Аннотация

Raft — алгоритм для решения задачи консенсуса в сети ненадёжных вычислений. Протокол широко используется в современных распределённых системах. Существует множество реализаций Raft с открытым исходным кодом на разных языках программирования, а также множество предложений по улучшению алгоритма. Цель данного исследования - рассмотреть такие предложения, на их основе реализовать улучшенную версию алгоритма и проанализировать последствия изменений на поведение конкретной системы.

Содержание

1 Консенсус в распределённых системах

1.1 Задача консенсуса

Консенсус - фундаментальная задача в распределённых отказоустойчивых системах. Достижение консенсуса подразумевает согласование некоторого хранимого значения набором серверов. При этом принимаемое в распределённой системе решение по значению должно быть окончательным: после достижения консенсуса по хранимому значению оно не должно меняться. Обычно алгоритмы консенсуса позволяют принимать решения до тех пор, пока большинство серверов распределённой системы продолжают быть доступными. Например, кластер из 7 серверов продолжит работать в случае потери 3 узлов. Если больше серверов выйдет из строя, система не сможет достигать консенсуса по новым операциям, но при этом она не должна возвращать ложные значения.

Задача консенсуса обычно возникает в контексте реплицируемых конечных автоматов (replicated state machines), используемых при построении отказоустойчивых распределённых систем.

1.2 Реплицированный конечный автомат

Формально конечный автомат определяется в виде кортежа:
$A = (S, X, Y, \delta, \lambda)$, где

- S — конечное множество состояний автомата;
- $s_0 \in S$ - начальное состояние автомата;
- X, Y — конечные входной и выходной алфавиты соответственно, из которых формируются строки, считываемые и выдаваемые автоматом;
- $\delta : S \times X \to S$ — функция переходов;
- $\lambda : S \times X \to Y$ — функция выходов.

Соответственно работа автомата выглядит следующим образом. Конечный автомат начинает работу в обозначенном состоянии s_0 и принимает на вход операции $x_i \in X$. Каждый полученный ввод x_i подается на вход в функцию перехода δ и функцию вывода λ для перехода в новое состояние $\delta(x_i) = s_i \in S$ и генерацию вывода $\lambda(x_i) = y_i \in Y$.

Конечный автомат называется детерминированным, если следующее состояние и выходное значение автомата однозначно определяется текущим состоянием и входным символом. То есть, если несколько копий одного и того же детерминированного конечного автомата стартуют в одинаковом состоянии и получают на вход одни и те же операции в одном и том же порядке, то в результате они придут в одинаковое конечное состояние, сгенерировав одинаковые последовательности выходных данных. Далее будут рассматриваться только детерминированные конечные автоматы.

1.3 Реплицированный лог операций

Реплицированные конечные автоматы обычно реализуются методом репликации лога операций. Каждый сервер распределённой системы хранит упорядоченную серию входных команд, которые выполняет реплика конечного автомата.

В таком подходе каждый сервер кластера хранит реплицированный конечный автомат и лог операций, а конкретный алгоритм консенсуса призван достичь отказоустойчивость автомата. То есть с точки зрения клиента работа с распределённой системой должна выглядеть как взаимодействие с одним надежным конечным автоматом даже в случае отказа части узлов.

На каждом сервере распределённой системы конечный автомат применяет операции из лога операций. В такой системе алгоритм консенсуса используется при согласовании операций содержащихся в логе. Более подробно: если любая реплика конечного автомата в кластере применяет n-ую операцию, то никакая другая реплика не должна применить отличную n-ую

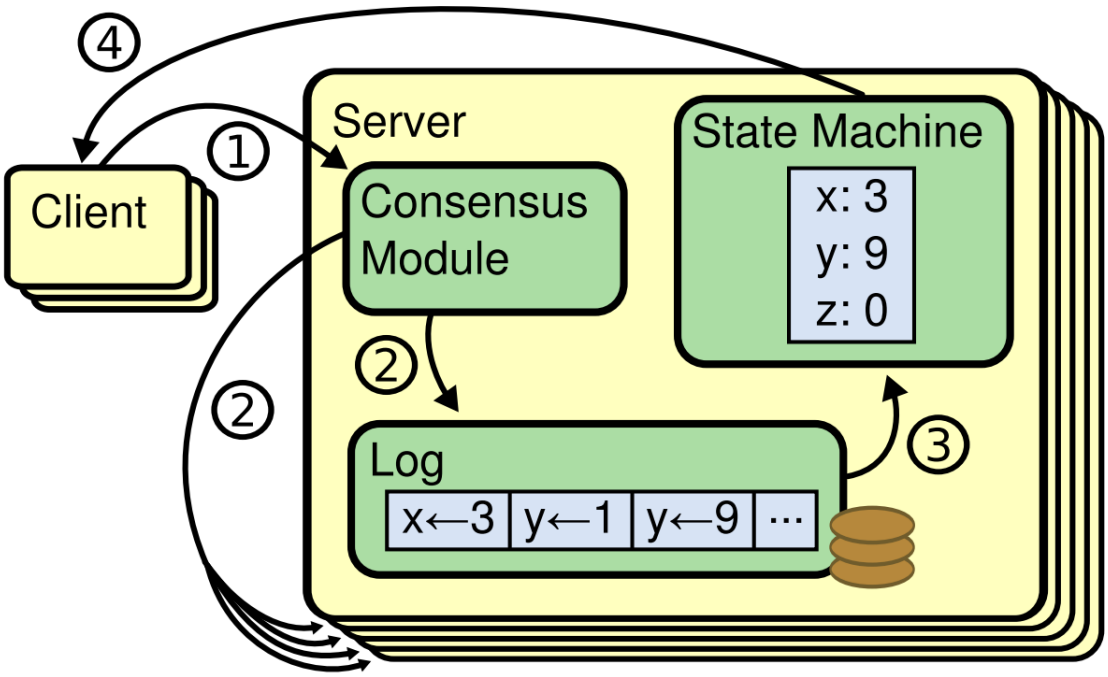

Рис. 1: Реплицируемый конечный автомат

операцию. Таким образом все реплики конечного автомата применяют одинаковый набор операций и, следовательно, выдают одну и ту же серию результатов и приходят к одной и той же серии состояний.

На рис.1 представлена схема реплицируемого конечного автомата реализованного с помощью реплицируемого лога операций. Каждый сервер содержит три основные компоненты: конечный автомат (state machine), лог операций (log) и модуль достижения консенсуса (consensus module). Запрос клиента, который влияет на состояние автомата, поступает на согласующий модуль. При достижении консенсуса между узлами системы операция добавляется в лог. Конечный автомат выполняет операции из лога.

2 Протокол Raft

2.1 Особенности протокола

Raft - алгоритм решающий задачу консенсуса. Алгоритм Raft описан в статье [1]. На фоне других протоколов консенсуса Raft выделяют следующие особенности.

Strong leader: Лидер в протоколе Raft имеет больший контроль над состоянием системы, чем в других протоколах. Например, записи в лог передаются только от лидера к остальными узлам. Это упрощает работу с логом и делает алгоритм более простым для понимания.

Leader election: В Raft процесс выбора лидера использует случайные промежутки времени. Такой механизм является несущественной надстройкой над механизмом отправки heartbeat-ов. При этом он решает возникающие в распределённой системе конфликты быстро и эффективно.

Membership changes: Изменение состава кластера выполняется при помощи нового подхода, основанного на совместном консенсусе двух разных конфигураций. Это позволяет распределённой системе продолжать работать при изменении конфигурации.

В протоколе Raft узлы распределённой системы могут находиться в трех состояниях: лидер (leader), кандидат (candidate), последователь (follower). На рис.2 изображены условия переходов между состояниями. В начале работы в кластере нет лидера, каждый узел кластера находится в состоянии последователя. Соответственно, консенсус реализуется через выбор лидера распределённой системы, который отвечает за репликацию лога операций на остальные узлы системы. В любой момент времени в кластере может быть максимум один лидер. Лидер также принимает новые операции от клиентов и сообщает последователям, когда реплицированные операции из лога можно применять на конечном автомате.

Описанное использование одного лидера упрощает работу с реплицированным логом. Лидер единолично определяет, куда в лог будут записаны новые операции, без обращений к остальным узлам распределённой системы. Таким образом информация в системе распространяется от лидера к остальным узлам. Лидер регулярно уведомляет последователей о своём существовании отправляя heartbeat-ы. Последователи имеют заранее установленный промежуток времени timeout, в который они ожидают получить сиг-

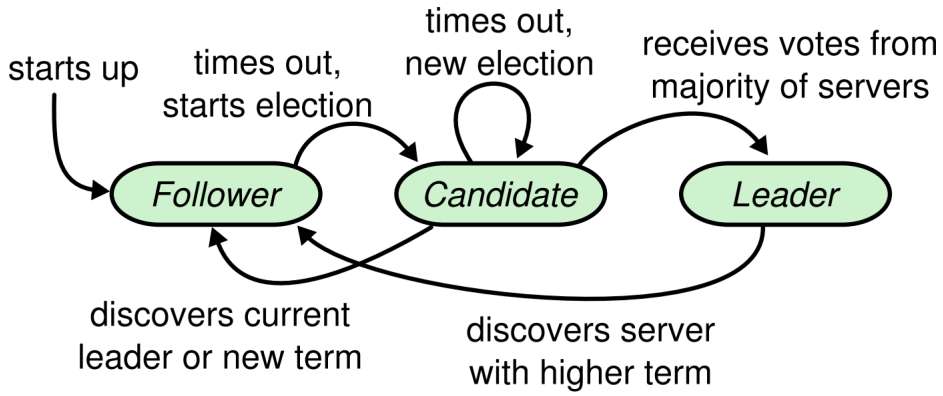

Рис. 2: Состояния узлов в протоколе Raft

нал от лидера. Когда поступает очередной heartbeat ожидание перезапускается. Если в отведённое время heartbeat получен не был, то узел сменяет свой статус с последователя на кандидата, и начинается процесс голосования.

Задача консенсуса в алгоритме Raft разбивается на две подзадачи: выбор лидера (leader election) и репликация лога (log replication).

2.2 Leader election

При инициализации алгоритма и при выходе из строя текущего лидера, появляется необходимость выбрать нового лидера.

В Raft все время работы распределённой системы разбивается на пронумерованные, монотонно растущие промежутки времени - term-ы. Каждый сервер хранит текущий term и отправляет его при обмене сообщениями с другими узлами. При получении большего значения term-а узел Raft обновляет своё текущее значение на большее. Term-ы выступают в роли логических часов и позволяют серверам распознавать устаревшую информацию и отличать не актуальных лидеров. Например, в ситуациях, когда лидер кластера отключился, был заменен, но вернулся в строй и снова отправляет сообщения. Такие сообщения будут сопровождаться старым term-ом, по которому их можно отсеять. На рис.3 изображено разделение работы кластера на term-ы.

Каждый term начинается с выборов лидера. Если выборы завершены успешно и избран один лидер, term остается актуальным и продолжается обычная работа алгоритма по репликации лога под контролем нового лидера. В случае провала выборов, начинается новый term с новыми выборами.

Выборы лидера запускаются сервером кандидатом. Сервер становится кандидатом, когда он не получает сообщений от лидера в течение периода времени - heartbeat timeout. Он начинает выборы с увеличения счетчика term, голосования за себя как нового лидера и отправки сообщения RequestVote всем остальным серверам с запросом отдать ему голос. В каждый отдельный term узел распределённой системы может проголосовать только за одного кандидата и делает это по принципу first-come-first-served, отдавая голос первому пришедшему кандидату. Если кандидат получает сообщение от другого сервера с term-ом, превышающим текущий term кандидата, то выборы считаются проигранными, кандидат возвращается в статус последователя и считает отправителя сообщения новым лидером. Кандидат собирает голоса в течение периода времени - election timeout. Если кандидат получает большинство голосов, то он становится новым лидером. Если ни того, ни другого не происходит, например, из-за разделения голосов, начинается новый срок и начинаются новые выборы.

