранее не была установлена, то коммутатор увеличивает значение счетчика, и расширяет функцию $installedTag$ таким образом, что $installedTag(P) = installedNum$. Затем $installedNum$ возвращается контроллеру — это и будет тег новой конфигурации τ^P.

   На данном этапе ни один пакет не затрагивается данной конфигурацией, поскольку пакеты с тегом τ^P в сети отсутствуют.

б) На второй фазе, контроллер отправляет значение τ^P на каждый коммутатор, обновляя значение $injectPacket := \tau^P$. Только после этого шага в сети могут появляться пакеты с тегом τ^P.

Удаление политики выполняется аналогично, с той лишь разницей, что вместо $installedTag$ расширяется функция $removedTag$. По завершении двухфазного обновления, каждый контроллер устанавливает команду $Confirm(P)$ или $ConfirmRemove(P)$ (для случаев установки и удаления соответственно); после завершения применения этой команды установка считается завершенной.

Каждый коммутатор помнит идентификатор контроллера, устанавливающего обновления. В случае отказа данного контроллера, остальные контроллеры избирают нового, и коммутаторы запоминают, что следует принимать обновления конфигурации только от него. Новоизбранный контроллер запрашивает тег последней установленной политики и значение $installedTag$, и если требуется, завершает установку политики (дожидаясь момента, когда она окажется в его локальной конфигурации, если необходимо). Затем контроллер устанавливает все такие политики P, что $+P \in C \wedge Confirm(P) \notin C$, после чего переходит к работе в штатном режиме, устанавливая все появляющиеся в конфигурации политики.

Используемый протокол реплицированной машины состояний позволяет производить репликацию нескольких политик конкурентно, однако описанный в данном разделе протокол установки политики на коммутаторы требует последовательного исполнения двухфазного обновления — установка следующей политики не может начаться до завершения установки предыдущей. Чтобы скомпенсировать этот недостаток, можно группировать политики, назначая целой группе политик один тег и устанавливая входящие в группу политики вместе. В случаях, когда они затрагивают разные коммутаторы, данная оптимизация может дать ощутимый выигрыш в производительности.

Существует проблема, связанная с нахождением и удалением устаревших политик. Протокол OpenFlow не описывает способа обнаружить окончание потока пакетов, ни метода проверки наличия пакета с заданным заголовком в сети. Одно из решений — если имеется возможность оценить максимальное время нахождения пакета в сети, пренебрегая пакетами, которые остаются в сети слишком долго, то вторую фазу можно выполнять не позже, чем через заданный промежуток времени после окончания первой фазы. Поиск лучшего решения данной проблемы оставлен для дальнейшей работы.

2.3.4. Псевдокод контрольного слоя

Далее приведен псевдокод контролера (листинг 15) и коммутатора (листинг 16).

Листинг 15 – Псевдокод алгоритма контроллера.

```
class Controller:
    installPolicy(P):
        config <- getConfig()
        ord = least i:
            for each policy G from config
              if P conflicts with G then i > ordinal(P)
        // request to replicated state machine
        decision <- applyRSM(P)
        if decision = Nack(P):
            return Nack(P)
        // else case is processed below
        Wait for Confirmed(P) from RSM
        return Ack(P)

    On Ack(P) from RSM:
```

```
        if this.isLeader:
            installPolicyIntoDataplane(P)

    installPolicyIntoDataplane(P):
        tag <- {}
        for s : switches:
            // communicate with switch in RPC manner
            // 'tag' should be the same for any switch
            tag <- call s.installPolicy(P)
        for s : switches:
            send MarkNew(tag) to s
        applyRSM(Cofirmed(P))
```

Интерфейс, предоставляемый контрольным слоем, состоит из функции *installPolicy*, отвечающей за весь процесс установки. Функция *installPolicyIntoDataplane* является вспомогательной функцией, выполняющей алгоритм двухфазного обновления, когда применение политики подтверждено реплицированной машиной состояний, и исполняется на каждом контроллере. Функция удаления политики не представлена, однако реализуется аналогично.

Листинг 16 – Псевдокод алгоритма коммутатора.

```
class Switch:
    installPolicy(P):
        if installedTagsInv[P] != None
            return installedTagsInv[P]
        tag <- installedTags.size() + 1
        installedTagsInv[P] = tag
        for e : elements(P):
            addRuleToMatchTable(mask(e), priority(e),
              nextSwitch(e))
        return tag

    On MarkNew(tag):
        injectTag = max(injectTag, tag)

    On IncomingPacket(p):
        if p just appeared in the network:
            p.tag = injectTag
        P <- Find P from Match Table such that
          mask(P) matches p and
          installedTagsInv[P] <= p.tag and
```

```
    priority(P) is the largest
  if P != None:
      forwardPacket(P.nextSwitch, p)
```

У коммутатора представлены три функции: *installPolicy* производит установку политики, запрошенную контролером на первой фазе двухфазного обновления, *markNewPackets* изменяет тег, которым помечаются пакеты, и *forwardPacket* обрабатывает пришедший пакет.

2.3.5. Доказательство выполнения свойств согласованной композиции политик

Покажем, что полученный алгоритм установки политик удовлетворяет свойствам, требуемых задачей согласованной композиции политик (СКП), представленной в разделе 1. Будем основываться на предположении, что используемая реплицированная машина состояний предоставляет свойства, ранее описанные для модификации Gen-Paxos, а также предоставляет гарантию последовательной согласованности (для РМС данное свойство формулируется аналогично тому, как оно было сформулировано в разделе 1, разница в отсутствии событий появления пакета в сети). Доказательство свойства последовательной согласованности Gen-Paxos не приводится в данной работе. Также напомним, что каждый контроллер считается корректно построенным, то есть не производит никакие два запроса (к РМС или к контрольному слою) конкурентно.

Докажем выполнение требуемых свойств. Для начала, введем именования событиям, происходящим в системе.

— $applyReq_P$ и $applyResp_P$ — события запроса $apply_P$ на установку или удаление политики P в реплицированную машину состояний (Gen-Paxos) и получения ответа; здесь мы не будем проводить различия между установкой и удалением;
— $installReq^1_P$, $installResp^1_P$ — событие запроса установки или удаления политики на коммутатор в первой фазе двухфазного обновления и событие получения ответа;
— $installReq^2_P$, $installResp^2_P$ — аналогично для второй фазы;
— $confirmReq_P, confirmResp$ — события отправки уведомления о завершении установки и его получения;
— ev_i — событие появления пакета в сети.

Запрос к контрольному слою на установку политики, начинающийся с $applyReq_P$ и завершающийся $confirmResp_P$, будем обозначать $install_P$. Пусть H некоторая история выполнения алгоритма. Требуется показать, что существует H' — завершение H, удовлетворяющее свойствам согласованности и завершения.

Для начала, существует H^* — завершение H. Действительно, рассмотрим исполнение алгоритма, соответствующее H. Предположим, что кроме имеющихся запросов на установку политики $applyReq_P$ в H, исполнение не содержит никаких других (дальнейших) запросов. Поскольку в конечном счете сеть синхронна, запросы $apply_P$ в конечном счете завершатся по свойству живучести, равно как завершатся запросы $installReq_P^k$. Таким образом, все запросы $install_P$ конечном счете завершатся ответом. История H^*, соответствующая исполнению, где все запросы были завершены, и есть искомое завершение истории H.

Рассмотрим требуемые от H^* свойства.

- Завершение — в конечном счете, каждый корректный контроллер p_i, принявший запрос $applyReq_P$, возвращает ack_i или $nack_i$ в H^*.

  Выполняется по построению H^*.
- Нетривиальность — выполняется в силу свойства нетривиальности для используемой реплицированной машины состояний.
- Последовательная согласованность — существует последовательная легальная история S, эквивалентная H^* и для которой $<_{H^*} \subseteq <_S$.

  Построим требуемую упорядоченную историю S путем расширения отношения предшествования $<_{H^*}$, и покажем, что отношение $<_S$ непротиворечиво (то есть, остается антисимметричным: $\nexists a, b: \ a <_S b \ \wedge \ b <_S a$) и новая история остается законной.

  Согласно описанной методике, политики устанавливаются в некотором порядке. Расширим в H^* отношение предшествования, и назовем новую историю H^{**}, так, чтобы, если политики P_1 и P_2 установились с тегами τ_1 и τ_2 соответственно и $\tau_1 < \tau_2$, то $installResp_{P_1}^2 <_{H^{**}} installReq_{P_2}^2$. Это не вносит конфликт с локальной историей контроллеров, поскольку если в H^* верно $install_{P_1} <_{H^*} install_{P_2}$, то $installResp_{P_1}^2 <_{H^*} applyResp_{P_1} <_{H^*} applyReq_{P_2} <_{H^*} installReq_{P_2}^1$,

то есть установка P_2 на коммутаторы произойдет после установки P_1 на коммутаторы.

Используемая нами реплицированная машина состояний удовлетворяет свойству линеаризуемости. Расширим в H^{**} отношение предшествования так, чтобы никакие два запроса к реплицированной машине состояния не конкурировали, и назовем новую историю H^{+}.

Заметим, что, поскольку запросы $install_P$ упорядочены в H^{+}, мы можем их пронумеровать в порядке, в котором они встречаются в H^{+}. Соответственно, политике P также можно сопоставить некоторый номер i, поэтому далее будем ссылаться на события по номеру, а не по политике, например, $confirmReq_i$.

Перечислим имеющиеся на данный момент отношения предшествования между событиями:

а) $applyResp_i <_{H^+} applyReq_{i+1}$ (запросы к реплицированной машине состояний упорядочены);

б) $applyReq_i <_{H^+} applyResp_i <_{H^+} installReq_i <_{H^+} \ldots <_{H^+} confirm_i$ (причинно-следственная связь);

в) $confirmResp_i <_{H^+} applyReq_j$, где i, j — запросы одного контролера, $j > i$.

Данные связи должны быть учтены при построении эквивалентной истории.

Расширим отношение предшествования истории H^{+} — и назовем новую историю S — так, чтобы выполнялось следующее:

– $confirmResp_i <_S applyReq_{i+1}$. Данное требование сильнее, чем связь (а). Оно не противоречит (б), поскольку оно регламентирует отношение предшествования для событий, относящихся к установке одной политики. Наконец, это требование усиливает связь (в), следовательно, оно не вносит противоречия в $<_S$.

– Для события появления пакета ev, если оно обрабатывалось в соответствие с политикой P, соответствующей номеру i, то добавим требование $confirmResp_i <_S ev <_S applyReq_{i+1}$. Оно не противоречит с существующими связями между событиями, поскольку определенные на данный момент связи не распространяются на события появлений пакетов в сети.

Полученное S последовательно в силу последних двух введенных требований (первое влечет неконкурентность любых $install_i$ и $install_j$, второе — неконкурентность любых ev_k и $install_i$).

Покажем, что $<_S$ законно.

– Политика применяется успешно только в том случае, если она не противоречит ранее установленным политикам в H.

  Следует из свойств согласованности и устойчивости используемой реплицированной машины состояний.

– Для события ev появления пакета p в сети, след пакета $\rho_{S,ev}$ соответствует композиции установленных до ev политик.

  Для начала, пакет имеет след, целиком построенный в соответствие с конфигурацией, заданным некоторым тегом. Поэтому достаточно показать, что назначенный пакету тег соответствует композиции установленных до ev политик.

  Действительно, существует цепочка предшествования событий $installResp_i^2 <_S confirmResp_i <_S ev$, где i — тег, согласно котором построен след пакета p. Следовательно, пакет p обрабатывался согласно конфигурации, соответствующей i, или некоторой конфигурацией, установленной позже.

  Однако, для следующей устанавливаемой в сеть политики P' верно $ev <_S applyReq_{P'} <_S installReq_{P'}^2$, то есть она установлена позже события ev, следовательно, пакет p обрабатывается в точности в соответствии с требуемой конфигурацией.

Таким образом, описанный протокол контрольного слоя решает задачу СКП.

Выводы по главе 2

В данной главе были рассмотрены модификация Generalized Paxos — Gen-Paxos, адаптированная под случай редких конфликтов политик и коммутации непротиворечивых команд, сформулированы требуемые от него свойства и доказано их выполнение. Также был описан алгоритм контрольного слоя с использованием Gen-Paxos, позволяющий устанавливать сетевые политики с предоставлением гарантий линеаризуемости и сохранения конфигурации для пакетов, что сильнее, нежели у рассмотренных аналогов (Ravana, BFT-Light, и другие), показано выполнение данных гарантий.

ГЛАВА 3. РЕАЛИЗАЦИЯ АЛГОРИТМА ПРИМЕНЕНИЯ СЕТЕВЫХ ПОЛИТИК ПРИ РАСПРЕДЕЛЕННОМ СЛОЕ УПРАВЛЕНИЯ, ТЕСТИРОВАНИЕ И СРАВНЕНИЕ С АНАЛОГАМИ

3.1. Введение

В целях апробации протокола был реализован прототип предложенного контроллера. Для этого использовался язык Haskell, поскольку он предоставляет удобные инструменты для реализации и тестирования распределенных протоколов, позволяет писать высокопроизводительные программы, и написание кода на этом языке потенциально приводит к меньшему число ошибок по сравнению с другими языками.

При реализации прототипа использовались следующие библиотеки, в орфографическом порядке: async, base, ansi-terminal, autoexporter, containers, data-default, exceptions, formatting, fmt, lens, message-pack, MonadRandom, monad-stm, msgpack, mtl, reflection, stm, stm-chans, tagged, template-haskell, text, text-format, time-units, time-warp, random, QuickCheck, universum. Кроме того, использовался проект, реализующий протокол OpenFlow на языке Haskell [41].

Прототип загружен на GitHub [42]. Приложение можно запускать в двух режимах:

а) Режим демо, в котором исполняется непосредственно описываемый протокол распределенного хранилища. Через файл конфигурации пользователь может задавать параметры протокола, такие как: расписание раундов, расписание предложений политик, неполадки сети, количество процессов-избирателем и процессов-исполнителей. Благодаря журналированию фаз пользователь может наблюдать за ходом работы протокола через терминал.

   Роли процессов разделены, будто выполняются с разных машин (что выражается в том, что разные роли процессов слушают сообщения на разных портах)

б) Режим контроллера, в котором приложение выполняет собственно функции SDN контроллера.