Raft решает проблему с разделением голосов использование рандомизированных таймаутов. Это снижает вероятность разделения голосов, поскольку серверы не станут кандидатами одновременно: таймер на одном из серверов истечет раньше, сервер выиграет выборы, затем станет лидером и отправит сообщения на другие узлы, прежде чем большинство последователей смогут стать кандидатами.

2.3 Log replication

За репликацию лога отвечает лидер. Он принимает запросы клиентов, которые содержат команды для реплицированного конечного автомата. Сна-

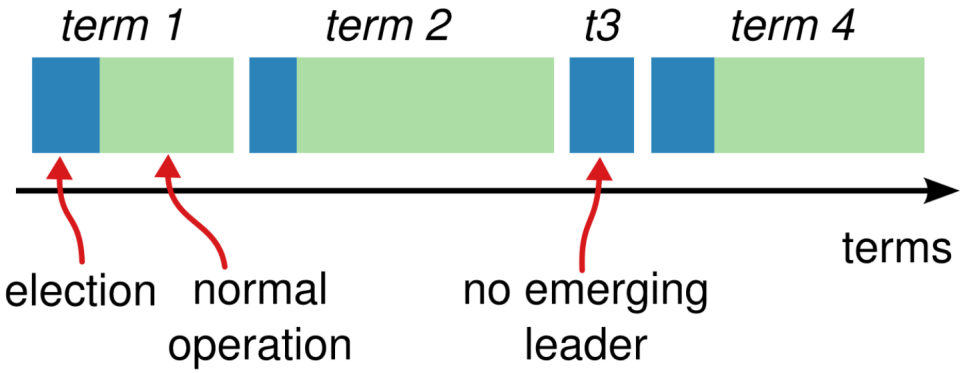

Рис. 3: Разделение работы кластера на term-ы

чала лидер добавляет операцию в свой локальный лог, затем рассылает её всем узлам кластера в виде AppendEntries сообщений. В случае недоступности последователей лидер бесконечно отправляет сообщения до тех пор, пока запись не будет сохранена в лог всеми узлами распределённой системы. На рис.4 показана схема репликации лога операций.

Когда лидер получает подтверждение о записи новой операции от более чем половины узлов кластера, он применяет эту операцию к своему локальному конечному автомату и она считается зафиксированной (commited). Если в алгоритме Raft операция зафиксирована, то считаются зафиксированы и все предыдущие операции в логе. Когда информация о том, что новая операция зафиксирована доходит до узла последователя, он применяет ее к своему локальному конечному автомату. Этот механизм обеспечивает репликацию и согласованность журналов между всеми серверами кластера.

В случае сбоя лидера логи операций могут прийти в несогласованное состояние. Более конкретно: в сценарии, когда лидер не успел распространить новую запись на большинство серверов и, следовательно, она не является зафиксированной. Устранять такое расхождение будет новый избранный лидер, распространяя свою версию журнала. Для этого лог операций лидера и последователя сравнивается, в них ищется самая поздняя совпадающая операция. В логе последователя удаляются все операции, которые идут по-

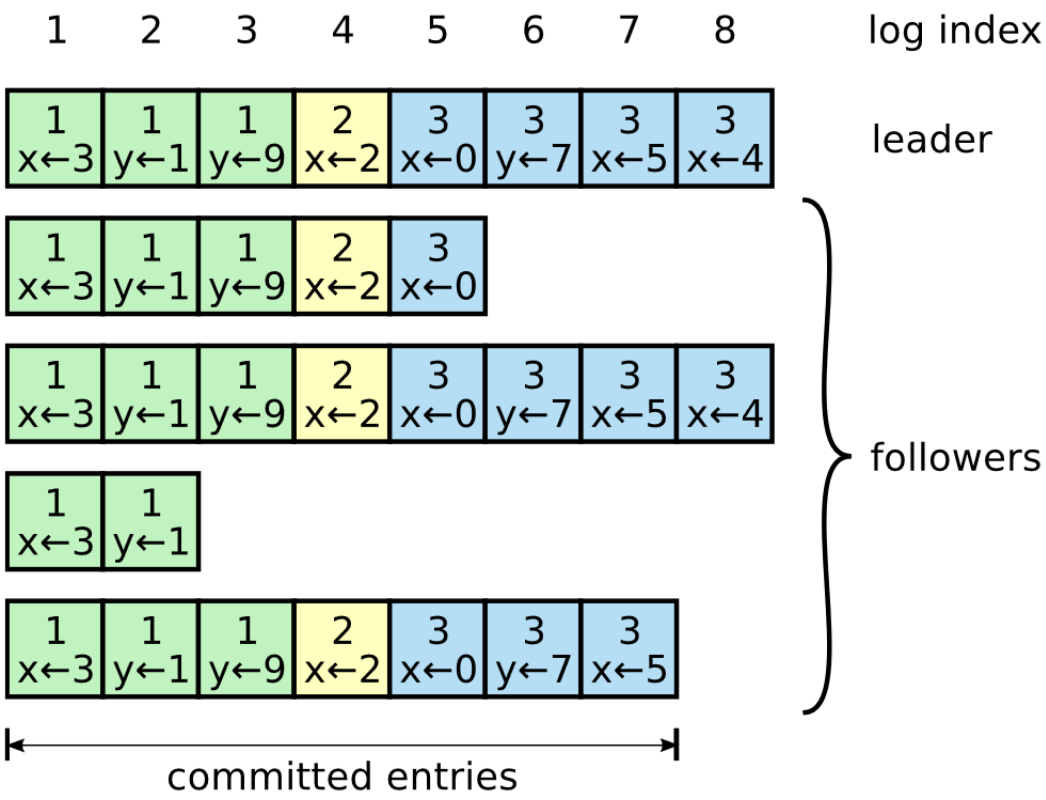

Рис. 4: Репликация лога операций

сле найденного совпадения, и начинается копирование лога лидера. Такой механизм позволяет восстановить согласованность логов после сбоев в кластере.

2.4 Гарантии

Протокол Raft гарантирует, что каждое из следующих свойств верно в любой момент времени:

- **Election Safety:** в каждый срок может быть выбран максимум один лидер
- **Leader Append-Only:** лидер не переписывает свои лог, он только добавляет в него новые записи
- **Log Matching:** если два лога содержат запись с одинаковым индексом и term-ом, то эти логи совпадают от начала и до данного индекса

- **Leader Completeness:** если запись в логе зафиксирована, то эта запись будет присутствовать в логах всех последующих лидеров
- **State Machine Safety:** если сервер распределённой системы применяет запись из лога на конечный автомат, то никакой другой сервер не применит другую операцию под тем же индексом

3 Обзор предложений по улучшению Raft

3.1 Группировка предложений

Рассматривая предложения по улучшению протокола Raft можно выделить три основные группы.

Первая группа объединяет предложения связанные с повышением безопасности алгоритма и его устойчивости при возможных атаках. В этой группе часто рассматривается улучшения Raft позволяющие решать задачу византийских генералов (Byzantine generals problem, Byzantine fault tolerance) и использование blockchain методов.

Вторая группа объединяет улучшения связанные с особенностями конкретных используемых сетевых технологий. Например, улучшения в контексте интернета вещей (internet of things), шестого поколения мобильной связи 6G, различных беспроводных сетей и даже в таких специфичных сценариях как сети зарядных станций для электромобилей.

Третья группа является самой обширной и включает в себя общие улучшения эффективности различных механизмов алгоритма Raft. Эти улучшения в свою очередь можно разделить на две подгруппы: улучшения механизма репликации лога и улучшения механизма выбора лидера.

3.2 Влияние лидера на производительность системы

Как было рассмотрено ранее, одной из особенностей алгоритма Raft является **strong leadership**. Данное свойство упрощает алгоритм, но в то же

Features \ Research	Raft	Raft-PLUS	Pirogue	hhRaft	Weighted RAFT	AdRaft	KRaft
Improvements	\	Leader election	Leader election	Leader election and safety	Leader election	Leader election and log replication	Leader election and log replication
Voting method	FCFS [1]	VBW [2]	FCFS	FCFS	VBW	FCFS	CBT [3]
Dynamically monitor cluster	no	yes	yes	yes	no	no	no
Active election	no	yes	no	no	no	no	no
Resist Byzantine attacks	no	no	no	yes	no	no	no

[1] First Come First Serve [2] Voting By Weight [3] Calculate By Table.

Рис. 5: Сравнение различных версий протокола Raft

время производительность лидера напрямую влияет на производительность всего кластера. Однако в алгоритме Raft при выборе лидера узел кластера отдаст свой голос кандидату, чей запрос пришёл первым, то есть по принципу first-come-first-served. В таком подходе производительность лидера не берется во внимание.

Другой особенностью алгоритма является тот факт, что механизм выбора нового лидера запускается, только тогда, когда пропадает контакт с текущим лидером - перестают приходить heartbeat-ы. Уязвимость такого подхода проявляется в ситуациях, когда текущий лидер теряет производительность, но не отключается окончательно. То есть лидер не будет переизбран и производительность всего кластера понизится.

Именно поэтому одним из направлений по улучшению протокола является набор предложений по созданию механизмов, гарантирующих, что распределённая система имеет производительного лидера.

В таблице на рис.5 представлено сравнение различных предложений по улучшению протокола Raft связанных с выбором и работой лидера. Большинство версий сохраняют first-come-first-served подход выбора лидера и не отслеживают состояние кластера, которое может динамично изменяться. Относительно обозначенных свойств в данном сравнении выделяется алгоритм Raft-PLUS.

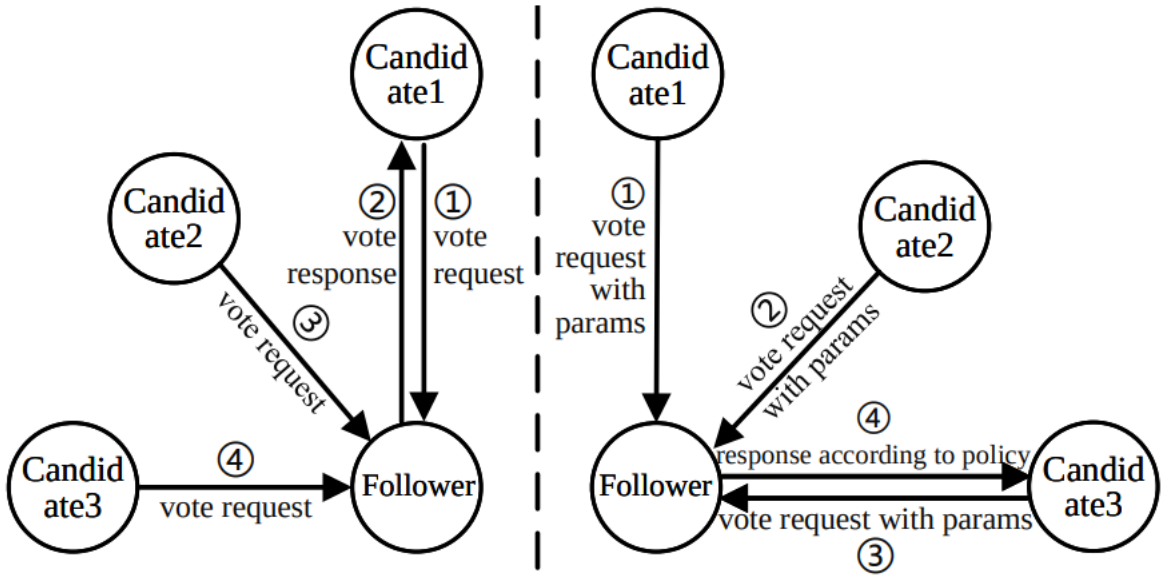

Рис. 6: Последователь после получения запроса в Raft (слева) и Raft-PLUS (справа) алгоритм.