   Каждый контроллер выполняет одновременно все роли: избирателя, исполнителя, предлагающего в случае уведомления о новом потоке пакетов, лидера в случае победы в голосовании. Общее количество контроллеров предполагается фиксированным.

При запуске контроллера, в параметрах командной строки пользователь указывает общее количество контроллеров, идентификатор данного контроллера, адреса и порты, с которых контроллер слушает сообщения от других контроллеров и от коммутаторов.

Для реализованного прототипа проведен анализ в двух основных направлениях.

а) Надежность алгоритма Gen-Paxos оценивается через тестирование прототипа. Хотя тестирование не гарантирует корректности алгоритма, в виду наличия теоретического доказательства мы считаем такой подход достаточным.

б) Оценки производительности алгоритма осуществляется через симуляцию программно-конфигурируемой сети и последующую оценку работы контроллера под нагрузкой. При похожих условиях запускаются контроллеры, аналогичные рассматриваемому в данной работе, и производится сравнение.

Далее в этой главе описываются компоненты приложения, детали реализации и использованные техники оптимизации. Затем приводится методология оценки реализованного прототипа и полученные результаты.

3.2. Реализация алгоритма применения сетевых политик при распределенном слое управления

Для начала рассмотрим технические решения, принятые в ходе разработки прототипа алгоритма.

3.2.1. Компоненты приложения

Реализация контроллера. Реализованный контроллер состоит из двух основных частей

а) Модуль OpenFlow отвечает за обмен сообщениями между контроллерами и коммутаторами. Он отвечает за
   — установление соединения с коммутаторами и выполнение протокола рукопожатия;
   — обнаружение событий происходящих в сети, в частности появление новых потоков пакетов;
   — установку сетевых политик на требуемые коммутаторы;

Описанная функциональность реализован на основе проекта AndreasVoellmy/openflow[41], реализующего контроллер на основе протокола OpenFlow.

б) Модуль распределенного хранилища отвечает за репликацию состояния, содержащего конфигурацию сети. Таким образом, каждый контроллер одновременно является одним из узлов, участвующих в репликации сетевой конфигурации. Данный модуль предоставляет интерфейс с методом $installCommand(command, onInstall)$, который инициирует репликацию политики $command$ и выполняет функцию $onInstall$ после успешной установки или отказа от применения политики.

Модуль реализован на основе алгоритма, рассмотренного в главе 2, подробное описание представлено далее.

Локальное хранилище. Для локального хранилища используется библиотека acid-state. Она предоставляет интерфейс хранения состояния с ACID гарантиями (атомарность, консистентность, изолированность, устойчивость), удобный для использования в функциональном языке.

Коммуникация по сети. Для общения контроллеров по сети используется библиотека time-warp. Она предоставляет интерфейс сетевой коммуникации, основанный на событиях. Сообщения, которыми обмениваются узлы, являются отдельными типами данных. Инициирующая сторона составляет и посылает одно такое сообщение, слушающая сторона объявляет несколько *слушателей сообщений* — каждый слушатель содержит способ обработки сообщений одного типа.

В быстрой версии алгоритма для коммуникации используется сетевой протокол UDP, поскольку

а) потеря сообщений не критична;
б) размер сообщений ограничен;
в) сокращение использования трафика и увеличение скорости передачи данных ставятся в приоритет.

В раундах восстановления используется протокол TCP, поскольку сообщения содержат конфигурацию целиком, таким образом потенциально не ограничены по размеру.

Модуль выбора лидера будет рассмотрен в одном из следующих разделов.

Оптимизации контроллера.

Пакетирование. В качестве оптимизации используется известная техника *пакетирования* (batching). Каждое сообщение, призванное передавать политику, взамен передает несколько политик за раз. На стадии ***Propose*** предложения политик складываются в буфер, и затем отправляются группой. Пользователь может указывать в конфигурации параметры пакетирования, а именно:

а) Максимальное количество политик в предложении;
б) Максимальное время заполнения буфера, после которого предложение с группой политик отправляется досрочно.

Локальная обработка сообщений себе. Определенную долю сообщений составляют сообщения, присылаемые процессом самому себе, особенно в случае небольшого числа контроллеров в сети. Для оптимизации использования трафика реализация сетевого интерфейса была модифицирована таким образом, чтобы отправка сообщения процессом самому себе сводилась к вызову соответствующего сообщению слушателя в отдельном потоке.

Нумерация и кэширование политик. Регламентируемый протоколом OpenFlow размер политики достаточно большой — только лишь маска занимает 40 байт, а размер политики с одним действием составляет приблизительно 90 байт. Как было оговорено в разделе 2.3.3, политики имеют идентификатор. На фазе, где политика рассылается всем контроллерам, каждый контроллер запоминает политику, и на следующих фазах передается лишь идентификатор.

Реализация коммутатора. Существующий на сегодняшний день OpenFlow протокол не требует от коммутаторов наличия некоторых функций, которые были описаны в главе про алгоритм контрольного слоя. Перечислим данные функции:

а) пометка пакета тегом;
б) нахождение правила обработки пакета с учетом тега;

в) сопоставление политикам тегов.

Для реализации этих функций была модифицирована существующая реализация коммутатора Open vSwitch[43].

3.2.2. Протокол выбора лидера

Для отсутствия единой точки отказа, любой протокол протокол распределенной системы, подразумевающий выполнение некоторых функций только одним из узлов (то есть централизованный или частично централизованный протокол) должен предусматривать возможность падения данного узла. Будем называть такой узел *координатором* (coordinator) (в отличие от термина *лидер*, который в данной работе мы будем использовать только применительно к алгоритмам на основе Paxos). Обычной практикой в таких протоколах является передача функций координатора другому, корректному, процессу [44].

Протокол, рассмотренный в данной работе, является частично централизованным — в режиме работы классического Paxos подразумевается наличие процесса-лидера. В случае отказа процесса-лидера классические раунды не могут успешно завершаться до тех пор, пока лидер не будет восстановлен.

Одно из преимуществ протокола Paxos заключается в том, что независимо от выбранного протокола выбора лидера, корректность Paxos гарантируется (в отличие, например, от алгоритма Raft, в котором используется специальный протокол выбора лидера; он же играет значимую роль в достижении корректности всего протокола [45]). Даже существование нескольких лидеров на протяжении произвольного промежутка времени не нарушает корректности в Paxos, однако может помешать достижению живучести в силу возникающих конфликтов между cstructs. Для достижения живучести, в алгоритме Paxos достаточно использовать алгоритм выбора лидера, который *в конечном счете* определяет одного лидера для всех процессов.

В данной работе автор выбирал между тремя известными алгоритмами выбора лидера [46].

— «Чистый» алгоритм.

Под таким алгоритмом в данном контексте будем понимать алгоритм, который выбирает лидера, основываясь на номере текущего раунда, и не зависящий от событий, возникающих во время исполнения протокола, в

частности от множества корректных лидеров на данный момент. Paxos допускает использование чистого алгоритма для выбора лидера.

Среди возможных вариантов чистого алгоритма желательно выбрать *справедливый* (fair) алгоритм, то есть такой алгоритм, который будет выбирать каждый из узлов лидером с одной и той же вероятностью при разных номерах раунда. Хорошим кандидатом на роль такого алгоритма является *циклический алгоритм* (round-robin algorithm) [47].

Преимущество такого подхода заключается в простоте реализации и отсутствии накладных расходов. Среди недостатков - неадаптивность алгоритма; если один из узлов остается сломанным в течение долго времени, то он будет продолжать выбираться лидером и периодически раунды не будут завершаться успешно, вызывая регулярную задержку в обслуживании.

— Теперь рассмотрим алгоритм, которые предполагает выбор, по-возможности, корректного процесса в качестве лидера.

Рассмотренный далее протокол подразумевают, что множество процессов, участвующих в нем, не изменяется, и каждому процессу присвоен уникальный идентификатор id_p; на множестве идентификаторов задана операция сравнения. Протокол стремится выбрать в качестве координатора корректный процесс с наибольшим идентификатором.

—

а) Когда процесс p замечает, что координатор не отвечает, он инициирует выборы, посылая сообщение $Election$ все процессам с большим идентификатором. Вопрос о том, как процесс определяет, что другой процесс не отвечает, будет рассмотрен далее.

б) Каждый процесс, получивший сообщение $Election$, отправляет в ответ сообщение Ok со своим идентификатором.

   1) Если ни один процесс не отвечает, p побеждает в выборах и становится новым координатором.

   2) Получив сообщения Ok от других процессов, p выбирает процесс с бо́льшим идентификатором и считает его новым координатором. Затем p отправляет сообщение $Grant$ выбранному процессу.

в) Новый координатор, отправляет сообщение $Coordinator$ всем процессам.

г) Любой процесс, в случае получения сообщения $Coordinator$, останавливает исполнение протокола.

В худшем случае данный алгоритм требует четыре коммуникационных шага, и может использовать $O(n^2)$ сообщений, где $n = |P|$.

Нетрудно показать, что если во время исполнения протокола множества корректных и некорректных процессов не меняются, все корректные процессы синхронны и никакие сообщения не теряются, то все корректные узлы выберут одного и того же лидера.

Использование кворумов. Одним из крайних случаев при выполнении любого распределенного протокола является разделение сети на изолированные подсети. При таком сценарии, в ходе выполнения любого из двух предложенных алгоритмов выбора лидера процессы каждой из подсетей могут выбрать своего координатора. При этом все процессы будут считать, что выбор координатора завершился успешно.

Для предотвращения рассмотренной ситуации будем считать, что координатор выбран только если он собрал голоса от кворума процессов (включая себя).

Обнаружение некорректных процессов. Как процессы должны понимать, являются ли процессы некорректными? Для этого придется использовать ограничение на время доставки сообщения. Если процесс не подтверждает получение сообщения в течение заранее установленного времени $T_{timeout}$, то такой процесс считается некорректным.

Чтобы процессы могли отслеживать факт падения координатора, координатор должен отправлять heartbeat-сообщение всем процессам с периодом $T_{heartbeat} < T_{timeout}$. Если процесс не получал heartbeat-сообщения в течение времени $T_{timeout}$, он инициирует выбор нового лидера.

Заметим, что в обоих рассмотренных протоколах рассылку $Coordinator$ можно отменить, заменив первым heartbeat-сообщением.

Динамическое задание ограничения по времени. В распределенной системе может быть затруднительно заранее задать такое $T_{timeout}$, которое подойдет для заданной сети. Для определения оптимального $T_{timeout}$

используется техника *экспоненциальной выдержки* (exponential backoff) [48]. Если алгоритм выбора координатора не завершается успешно, то есть не собирает кворума голосов, то $T_{timeout}$ увеличивается в некоторое заданное количество раз $T_{timeout} := T_{timeout} \times k_{exp}$, и алгоритм повторяется снова. Если алгоритм завершается успешно, то процессы оценивают время, потребовавшееся для доставки сообщений другим процессам, и m-процентиль по полученному распределению принимают за новый $T_{timeout}$.

Достоинства и недостатки. Главное достоинство двух рассмотренных алгоритмов выбора координатора в том, что если статусы процессов (корректный или некорректных) остаются неизменны за время голосования, то данные алгоритмы не могут выбрать некорректный процесс в качестве координатора. С другой стороны, данные алгоритмы отличаются содержат относительно большое количество технических деталей для задачи, где требуется обеспечение столь слабых гарантий. Также рассмотренные алгоритмы подразумевают интенсивный обмен сообщениями и требуют некоторое время, пропорциональное сетевым задержкам, для завершения.

3.3. Тестирование надежности протокола

Тестирование Gen-Paxos. Всеобъемлющее тестирование протоколов распределенных систем как правило является трудоемкой задачей, поскольку число возможных путей исполнения сильно растет по мере усложнения протокола.

Для алгоритма Gen-Paxos были реализованы следующие виды тестов:

а) Юнит-тесты покрывают отдельные функции, работающие с командами, cstruct, кворумами.
б) Системные тесты запускают алгоритм целиком при различных условиях.

Написание юнит-тестов не представляет особого интереса, поэтому далее подробно описывается методика реализации системных тестов.

В ходе тестирования, при запусках алгоритма желательно эмулировать всевозможные условия, возникающие при запуске в реальных системах. Сюда входят неполадки сети, в частности:

— Задержка доставки пакетов;

— Переупорядочивание сообщений;
— Потеря и дублирование сообщений (в случае использования протокола UDP);
— Отказ узла.

Кроме того, протокол имеет множество параметров, указываемых пользователем, таких как период исполнения раундов Классического Paxos, период повторных предложений политик, количество участвующих в протоколе узлов.

Для эмуляции сетевых задержек и отказов узлов использовалась библиотека time-warp. Она предоставляет удобный интерфейс для задания сложных правил, регулирующих состояние сети, в том числе применимых для отдельных узлов или в ограниченном промежутке времени. Также, она содержит опцию эмуляции многопоточной среды, при которой нивелируются используемые протоколом временны́е задержки с сохранением порядка исполнения потоков, позволяя тем самым исполнить большее количество тестов за тот же промежуток времени.

В конечном счете была реализована проверка множества различных свойств, например:

— На протяжении работы протокола работали два избирателя из трех $\Rightarrow$ исполнитель примет преложенные значения;
— Задержка на доставку пакетов временно превышает длительность раунда, затем протокол работает в отсутствие задержек доставки на протяжении некоторого времени, достаточного для выполнения 1-2 раундов $\Rightarrow$ исполнитель примет предложенные значения;
— Были предложены конфликтующие друг с другом команды $\{C_1, C_2, \dots, C_n\} \quad \forall i, j : i \neq j \Rightarrow \overline{isCommandSound(C_i \sqcup C_j)}$ $\Rightarrow$ все исполнители примут одну команду C_i, причем одну и ту же.

Помимо упомянутых примеров элементарных свойств, проверялись свойства с предусловиями, похожими на условия реальной системы: долгая работа протокола и множество предложенных команд, случайные потери сообщений и варьирующиеся задержки доставки.