4 Протокол Raft-PLUS

Протокол Raft-PLUS предлагает решение описанных проблем, в основе которого лежит разделение работы кластера на две фазы: фаза выбора лидера и фаза оппозиции существующему лидеру. Raft-PLUS описан в статье [2].

4.1 Фаза выбора лидера

Фаза выбора лидера очевидным образом является модификацией оригинального механизма выбора лидера. Основным отличием является отказ от принципа голосования first-come-first-served. Узел, который получил RequestVote сообщение от кандидата, не сразу отдаст свой голос, а запустит процесс сбора голосов от других кандидатов. На рис.6 представлено сравнение двух подходов голосования.

Этот процесс должен быть лимитирован по времени, чтобы голосование не выходило за рамки таймаута выборов и могло успешно закончиться. Авторы протокола предлагают заканчивать сбор голосов при получении запросов на голосование от трети кластера или при прохождении промежутка

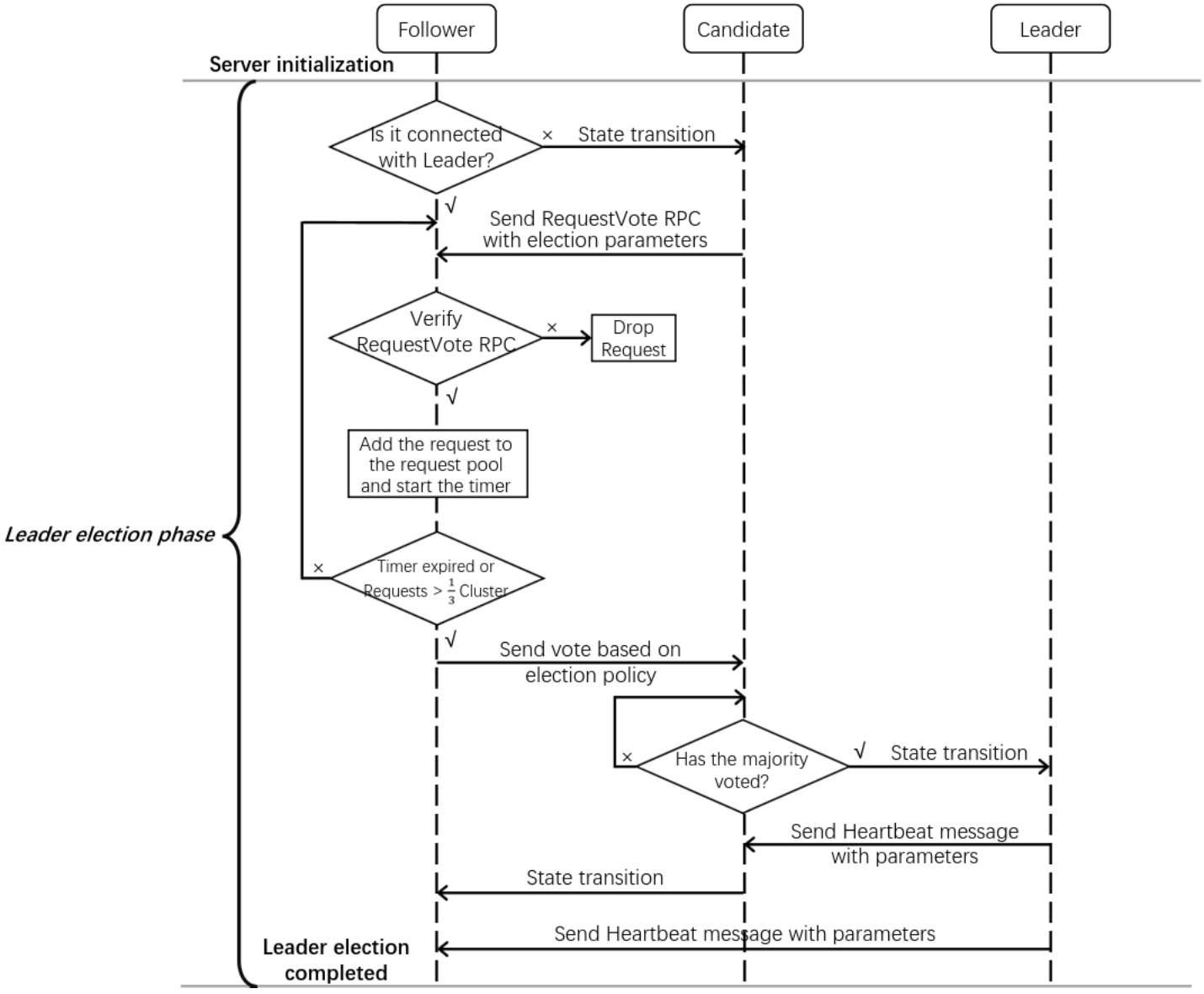

Рис. 7: Фаза выбора лидера в Raft-PLUS

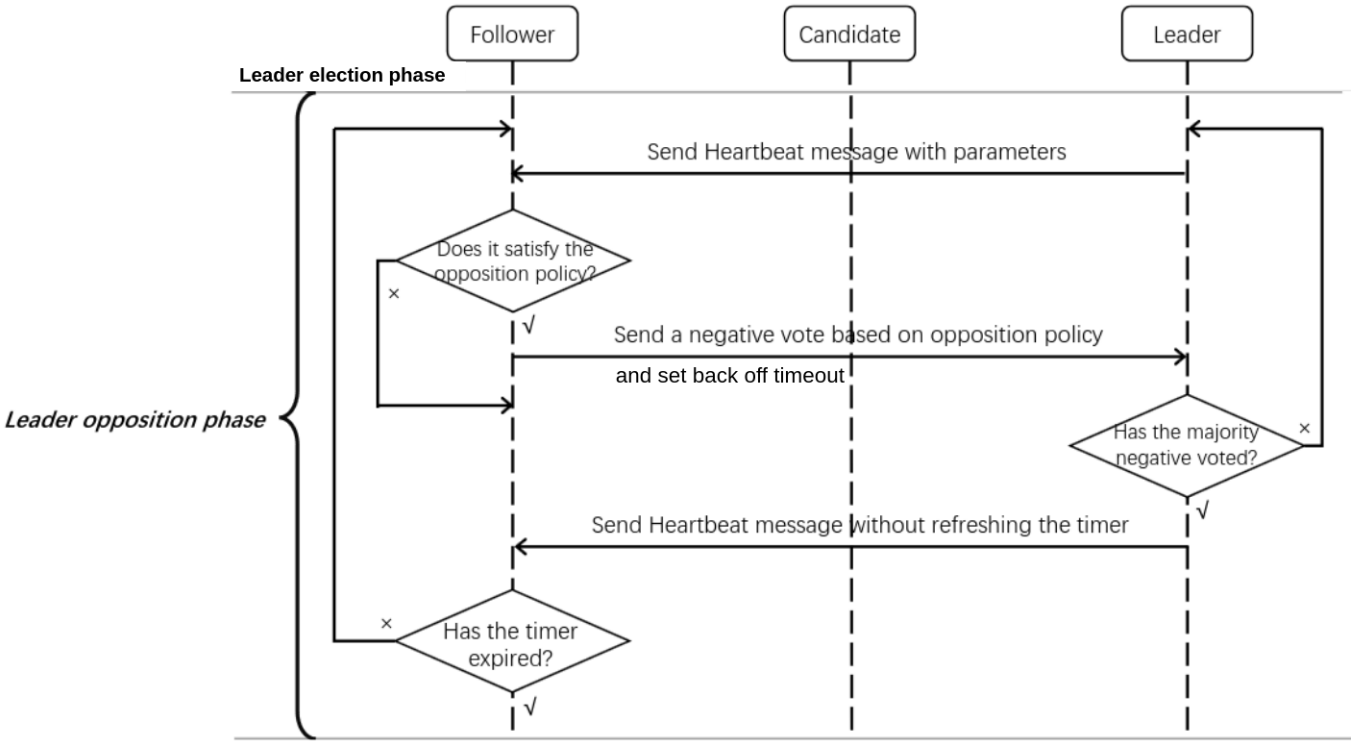

Рис. 8: Фаза оппозиции лидеру в Raft-PLUS

времени равного трети таймаута выборов. Когда сбор голосов завершается, узел определяет лучший вариант - сортирует кандидатов по заранее заданной метрике и отдает свой голос победителю. На рис.7 представлена подробная схема выборов лидера в алгоритме Raft-PLUS.

4.2 Фаза оппозиции лидеру

Фаза оппозиции лидеру является модификацией обычной работы кластера с выбранным лидером. Отличие заключается в том, что последователи могут свергнуть лидера, если его производительность перестанет их устраивать. Каждый узел кластера следит за работой лидера и оценивает её по заранее заданной метрике и, если лидер начинает испытывать проблемы, отправляет так называемый негативный голос лидеру. В случае, если лидер собирает негативные голоса от большинства распределённой системы, он завершает своё лидерство. При этом каждый последователь может отправить только один негативный голос за term, чтобы оппозиция определялась имен-

но большинством узлов, а не повторными сообщениями. На рис.8 представлена подробная схема оппозиции лидеру в алгоритме Raft-PLUS.

4.3 Метрики

В обеих фазах производительность узлов кластера рассматривается по собранным метрикам. При этом одни метрики подходят как для выбора кандидата, так и для свержения лидера, в то время как другие специфичны для одной из фаз. Важно также отметить, что узлы в кластере не обязательно должны использовать одну и ту же метрику. Напротив, использование разных метрик приводит к разносторонней оценки производительности. Более того, один сервер может использовать разные метрики при выборе лидера и при оппозиции.

Метрики, которые можно использовать в обеих фазах:

1. Скорость записи операции на диск
2. Доступная оперативная память
3. Утилизация процессора
4. Сетевая задержка между отправки сообщения между кандидатом и голосующим узлом

Метрики специфичные для голосования:

1. Количество принятых клиентских запросов за все время
2. Количество прошлых выигранных term-ов

Метрики специфичные для оппозиции:

1. Количество принятых клиентских запросов за текущий term
2. Скорость фиксации операции

5 Consul

Для изучения влияния предложенных в Raft-PLUS улучшений на поведение конкретной распределённой системы в данном исследовании было выбрано приложение Consul.

Consul [3] - приложение предоставляющее функционал service discovery. Приложение написано на языке программирования Go, активно развивается, имеет версию с открытым исходным кодом и платную версию, поддерживающую дополнительный функционал.

Для решения задачи консенсуса разработчики Consul-а используют свою реализацию протокола Raft. Это реализация была дополнена предложенными в Raft-PLUS модификациями и протестирована в работе с Consul-ом

5.1 Service discovery

Service discovery - процесс автоматического обнаружения сервисов в распределённой системе и мониторинга их состояния. Service discovery включает в себя регистрацию и поддержку актуальной информации о всех сервисах в service catalog-е. Такой service catalog выступает в роли единого источника информации, который позволяет сервисам в распределённой системе общаться и отправлять друг другу запросы.

Среди преимуществ использования service discovery подхода в распределённой системе обычно выделяю следующие пункты.

- Автоматическое обнаружение сервисов позволяет использовать динамичные IP адресов и порты.
- Упрощенная логика горизонтального масштабирования. Добавляемые сервисы будут автоматически включены в работу приложения.
- Вынесение логики обнаружения сервисов из приложения.

- Повышение отказоустойчивости за счёт постоянной проверки состояния сервисов.
- Балансировка нагрузки путем распределения запросов по набору соответствующих сервисов.

5.2 Архитектура

Архитектурно Consul делится на две части - control plane и data plane. Control plane позволяет регистрировать, запрашивать и защищать сервисы, развернутые в кластере. Именно control plane поддерживает актуальный service catalog содержащий информацию о сервисах и их IP адресах. Data plane содержит сами сервисы приложения.

Control plane состоит из узлов агентов (Consul agents), которые образуют распределённую систему. Агенты разделяются на два типа: серверные (server agents) и клиентские (client agents).

5.3 Server agents

Серверные агенты Consul-а хранят всю информацию о состоянии системы, включая IP адреса узлов приложения, информацию об их состоянии и работоспособности, конфигурационную информацию. Для обеспечения сохранности данных и отказоустойчивости системы серверные агенты согласуют свою работу с помощью протокола Raft. В отличие от клиентских, серверные агенты работают на выделенных узлах, поскольку они более ресурсоемки.

5.4 Client agents

Клиентские агенты — это легковесные процессы, составляющие большую часть Consul кластера. Клиентские агенты работают на каждом узле, где работают сервисы приложения. Они собирают информацию о работе серви-

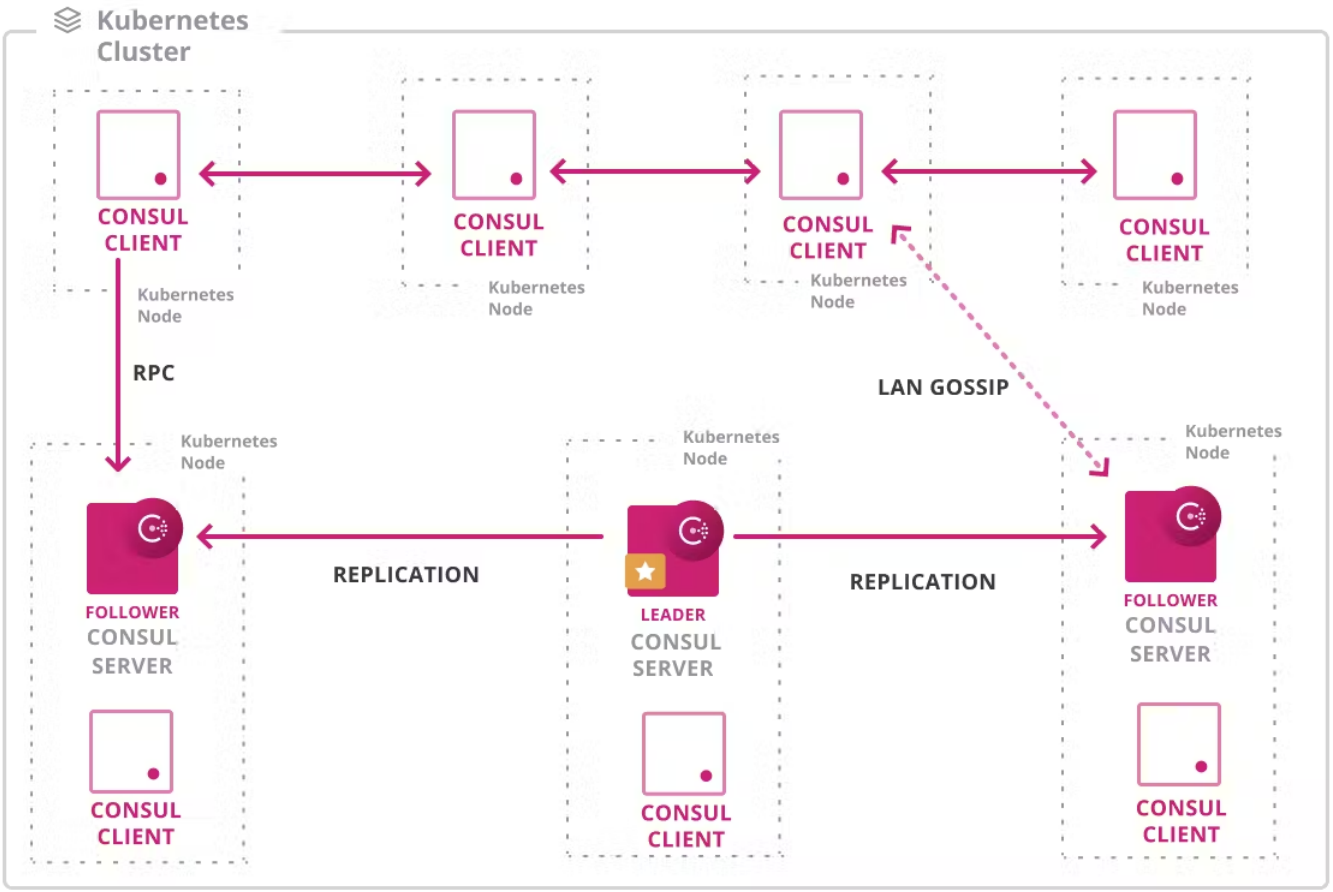

Рис. 9: Архитектура кластера Consul с использованием Consul Dataplane

сов и сообщают её серверным агентам. Таким образом, клиентские агенты активно взаимодействуют с серверными и почти не поддерживают собственного состояния. Для взаимодействия с серверными агентами клиентские используют удаленные вызовы процедур (RPC).

5.5 Gossip protocol

Помимо описанных механизмов и серверные и клиентские агенты также общаются друг с другом по gossip протоколу. С помощью него распространяется информация о работе узлов самого Consul-a. Например, информация о членстве позволяет клиентским серверам автоматически обнаруживать серверные, сокращая объем необходимой настройки. На рис.9 представлена общая архитектура Consul-a на kubernetes кластере.

6 Использование Raft-PLUS в Consul

Для тестирования работы Consul-a, использующего реализованную Raft-PLUS версию алгоритма, приложение было запущено на Kubernetes кластере в облачной платформе Yandex Cloud. Репозиторий с кодом реализации [4]. Тестирование проводилось с 9-ю серверными агентами.

Kubernetes — приложение с открытым исходным кодом предназначенное для оркестровки контейнеризированных приложений. Kubernetes позволяет автоматизировать их развёртывание, масштабирование и координацию в условиях кластера. Поддерживает основные технологии контейнеризации, включая Docker, rkt, а также возможные технологии аппаратной виртуализации.

6.1 Litmus Chaos

Для тестирования приложения в условиях высокой нагрузки и сбоев узлов используется приложение для организации хаос-инжиниринга Litmus Chaos [5].

Хаос-инжиниринг - это дисциплина, направленная на тестирование и повышение отказоустойчивости сложных распределенных систем путем проведения контролируемых экспериментов, имитирующих реальные сценарии отказов. Сценарии включают в себя падение производительности узлов через повышение нагрузки на встроенную и оперативную память, повышение утилизации процессора, создание проблем с сетью между серверами кластера. Такой подход помогает выявить и устранить потенциальные проблемы до того, как они появились и могли бы привести к значительным сбоям, сократить время простоя и повысить общую доступность систем.

Фундаментальными элементами архитектуры Litmus Chaos являются хаос-эксперименты. Хаос-эксперимент - это набор различных операций, объединенных вместе для достижения желаемого эффекта хаоса в кластере Kubernetes.

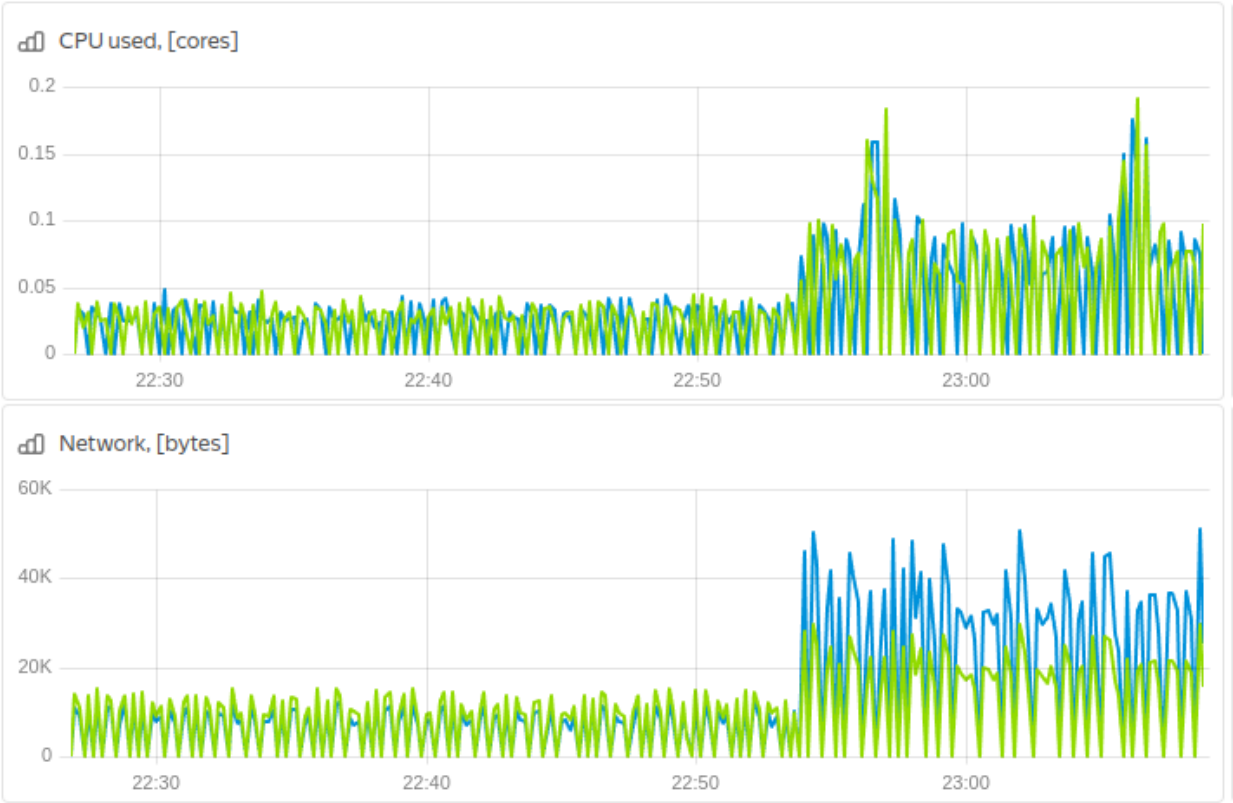

Рис. 10: Хаос-эксперимент, повышение нагрузки на процессор

Приложение также позволяет выполнить серию предварительных операций перед внедрением проблемных сценариев. Хаос-эксперимент позволяет настроить параллельный запуск операций, что позволяет создавать сложные сценарии, комбинирующие одновременное ухудшение производительности по набору характеристик. На рис.11 и рис.12 представлены примеры влияния хаос-экспериментов на показатели сервера в кластере.