Для автоматической генерации тестовых параметров использовалась библиотека QuickCheck. В число варьируемых с ее помощью параметров входят длительность раундов, частота предложения политик, длительность и

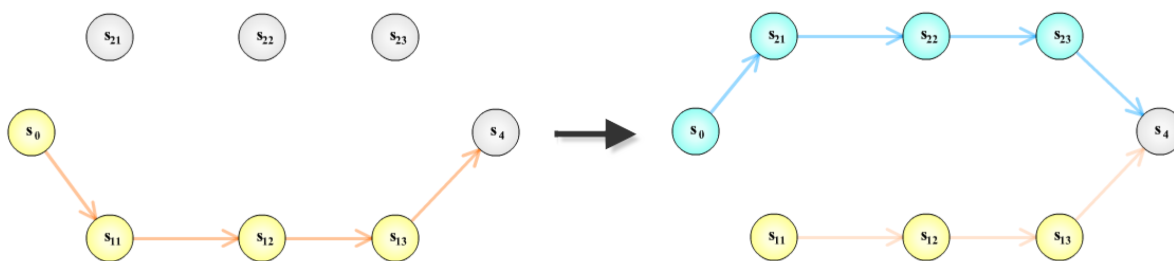


Рисунок 4 – Установка новой сетевой политики, при которой возможна потеря пакетов в случае отсутствия гарантий сохранения конфигурации для пакетов.

величина сетевых задержек. Полный список проверяемых свойств можно найти в репозитории с прототипом [42].

Тестирование сохранения конфигурации для пакетов и сравнение. Для проверки гарантии сохранения конфигурации для пакетов, контроллер запускался в симулированной программно-конфигурируемой сети, и воспроизводился сценарий установки такой сетевой политики, при котором возможны потери пакетов в случае использования алгоритма, предоставляющего более слабые гарантии.

Для симуляции программно-конфигурируемой сети использовался инструмент mininet [49]. Используемая в целях тестирования сеть состояла из двух контролеров и восьми коммутаторов; архитектура слоя данных представлена на рисунке 4. Изначально существует путь, через коммутаторы s_{11}, s_{12} и s_{13}, и пакеты, которым требуется переместиться от коммутатора s_0 к коммутатору s_4, следуют ему. Затем через один из контроллеров инициируется установка политики, согласно которой пакеты должны следовать через коммутаторы s_{21}, s_{22} и s_{23}. Если коммутатор s_0 начнет передавать пакеты согласно новой конфигурации до того, как коммутаторы s_{21}, s_{22} и s_{23} узнают о ней, то дошедший до одного из этих коммутаторов пакет не будет иметь правила обработки и будет отброшен.

Тестирование заключается в оценке среднего времени, в течение которого пакеты, отправляемые от s_0 к s_4 отбрасываются, во время описанного обновления конфигурации. Для этого при помощи инструмента iperf отправляется непрерывный поток пакетов, и при помощи утилиты tcpdump измеряется промежуток времени, когда пакеты не приходили на коммутатор s_4.

Рисунок 5 – Сравнение времени, в течение которого теряются пакеты, при установке сетевой политики с использованием различных протоколов контрольного слоя.

На рисунке 5 показаны результаты эксперимента для контроллера на основе Gen-Paxos, контроллеров Ravana и BFT-Light. Как можно видеть, только контроллер на основе Gen-Paxos предоставляет достаточные гарантии для предотвращения потерь пакетов при обновлении конфигурации сети.

Также существуют сценарии, в которых сохранение конфигурации для пакетов играет более существенную роль. Некоторые архитектуры сети требуют чтобы пакеты проходили через различные специальные устройства, такие как *брандмауэр* (firewall), *прокси* (proxy), *транслятор сетевых адресов* (network

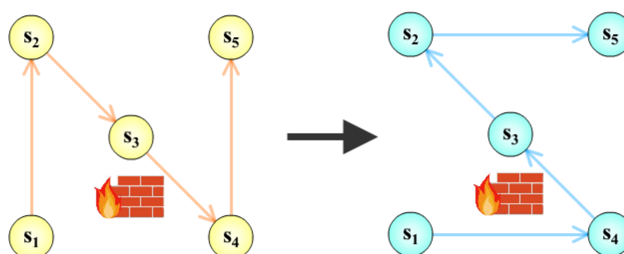


Рисунок 6 – Установка новой сетевой политики, при которой пакеты могут следовать до точки назначения в обход брандмауэра.

Рисунок 7 – Сравнение количества пакетов, прошедших до точки назначения в обход брандмауэра в ходе установки политики, при использовании различных протоколов контрольного слоя.

address translator, NAT). Нарушение данного свойства может привести к временной потере трафика или нарушению безопасности.

Был проведен эксперимент, в котором маршрут следования пакетов изменяется, при этом требуется, чтобы каждый пакет проходил через брандмауэр, в противном случае часть небезопасных пакетов может быть пропущена через сеть. На рисунке 6 показан сценарий установку обновления, изменяющего маршрут следования пакетов. В случае, если коммутатор s_4 установит обновление раньше коммутатора s_1, то часть пакетов пройдет по маршруту $s_1 \rightarrow s_4 \rightarrow s_5$, минуя брандмауэр.

Измерялась разница между объемом трафика, дошедшего до точки назначения, и объемом трафика, прошедшего через брандмауэр (одинаковые пакеты не учитывались). Результаты для различных протоколов контролеров представлены на рисунке 7. Как можно видеть, протоколы аналогичные Gen-Paxos допускают прохождение пакетов в обход брандмауэра, тем самым приводя к нарушению безопасности.

3.4. Измерение производительности и сравнение с аналогами

Методика измерения. Традиционный метод измерения производительности контроллера заключается в подключении одного или нескольких коммутаторов к контроллеру и инициализации множества входящих потоков пакетов на эти

коммутаторы. Согласно протоколу OpenFlow, если коммутатору приходит пакет из неизвестного потока, то коммутатор отправляет информацию об этом пакете на контроллер. SDN приложение, запущенное на контроллере, определяет, какая политика должна быть установлена для обработки нового потока пакетов. Такой способ управления конфигурацией сети называется реактивным.

Для симуляции описанной ситуации, как правило, используется инструмент cbench [50]. Cbench работает следующим образом:

а) симулируется несколько коммутаторов, которые подключаются к заданному контроллеру;
б) каждый коммутатор начинает генерировать и отправлять на контроллер уникальные ***Packet-In*** события, которые обозначают появление нового потока пакетов;
в) считается количество ответов контроллера.

Производительность контроллера представляется как количество событий, на которые контроллер успел ответить за единицу времени. При этом берется простое SDN приложение — *обучающийся коммутатор* (learning switch) — поэтому само приложение незначительно влияет на суммарную производительность.

Данный инструмент используется в том числе для случая присутствия нескольких контроллеров в сети [22]. При этом эмулируемые коммутаторы соединяются с одним указанным контроллером, а остальные контролеры служат лишь в качестве серверов распределенного хранилища данных.

Как правило, для оценки работы контроллера измеряются две величины: пропускная способность (единица измерения — количество обработанных событий в секунду), и задержка при репликации и установке политики (единица измерения — миллисекунды). Однако, для сравнения различных протоколов контроллера данные величины не являются показательными, поскольку они зависят как от протокола, так и от платформы контроллера, взятой за основу прототипа. На сегодняшний день существует множество платформ контроллеров, реализованных на языках от Python до C++, и различающихся по производительности на порядок и больше, что видно на рисунке 8. Чтобы сфокусироваться на оценке производительности непосредственно протокола, требуется измерять величины, не зависящие от деталей реализации.