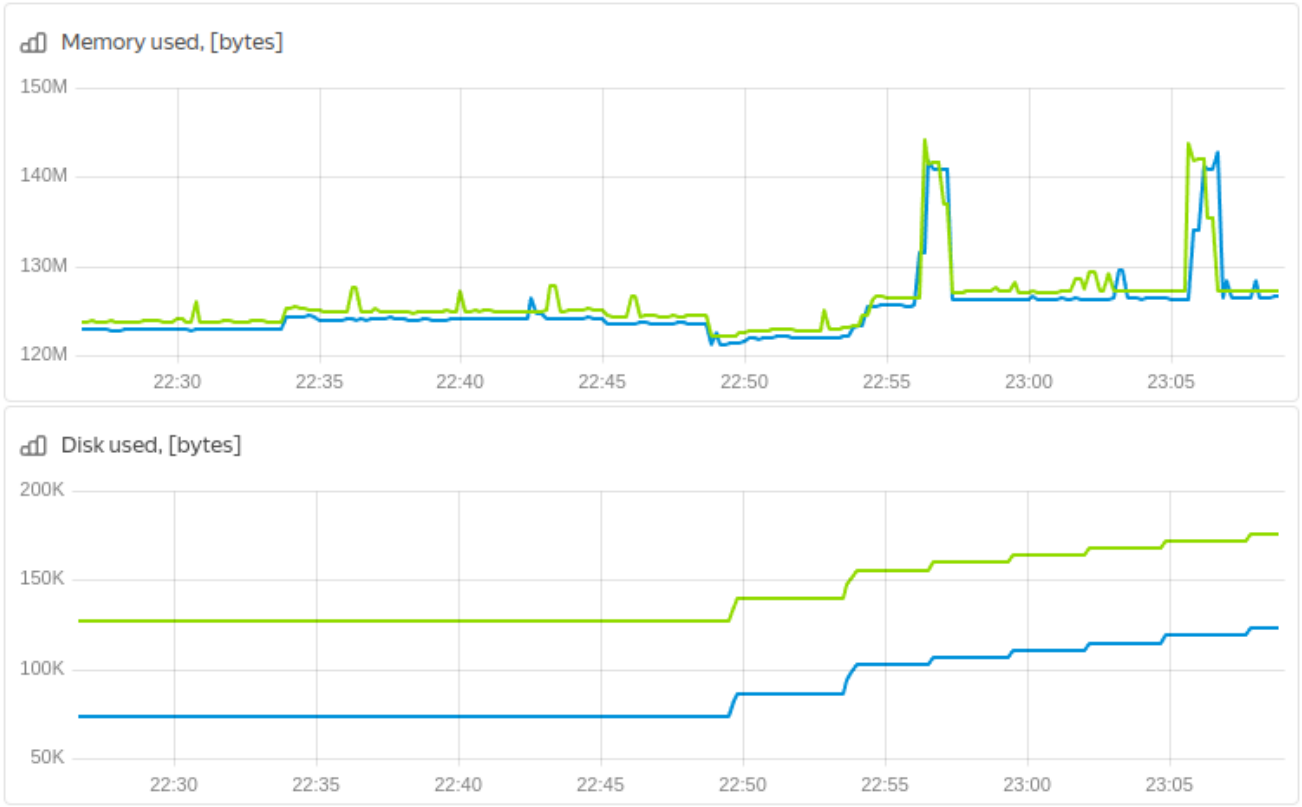

Рис. 11: Хаос-эксперимент, повышение нагрузки на оперативную память

7 Результаты тестирования

При тестировании оценивать производительность кластера будем по времени коммита операции в реплицированный лог. Это время включает в себя запись полученной от клиента операции на диск лидера, отправку операции всем узлам кластера и ожидание подтверждения записи от большинства серверов. На графиках будут представлены три квантиля этой величины - 0.5, 0.9 и 0.99.

Рассмотрим сначала влияние различных хаос-экспериментов, понижающих производительность лидера, на показатели системы. Для это отключим механизм оппозиции, чтобы лидер кластера не менялся.

7.1 Нагрузка на оперативную память

Нагрузим оперативную память лидера - займём 70 из 200 мегабайт, которые Kubernetes выделил серверному агенту лидеру. При этом сам лидер при обычной работе занимает около 100 мегабайт.

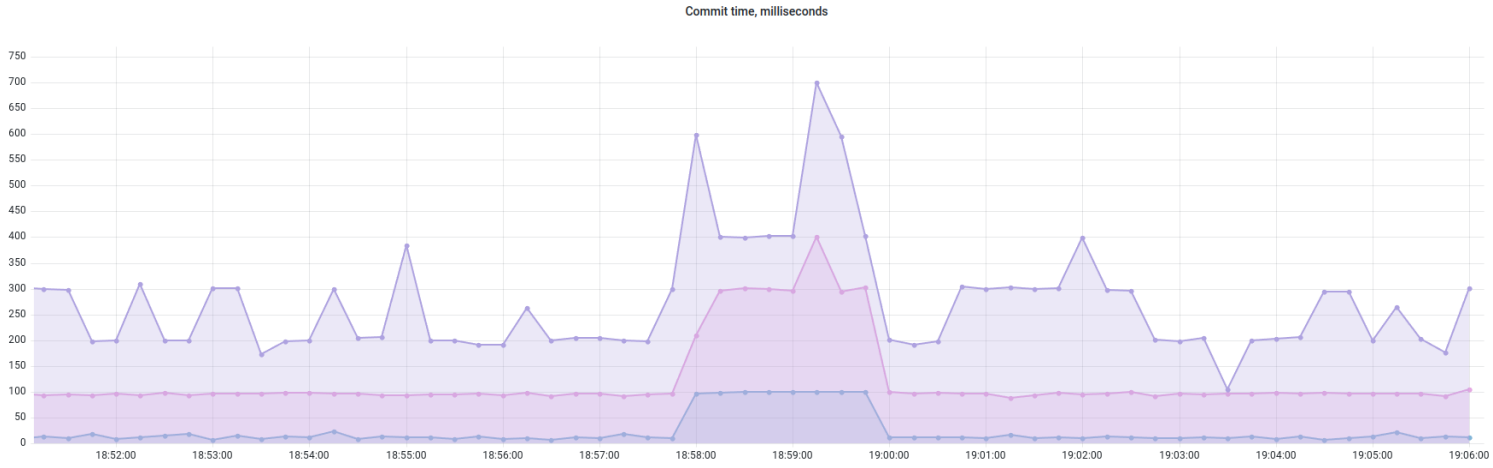

Рис. 12: Commit time, нагрузка на оперативную память

На графике времени commit-а наблюдаем ухудшение показателя. В самых худших случаях (0.99 квантиль) операция стала занимать 600 - 700 миллисекунд. Квантили 0.9 и 0.5 также возросли - до 300 и 100 миллисекунд соответственно.

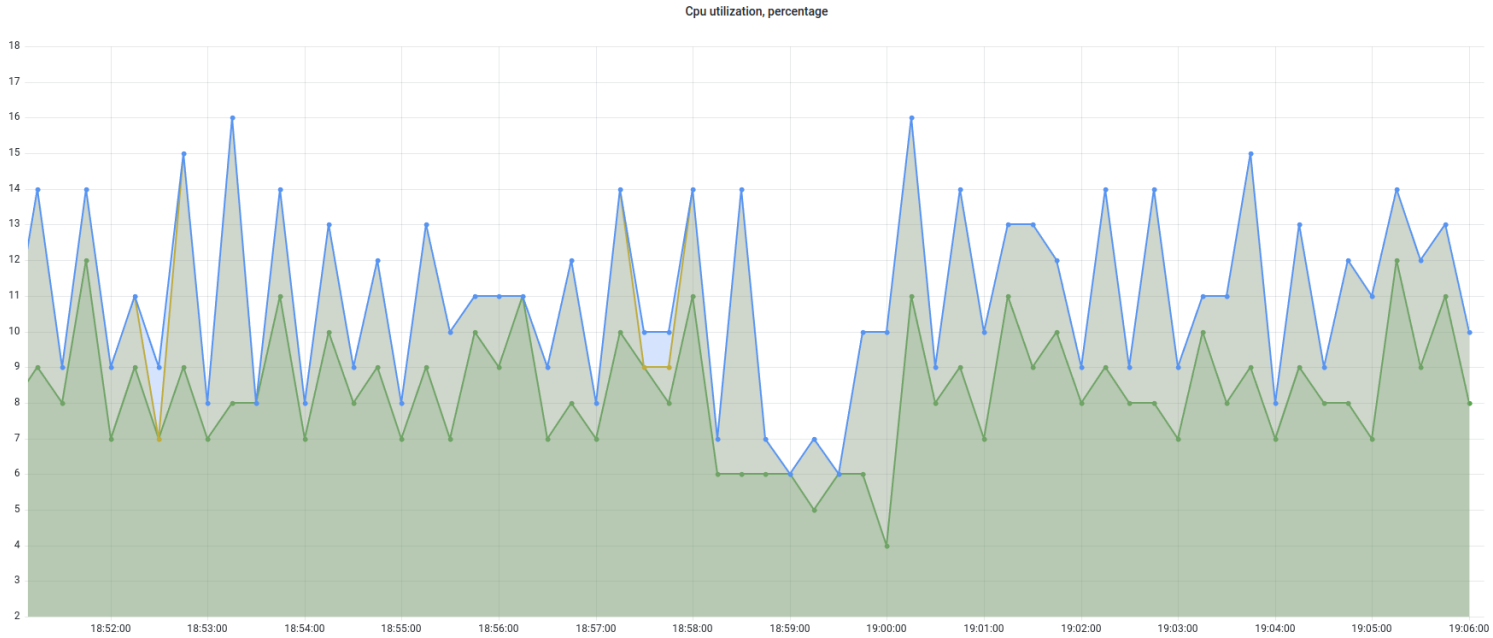

Рис. 13: Cpu utilization, нагрузка на оперативную память

На графике утилизации процессора наблюдаем обратный эффект - все три квантиля утилизации процессора снизились. Возможное объяснение: с повышением нагрузки на оперативную память система замедлилась, вследствие чего понизилась нагрузка на процессор - сервер не успевает поставлять операции на процессор.

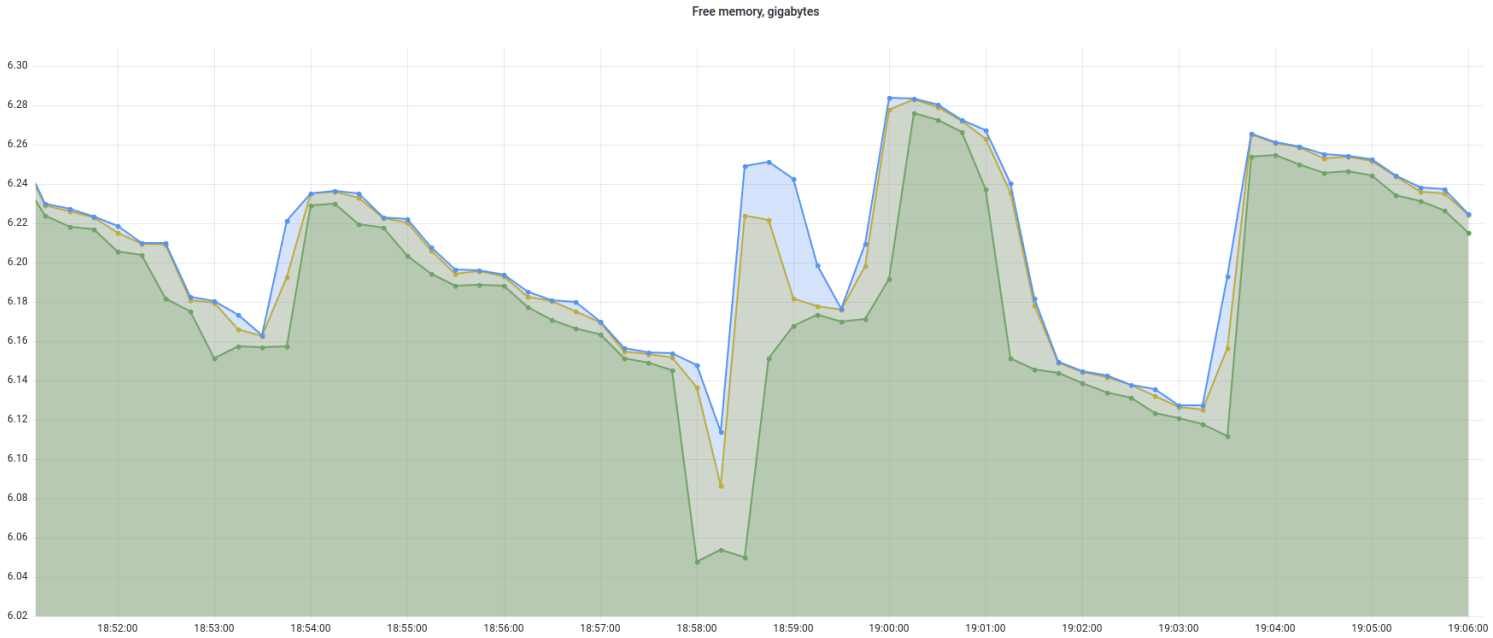

Рис. 14: Free memory, нагрузка на оперативную память

На графике оперативной памяти видим ожидаемое понижение показателя. Помимо этого наблюдаем понижение показателя, которое произошло не в рамках хаос-экспериментов. Первое отличается меньшим промежутком времени и более резким характером. Также видим разрыв между квантилями: сначала резко понижается 0.5 квантиль, затем он выравнивается, но подскакивают значения 0.9 и 0.99 квантиля. Вероятное объяснение второго - периоды сборки мусора (garbage collection) в приложении. Примечательно, что внутри контейнера операционная система сообщает о большем количестве доступной оперативной памяти, чем выделено Kubernetes-ом.

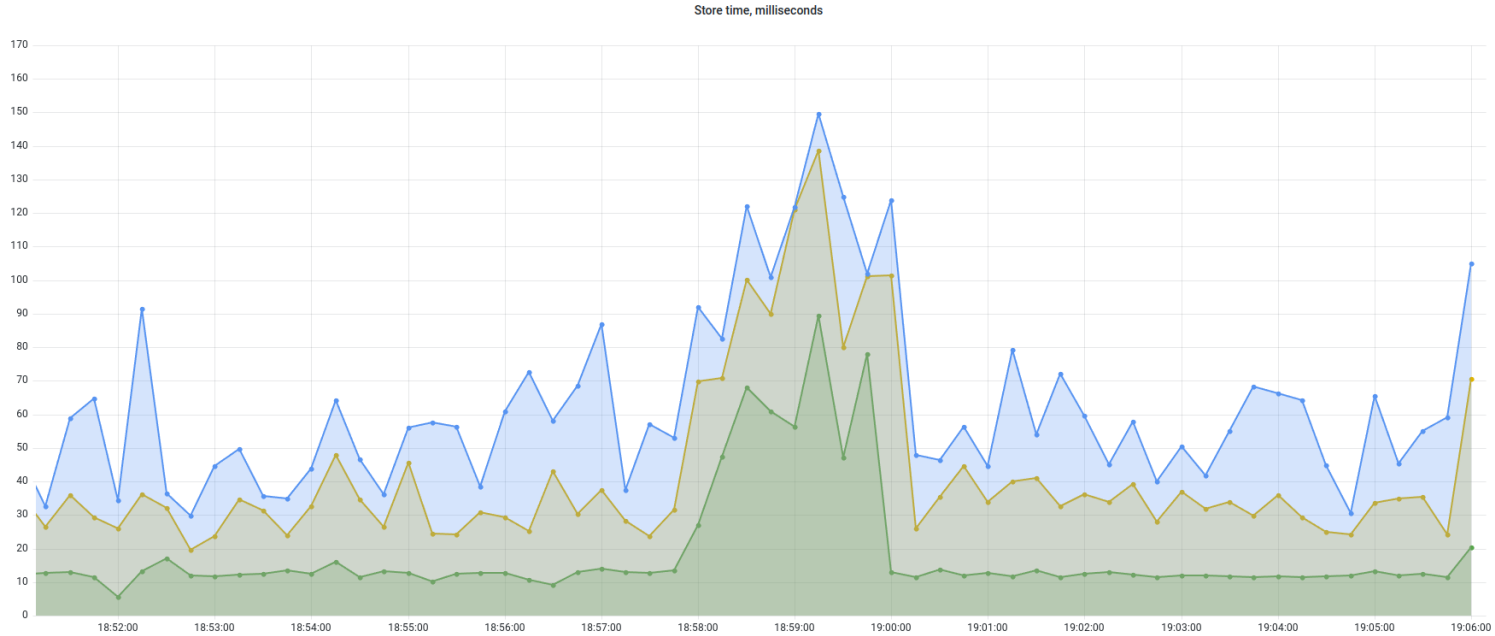

Рис. 15: Store time, нагрузка на оперативную память

Все три квантиля времени записи операции на диск лидера значительно возросли.

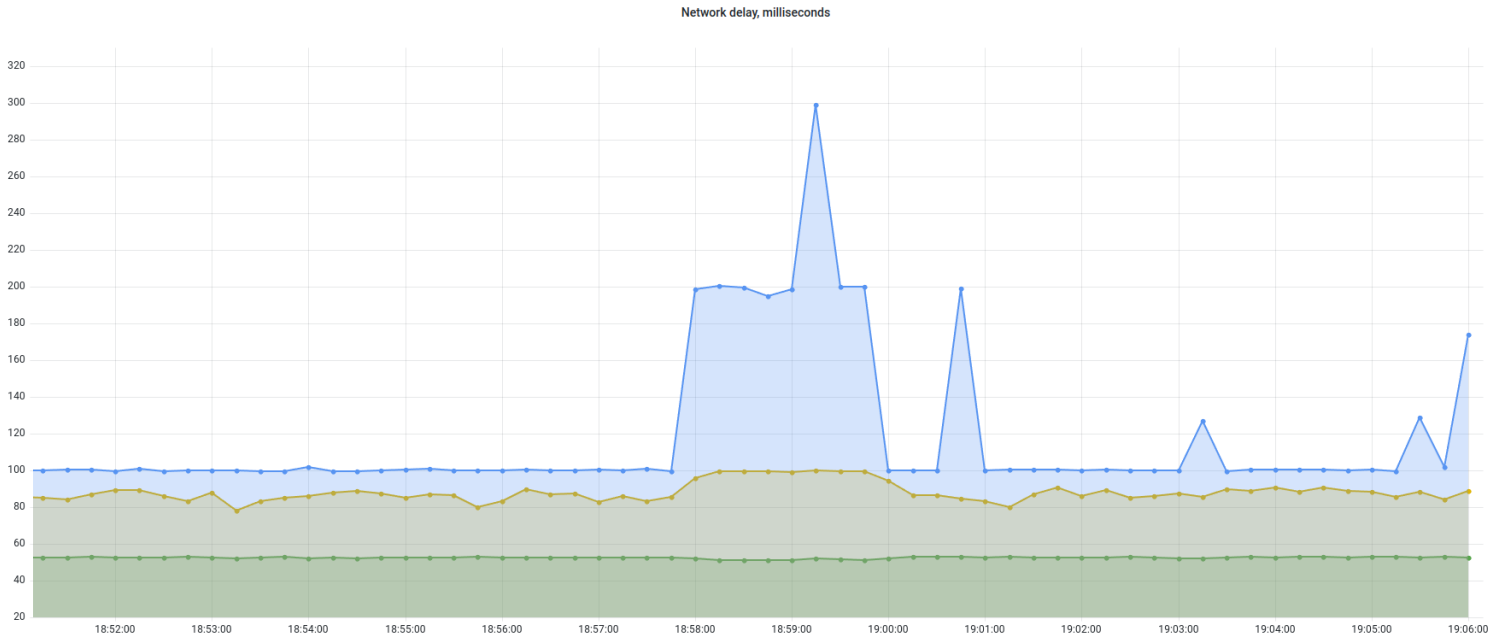

Рис. 16: Network delay, нагрузка на оперативную память

На графике сетевой задержки также наблюдается ухудшение, но только 0.99 квантиля.

Вывод:

В результате эксперимента наблюдаем сильную зависимость между замерами времени commit-а операции и времени записи операции на диск. В меньшей степени тенденции следует метрика сетевой задержки. Замеры доступной оперативной памяти соответствуют эксперименту, но похожие, хоть и менее явные, явления наблюдаются и вне зависимости от хаос-экспериментов. Показатели утилизации процессора наоборот улучшились.

7.2 Нагрузка на процессор

Нагрузим процессор лидера - повысим в двух отдельных экспериментах утилизацию до 90 и 100 процентов.

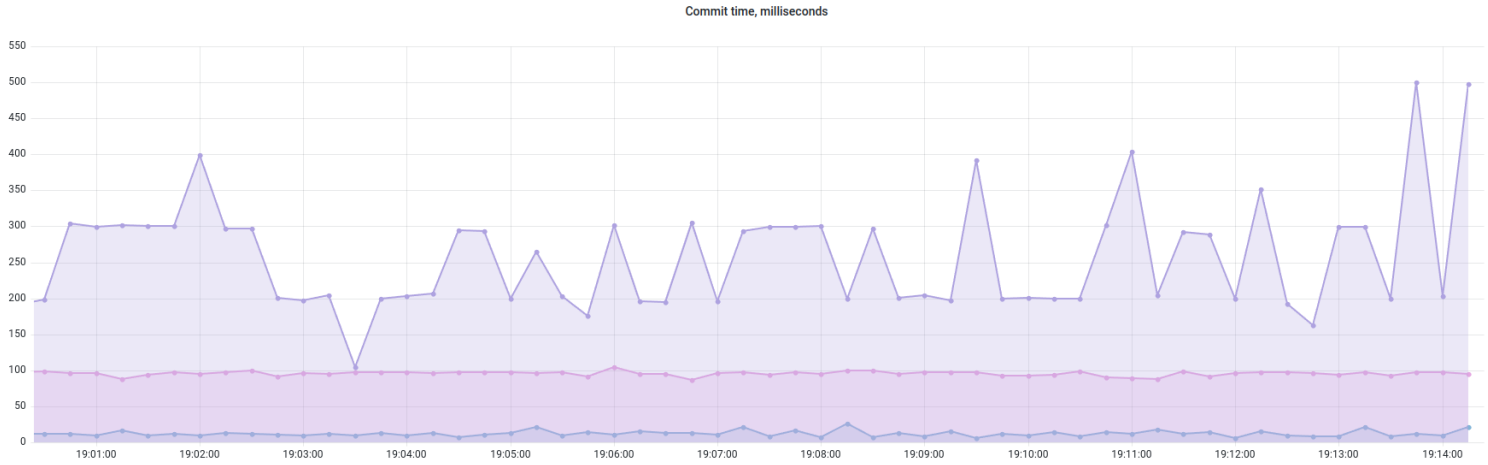

Рис. 17: Commit time, нагрузка на процессор

На графике времени commit-а операции не наблюдаем изменений.

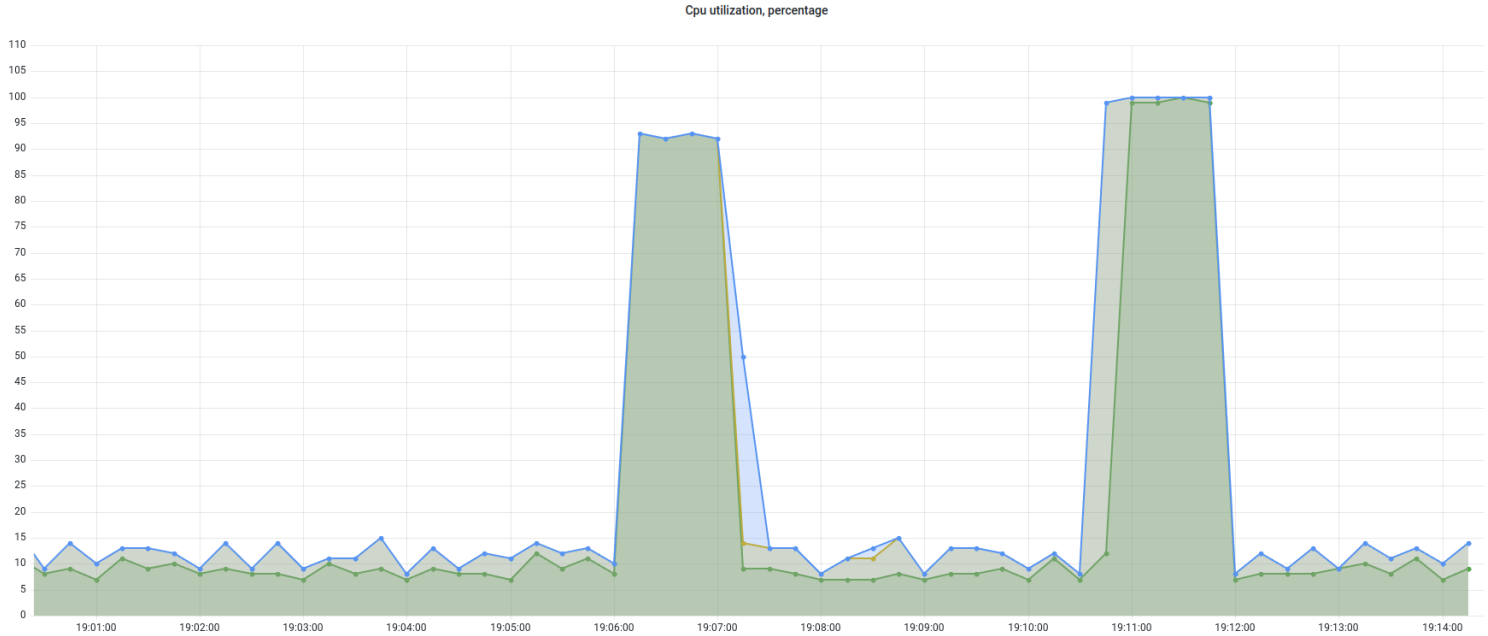

Рис. 18: Cpu utilization, нагрузка на процессор

Замеры утилизации процессора отображают заданные в хаос-эксперименте показатели: наблюдаем два резких возрастания до 90 и 100 процентов.