Производительность контроллера в основном складывается из:

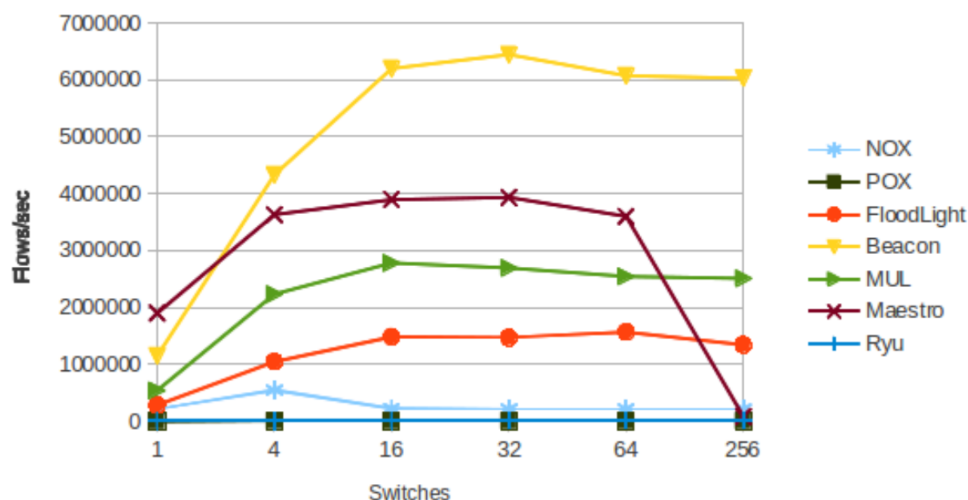


Рисунок 8 – Сравнение платформ контроллера.
A. Shalimov, Advanced study of SDN/OpenFlow controllers.

— Вычислений, связанных с логикой работы протокола;
— Операций с сетью (включая сериализацию и десириализацию сообщений);
— Операции чтения/записи в устойчивое локальное хранилище.

В случае использования подходящего языка программирования и машины достаточной производительности, вычисления и взаимодействие с локальным хранилищем занимают небольшую часть ресурсов машины по сравнению с сетевыми операциями. Поэтому, при оценке протокола следует фокусироваться на измерении сетевых взаимодействий контроллера.

В качестве оцениваемых величин предлагаются следующие:

— Задержка при репликации политики, измеряемая в коммуникационных шагах. Для этого используется инструмент netem, содержащийся в дистрибутиве linux, позволяющий эмулировать сетевые задержки на уровне операционной системы. При установлении задержки доставки в 100 миллисекунд, если установка политики контроллером занимает 500 миллисекунд, можно судить о пяти коммуникационных шагах, требуемых для репликации и установки сетевой политики.
— Сетевой трафик, используемый для установки одной политики в среднем при общении между контролерами. Единица измерения — байты на политику. При этом измеряется размер пакетов на канальном уровне модели OSI, чтобы точнее оценить эффект, оказываемый сетевыми

операциями на производительность. Сбор информации об использовании трафика происходит с помощью инструмента tcpdump.

Оценивается как суммарное использование трафика всеми контроллерами сети, так и максимальный трафик (входящий и исходящий) используемый одним узлом: первое позволяет оценить пропускную способность в случае использования топологии сети, близкой к полносвязной, второе позволяет оценить максмальную нагрузку на узел (как правило, на узел-координатор) и пропускную способность в случае использования топологии шина или звезда.

Большинство протоколов распределенного хранилища поддерживают пакетирование политик (то есть отправку нескольких политик в одном сообщении), что позволяет сократить расход трафика. Поэтому при измерении включается пакетирование, если протокол его поддерживает. Все оцениваемые контроллеры запускались с буферами, вмещающими 100 политик, и максимальным временем ожидания перед отправкой пакета установленным в несколько секунд для полного укомплектования сообщений политиками.

По описанной методике сравнивались следующие контроллеры:

а) Протокол, описанный в данной работе;
б) Ravana;
в) BFT-Light.

Для Ravana была проведена оптимизация, согласно которой по сети передаются не сами политики, а их идентификаторы, когда это возможно. Используемая реализация BFT-Light уже имеет эту оптимизацию.

Результаты. Полученные результаты показаны на графиках.

На рисунке 9 представлена задержка при применении политики, измеренная в коммуникационных шагах. Алгоритм на основе Gen-Paxos, не использующий двухфазное обновление и предоставляющий гарантии похожие на те, что дают другие протоколы, показывает лучший результат по-сравнению с аналогами; при использовании исходного алгоритма с согласованностью на уровне пакетов, получен сравнимый с другими протоколами результат.

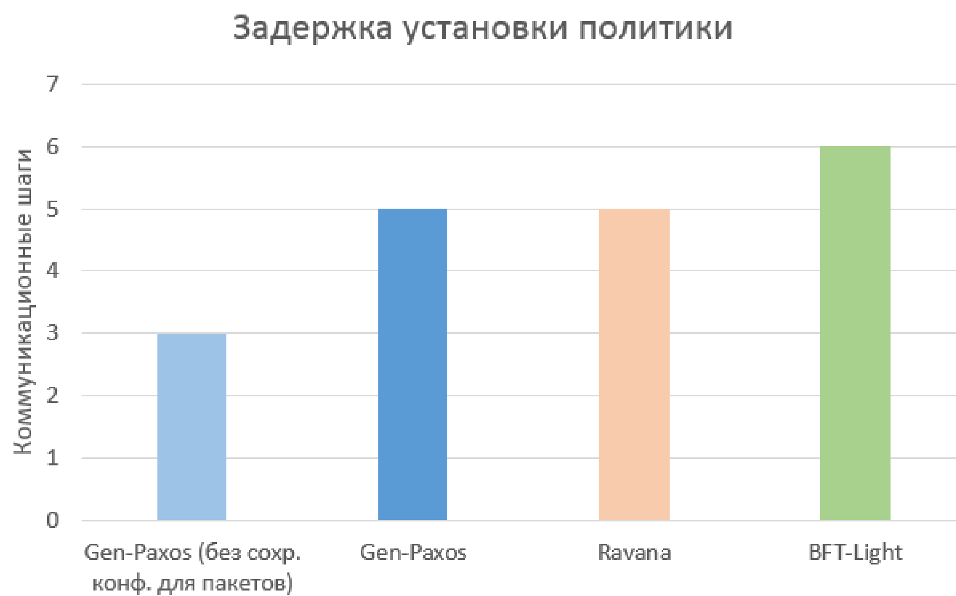


Рисунок 9 – Сравнение задержки установки политики, от момента получения запроса контроллером до установки на коммутаторы.

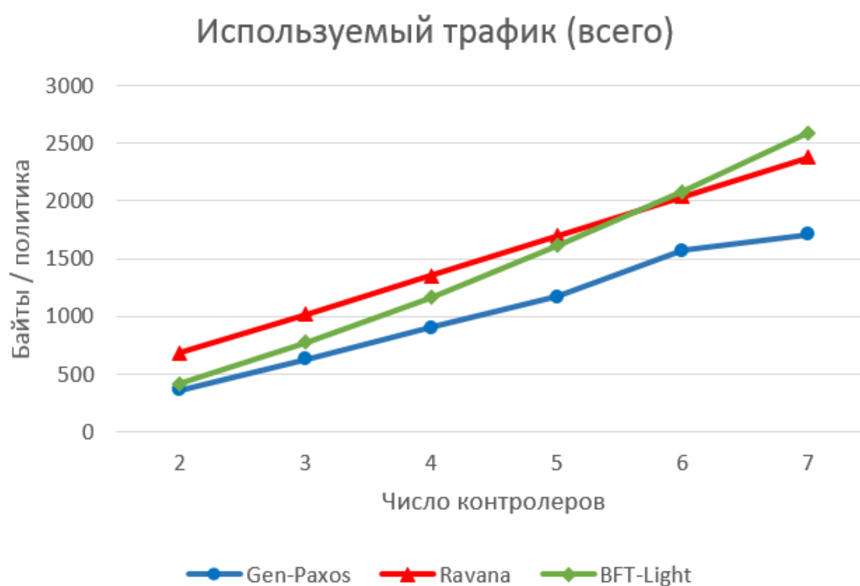


Рисунок 10 – Сравнение суммарного использования трафика.