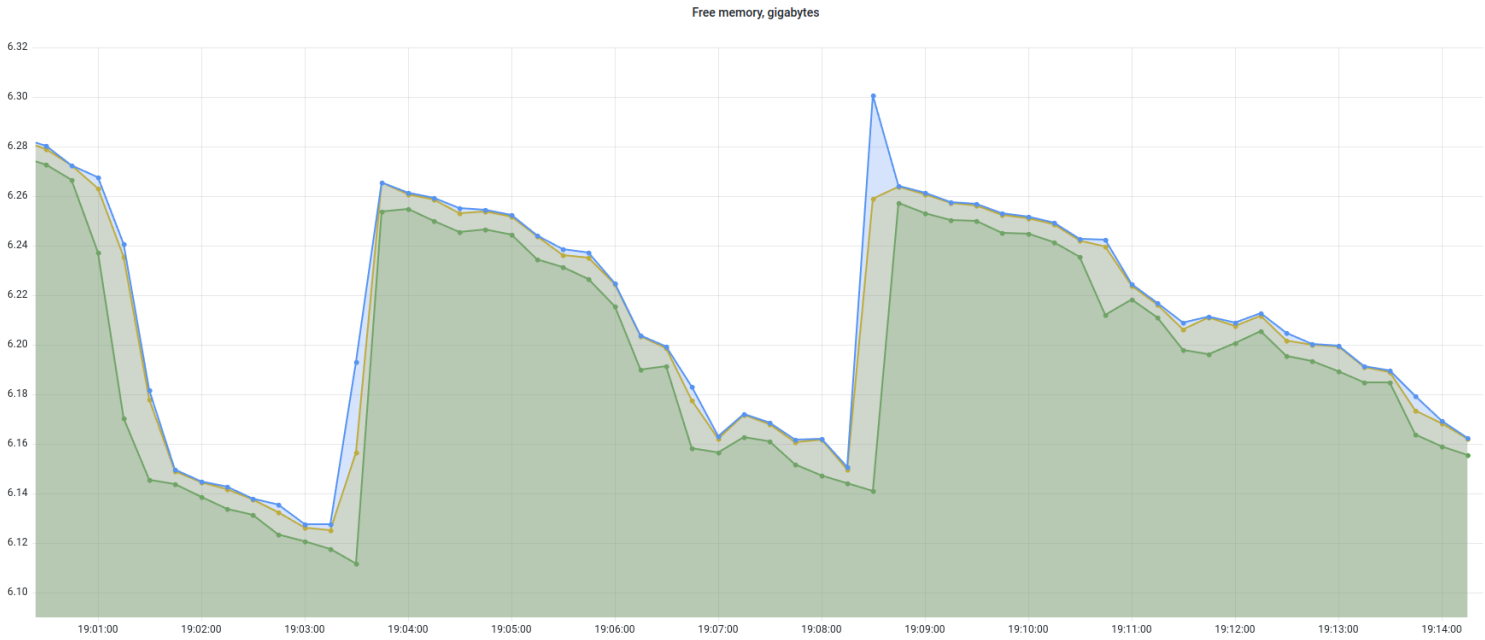

Рис. 19: Free memory, нагрузка на процессор

На графике оперативной памяти наблюдаем еще одно понижение. Оно аналогично понижению, которое наблюдали ранее (оно видно на графике слева) и которое не было связано с экспериментом.

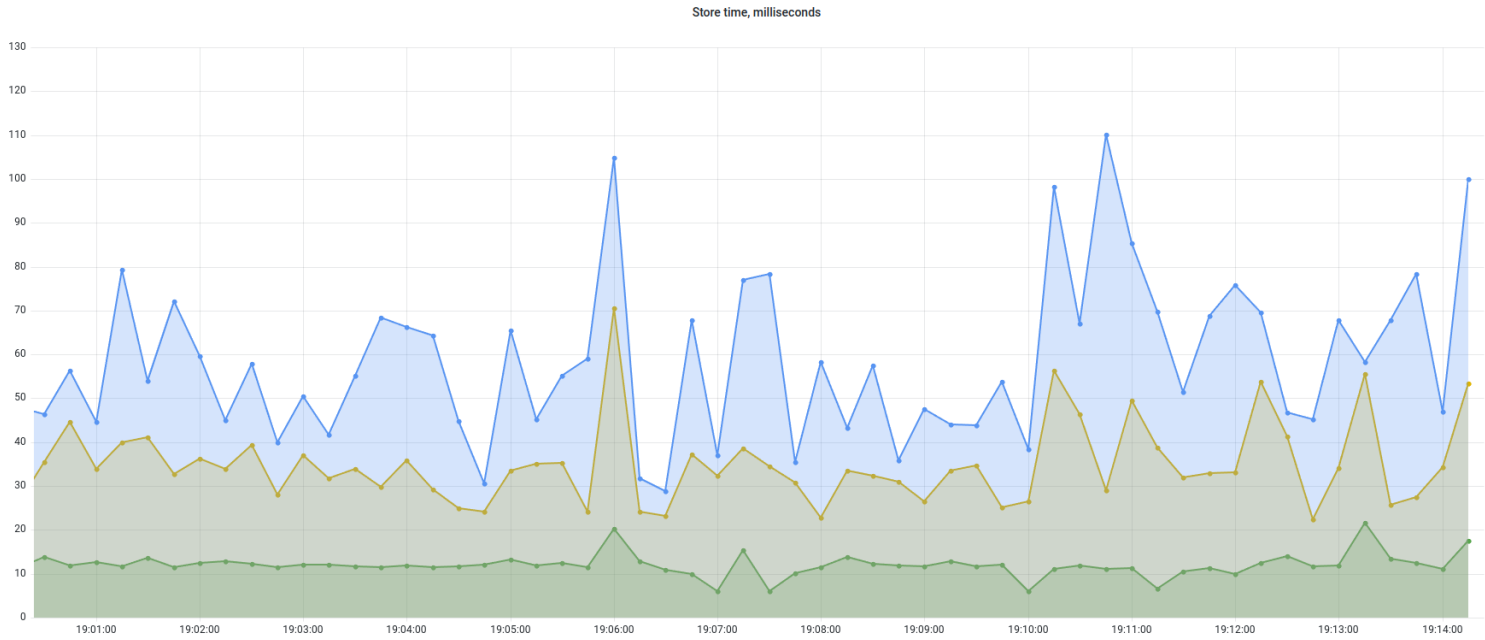

Рис. 20: Store time, нагрузка на процессор

На графике времени записи операции на диск можно выделить два участка, на которых метрика незначительно ухудшается. Эти два участка соответ-

ствуют двум экспериментам.

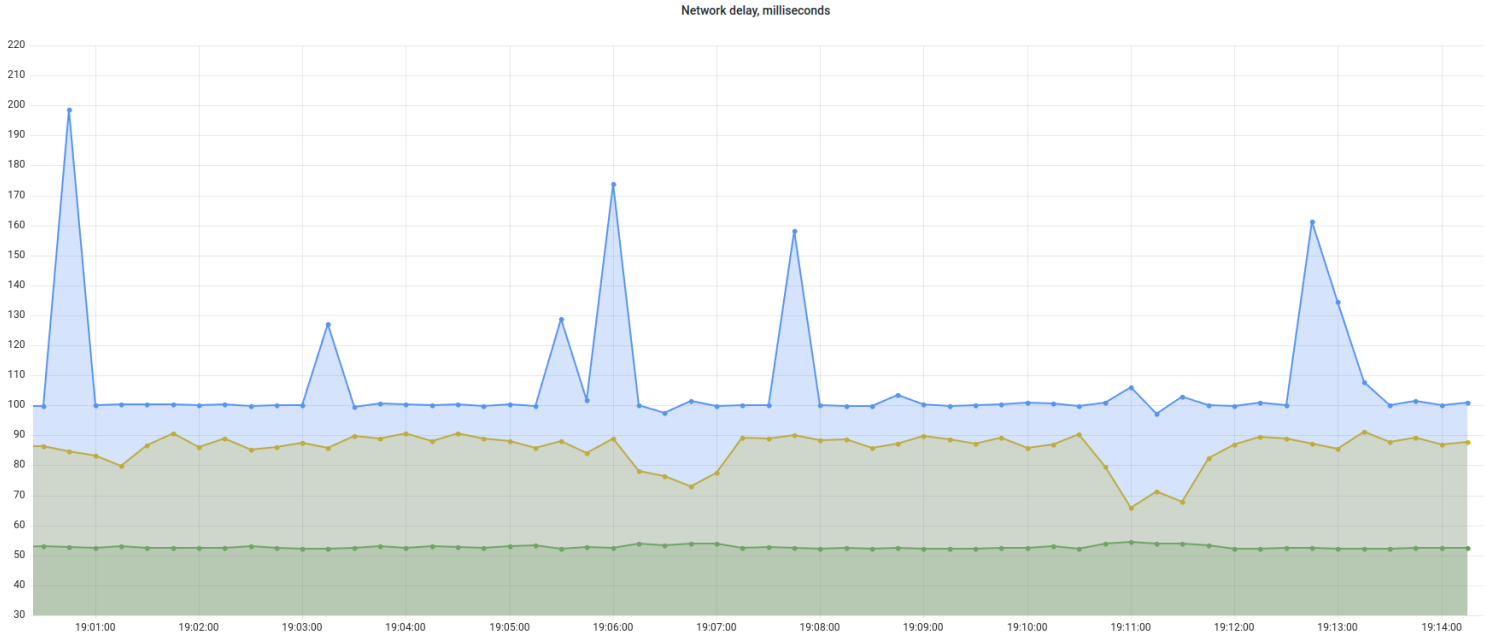

Рис. 21: Network delay, нагрузка на процессор

Сетевая задержка показывает очень незначительную обратную зависимость: 0.9 и 0.5 квантили проседают на временных отрезках, которые соответствуют эксперименту.

Вывод:

Кратковременное повышение утилизации процессора оказывает очень незначительное негативное влияние на время записи операции. Влияние на сетевую задержку оказалось обратным, но также незначительным. Эксперимент не вызвал наблюдаемых изменений в показателях других метрик.

7.3 Нагрузка на сеть

Нагрузим сеть - добавим задержку 200 миллисекунд.