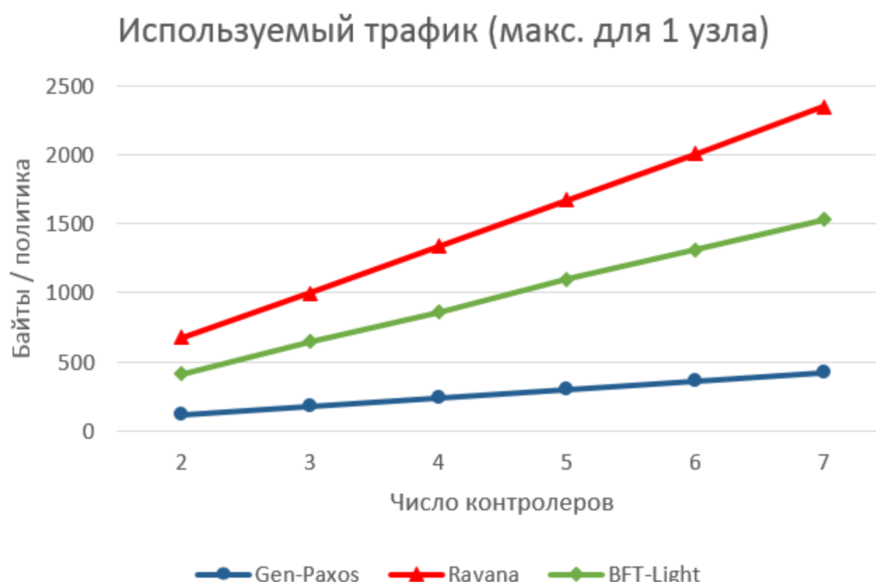


Рисунок 11 – Сравнение максимального использования трафика одним узлом (исходящий и входящий трафик).

На рисунках 10 и 11 показано использование трафика для разных протоколов контрольного слоя. Поскольку на сегодняшний день программно-конфигурируемые сети используются, как правило, в дата-центрах для достижения устойчивости сети к отказам контроллеров, то число используемых контроллеров обычно невелико. Как можно видеть, контрольный слой на основе Gen-Paxos использует меньше трафика, чем прочие прототипы, что особенно заметно при использовании сравнительно большого числа контроллеров.

Выводы по главе 3

В данной главе была описана методика экспериментальной оценки надежности и производительности протокола, были представлены результаты измерения производительности в сравнении с аналогичными контроллерами. В направлении оценки надежности было реализовано множество тестовых сценариев, описывающих использование алгоритма с различными параметрами при заданных неполадках сети. Оценка производительности показала, что при отключении двухфазного обновления и предоставлении гарантий, схожих с предоставляемыми другими протоколами, алгоритм требует меньшую задержку при репликации политики и использует меньше сетевого трафика, нежели аналогичные протоколы, и таким образом использует более оптимальную реализацию реплицированной машины состояний.

ЗАКЛЮЧЕНИЕ

В рамках данной работы был разработан алгоритм контрольного слоя программно-конфигурируемой сети, предоставляющий гарантию согласованности сохранения конфигурации для пакетов и линеаризуемости. В отсутствие гарантии сохранения конфигурации для пакетов при установке новых сетевых политик возможно появление «черных дыр» и «циклов» в маршрутах сетевых пакетов, и существующие на сегодняшний день протоколы контрольного слоя предоставляют более слабые гарантии.

Кроме того, в качестве алгоритма реплицированной машины состояний использовалась модификация Generalized Paxos, которая оптимизирован под случай редких конфликтов команд, что верно в случае программно-конфигурируемых сетей. При использовании упрощенной версии протокола с отсутствием свойств сохранения конфигурации для пакетов и линеаризуемости алгоритм демонстрирует использование меньшего числа коммуникационных шагов и трафика по сравнению с другими алгоритмами. И хотя в этом случае алгоритм предоставляет относительно слабые гарантии, вместе с использованием методики двухфазного обновления достигаются «идеальные» гарантии, поэтому мы считаем рассмотренную модификацию Generalized Paxos более оптимальным алгоритмом для хранения конфигурации сети.

Наконец, был реализован прототип описанного алгоритма контрольного слоя. Были реализованы тестовые сценарии для алгоритма Gen-Paxos в целях подтверждения его работоспособности, а также проведена оценка производительности полученного протокола контрольного слоя.

Следующие вопросы не были рассмотрены и оставлены для дальнейшей работы:

а) Расширение протокола на случай Византийской модели отказов. Напомним, что в данной модели, помимо возможности отказов узлов и потерь сообщений, часть процессов может отправлять произвольные сообщения, возможно стремясь нарушить корректность протокола.

   Существует версия алгоритма Generalized Paxos, позволяющая работу в случае Византийских ошибок [51]. Под вопросом возможность разработки протокола распределенного контрольного слоя, работающего в случае наличия контроллеров, пытающихся нарушить работу протокола.

б) Обработка политик общего вида: политики, модифицирующие заголовок пакета, и политики мониторинга.
в) Протокол, позволяющий работу в случае возможного отказа коммутаторов. Данный случай не был рассмотрен, поскольку, как правило, отказ коммутатора означает необходимость модификации или иной интерпретации конфигурации сети для предотвращения потери пакетов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Архитектура SDN. — 2016. — [Online; accessed 04-May-2018]. https://commsbusiness.co.uk/category/network-servicesservice-providers/.

2 Abstractions for network update / M. Reitblatt [и др.] // Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication. — ACM. 2012. — С. 323–334.

3 On the design of practical fault-tolerant SDN controllers / F. Botelho [и др.] // Software Defined Networks (EWSDN), 2014 Third European Workshop on. — IEEE. 2014. — С. 73–78.

4 HP OpenFlow Protocol Overview. — 2013. — [Online; accessed 16-March-2018]. https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf.

5 NOX: towards an operating system for networks / N. Gude [и др.] // ACM SIGCOMM Computer Communication Review. — 2008. — Т. 38, № 3. — С. 105–110.

6 *Erickson D.* The beacon openflow controller // Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking. — ACM. 2013. — С. 13–18.

7 Advanced study of SDN/OpenFlow controllers / A. Shalimov [и др.] // Proceedings of the 9th central & eastern european software engineering conference in russia. — ACM. 2013. — С. 1.

8 *Liang C.* Floodlight tutorial. — [Online; accessed 17-March-2018]. http://x86.cs.duke.edu/courses/fall14/compsci590.4/notes/slides_floodlight_updated.pdf.

9 Ryu. — [Online; accessed 17-March-2018]. http://osrg.github.io/ryu/.

10 *Garcia B. R.* OpenDaylight SDN controller platform // Faculty of the Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona, Universitat Politècnica de Catalunya. Barcelona. — 2015.

11 Onix: A distributed control platform for large-scale production networks. / T. Koponen [и др.] // OSDI. Т. 10. — 2010. — С. 1–6.

12 ZooKeeper: Wait-free Coordination for Internet-scale Systems. / P. Hunt [и др.] // USENIX annual technical conference. Т. 8. — Boston, MA, USA. 2010.

13 *Junqueira F. P.*, *Reed B. C.*, *Serafini M.* Zab: High-performance broadcast for primary-backup systems // Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on. — IEEE. 2011. — С. 245–256.

14 ONOS: towards an open, distributed SDN OS / P. Berde [и др.] // Proceedings of the third workshop on Hot topics in software defined networking. — ACM. 2014. — С. 1–6.

15 *Lakshman A.*, *Malik P.* Cassandra: a decentralized structured storage system // ACM SIGOPS Operating Systems Review. — 2010. — Т. 44, № 2. — С. 35–40.

16 TinkerPop Blueprints. — [Online; accessed 17-March-2018]. https://github.com/jdellithorpe/blueprints-ramcloud-graph.

17 The case for RAMClouds: scalable high-performance storage entirely in DRAM / J. Ousterhout [и др.] // ACM SIGOPS Operating Systems Review. — 2010. — Т. 43, № 4. — С. 92–105.

18 The case for reliable software transactional networking / M. Canini [и др.] // arXiv preprint arXiv:1305.7429. — 2013.

19 Software transactional networking: Concurrent and consistent policy composition / M. Canini [и др.] // Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking. — ACM. 2013. — С. 1–6.

20 Design and implementation of a consistent data store for a distributed SDN control plane / F. Botelho [и др.] // Dependable Computing Conference (EDCC), 2016 12th European. — IEEE. 2016. — С. 169–180.

21 *Castro M.*, *Liskov B.* Practical Byzantine fault tolerance and proactive recovery // ACM Transactions on Computer Systems (TOCS). — 2002. — Т. 20, № 4. — С. 398–461.

22 Ravana: Controller fault-tolerance in software-defined networking / N. Katta [и др.] // Proceedings of the 1st ACM SIGCOMM symposium on software defined networking research. — ACM. 2015. — С. 4.

23 A distributed and robust sdn control plane for transactional network updates / M. Canini [и др.] // Computer Communications (INFOCOM), 2015 IEEE Conference on. — IEEE. 2015. — С. 190–198.

24 Virtual time and global states of distributed systems / F. Mattern [и др.] // Parallel and Distributed Algorithms. — 1989. — Т. 1, № 23. — С. 215–226.

25 *Fischer M. J.*, *Lynch N. A.*, *Paterson M. S.* Impossibility of distributed consensus with one faulty process // Journal of the ACM (JACM). — 1985. — Т. 32, № 2. — С. 374–382.

26 *Browne J.* Brewer's CAP theorem // J. Browne blog. — 2009.

27 *Thomas R. H.* A majority consensus approach to concurrency control for multiple copy databases // ACM Transactions on Database Systems (TODS). — 1979. — Т. 4, № 2. — С. 180–209.

28 *Lamport L.* The part-time parliament // ACM Transactions on Computer Systems (TOCS). — 1998. — Т. 16, № 2. — С. 133–169.

29 Paxos made simple / L. Lamport [и др.] // ACM Sigact News. — 2001. — Т. 32, № 4. — С. 18–25.

30 *Lamport L.* Fast paxos // Distributed Computing. — 2006. — Т. 19, № 2. — С. 79–103.

31 *Camargos L. J.*, *Schmidt R. M.*, *Pedone F.* Multicoordinated paxos // Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing. — ACM. 2007. — С. 316–317.

32 *Lamport L. B.* Generalized paxos. — 413.2010. — US Patent 7,698,465.

33 Resilience of sdns based on active and passive replication mechanisms / P. Fonseca [и др.] // Global Communications Conference (GLOBECOM), 2013 IEEE. — IEEE. 2013. — С. 2188–2193.

34 *Sutra P.*, *Shapiro M.* Fast genuine generalized consensus // Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on. — IEEE. 2011. — С. 255–264.

35 *Mao Y.*, *Junqueira F. P.*, *Marzullo K.* Mencius: building efficient replicated state machines for WANs // OSDI. Т. 8. — 2008. — С. 369–384.

36 Fast mencius: Mencius with low commit latency / W. Wei [и др.] // INFOCOM, 2013 Proceedings IEEE. — IEEE. 2013. — С. 881–889.

37 *Moraru I.*, *Andersen D. G.*, *Kaminsky M.* There is more consensus in egalitarian parliaments // Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. — ACM. 2013. — С. 358–372.

38 Speeding up consensus by chasing fast decisions / B. Arun [и др.] // Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on. — IEEE. 2017. — С. 49–60.

39 Making fast consensus generally faster / S. Peluso [и др.] // Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on. — IEEE. 2016. — С. 156–167.

40 *Mijiddorj D.*, *Tarigan I. D. F.*, *Kim D.-S.* Fast-Failover Mechanisms using Parenthood Distribution Controllers // Proceedings of Symposium of the Korean Institute of communications and Information Sciences. — 2016. — С. 212–213.

41 *Voellmy A.* OpenFlow protocol implementation on Haskell. — 2015. — [Online; accessed 04-May-2018]. https://github.com/AndreasVoellmy/openflow.

42 https://github.com/Martoon-00/sdn-policy.

43 *Ben Pfaff J. P. e. a.* Implementation of Open vSwitch. — 2009. — [Online; accessed 04-May-2018]. https://github.com/openvswitch/ovs.

44 *Gupta P.*, *Vishwakarma R. G.* Comparison of Various Election Algorithms in Distributed System // International Journal of Computer Applications. — 2012. — Т. 53, № 12.

45 *Ongaro D.*, *Ousterhout J. K.* In search of an understandable consensus algorithm. // USENIX Annual Technical Conference. — 2014. — С. 305–319.

46 *Balhara S.*, *Khanna K.* Leader election algorithms in distributed systems // Journal of Computer Science and Information Technology, IJCSMC. — 2014. — Т. 3, № 6. — С. 374–379.

47 *Tanenbaum A. S.*, *Woodhull A. S.* Operating systems: design and implementation. Т. 2. — Prentice-Hall Englewood Cliffs, NJ, 1987.

48 How to scale exponential backoff: Constant throughput, polylog access attempts, and robustness / M. A. Bender [и др.] // Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms. — Society for Industrial, Applied Mathematics. 2016. — С. 636–654.

49 *al. B. O. et*. Emulator for rapid prototyping of Software Defined Networks. — 2009. — [Online; accessed 04-May-2018]. https://github.com/mininet/mininet.

50 *Khattak Z. K.*, *Awais M.*, *Iqbal A.* Performance evaluation of OpenDaylight SDN controller // Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on. — IEEE. 2014. — С. 671–676.

51 *Pires M.*, *Ravi S.*, *Rodrigues R.* Generalized Paxos Made Byzantine (and Less Complex) // International Symposium on Stabilization, Safety, and Security of Distributed Systems. — Springer. 2017. — С. 203–218.