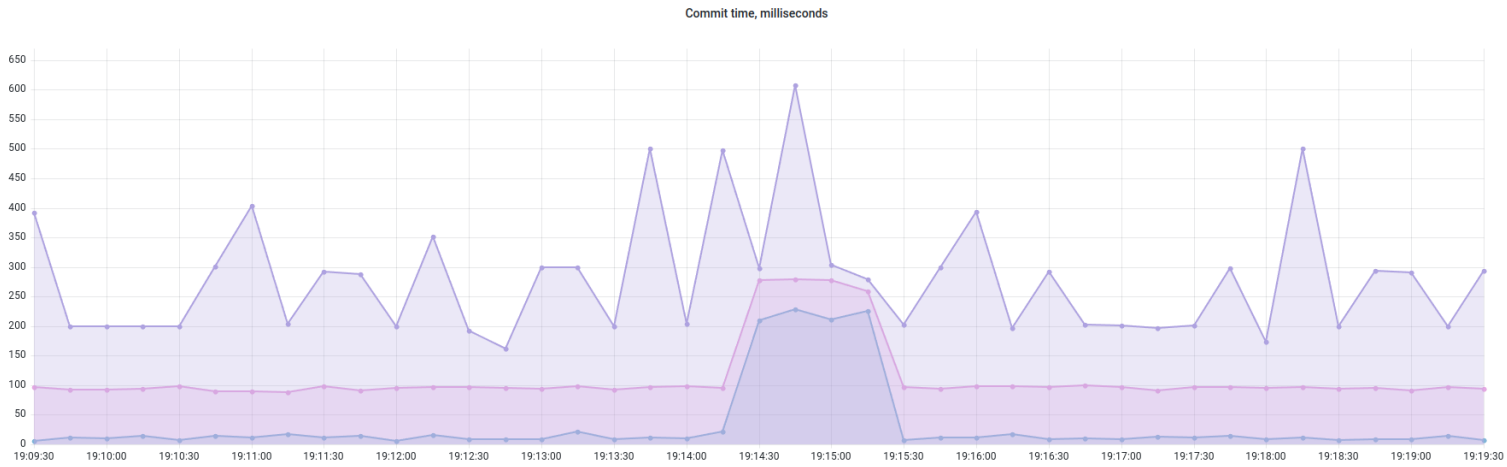

Рис. 22: Commit time, нагрузка на сеть

На графике времени commit-а операции наблюдаем очень явное увеличение всех квантилей на обозначенные 200 миллисекунд. Данный эффект ожидаем, ведь время фиксации операции напрямую зависит от времени получения подтверждения от последователей. Таким образом, с ухудшением сетевых характеристик лидер вынужден дольше ждать.

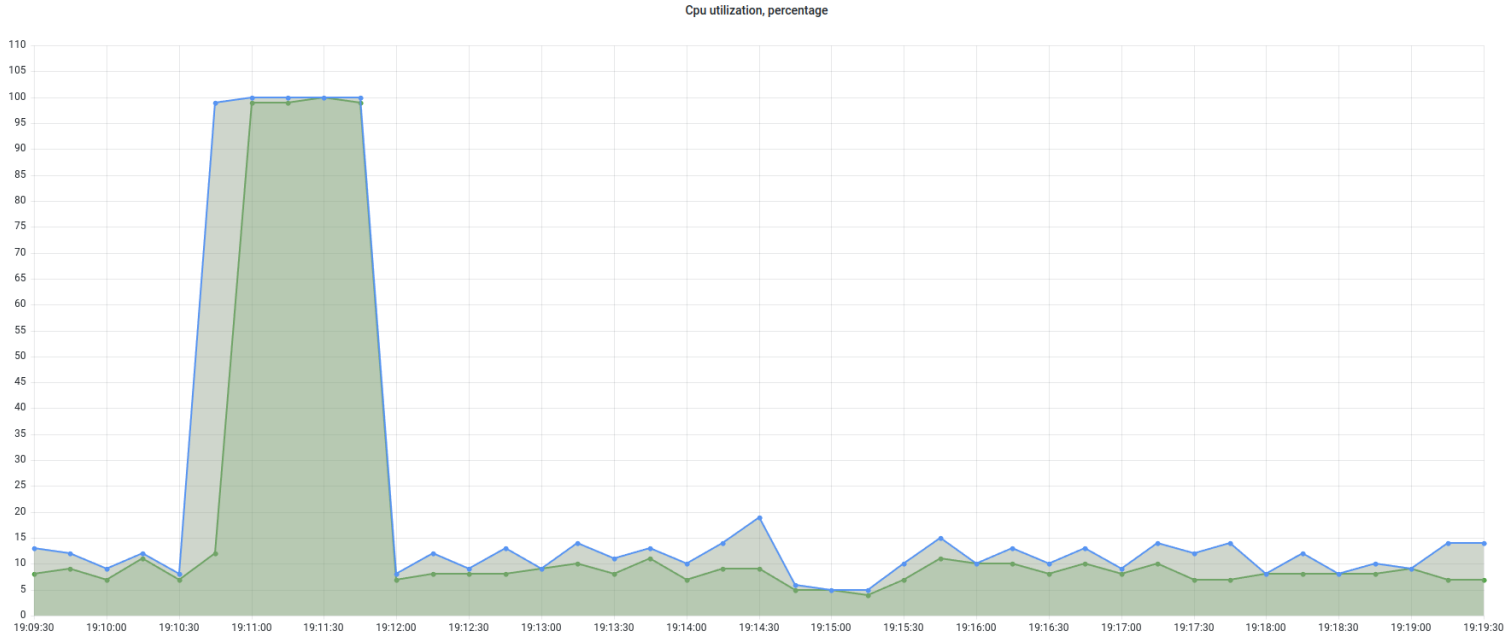

Рис. 23: Cpu utilization, нагрузка на сеть

График утилизации процессора повторяет результаты первого эксперимента. Наблюдаем обратный эффект - все три квантиля утилизации процессора немного снизились. С большей задержкой передачи по сети система

замедлилась, вследствие чего понизилась нагрузка на процессор.

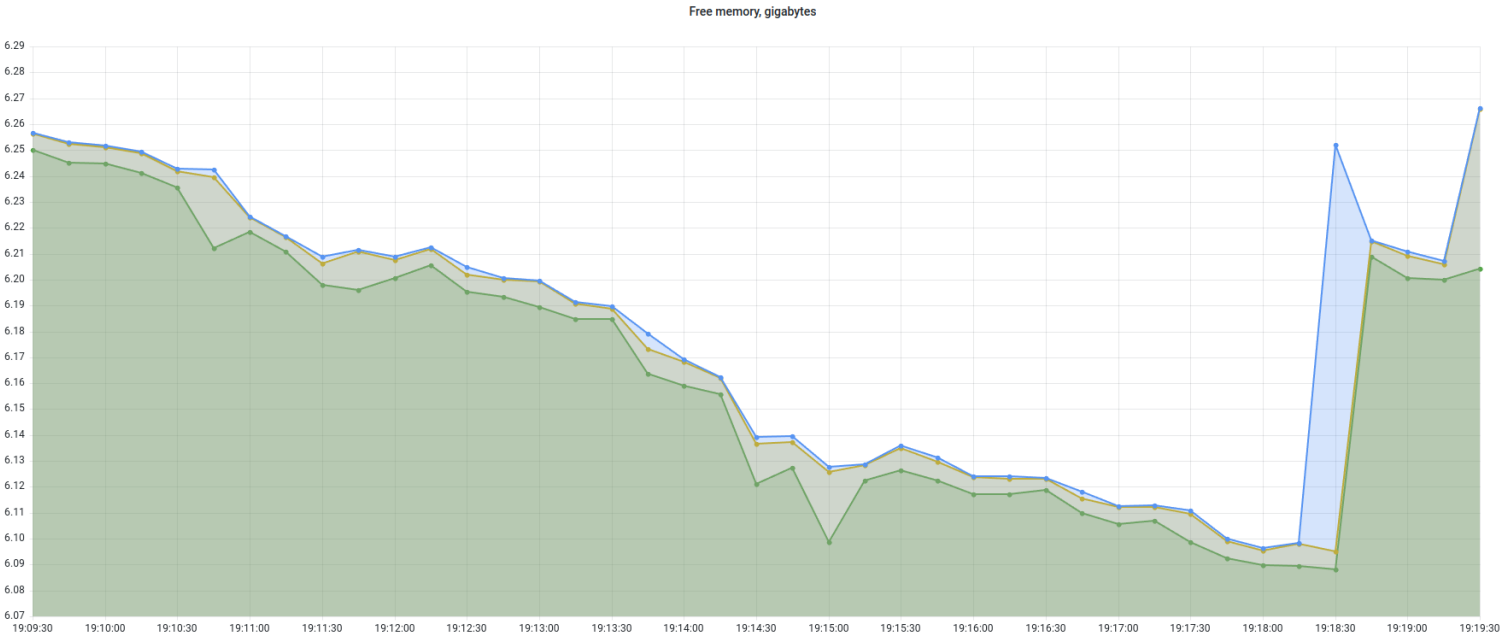

Рис. 24: Free memory, нагрузка на сеть

График доступной оперативной памяти повторяет свое поведение - содержит характерное понижение значения, после которого оно резко возрастает. Данное поведение объясняется процессом сборки мусора в приложении. Однако можем также выделить очень незначительное проседание 0.5 квантиля в момент проведения эксперимента.

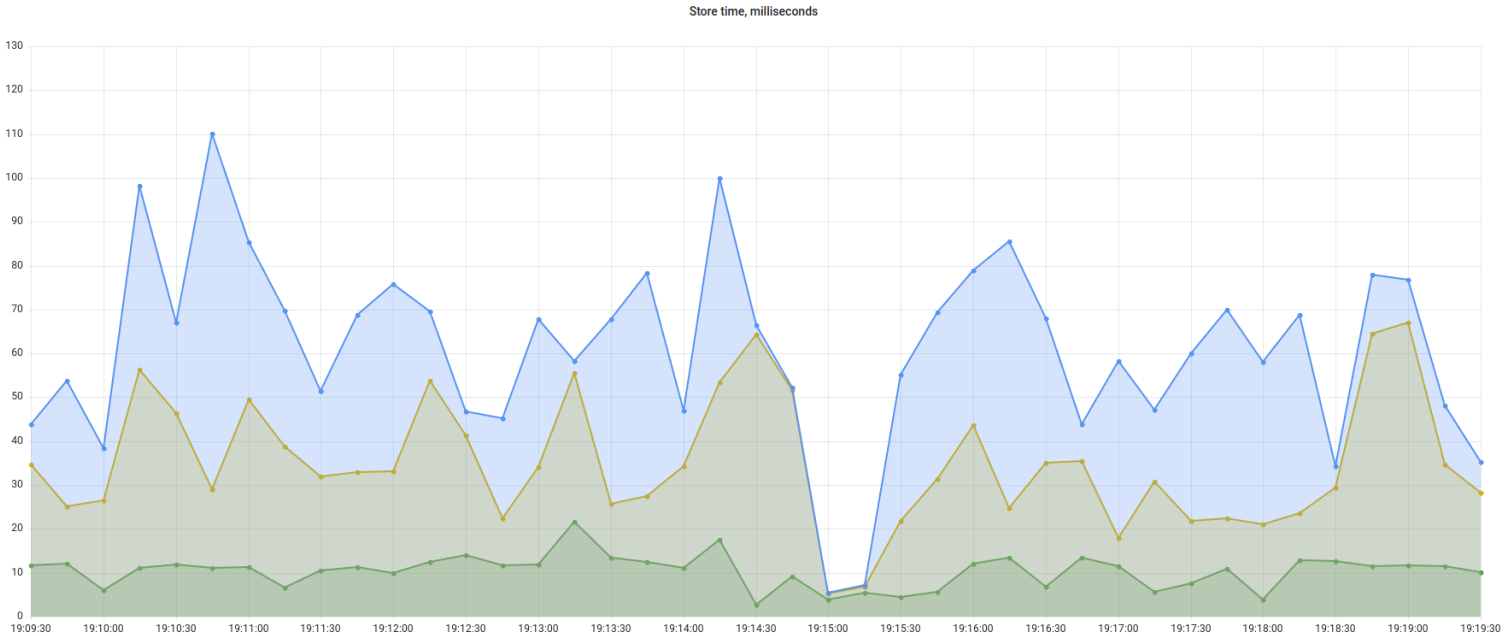

Рис. 25: Store time, нагрузка на сеть

Время записи операции на диск очень заметно уменьшается. Из-за сетевой задержки лидер согласует меньше операций в единицу времени и из-за этого меньше операций в единицу времени поставляется на запись.

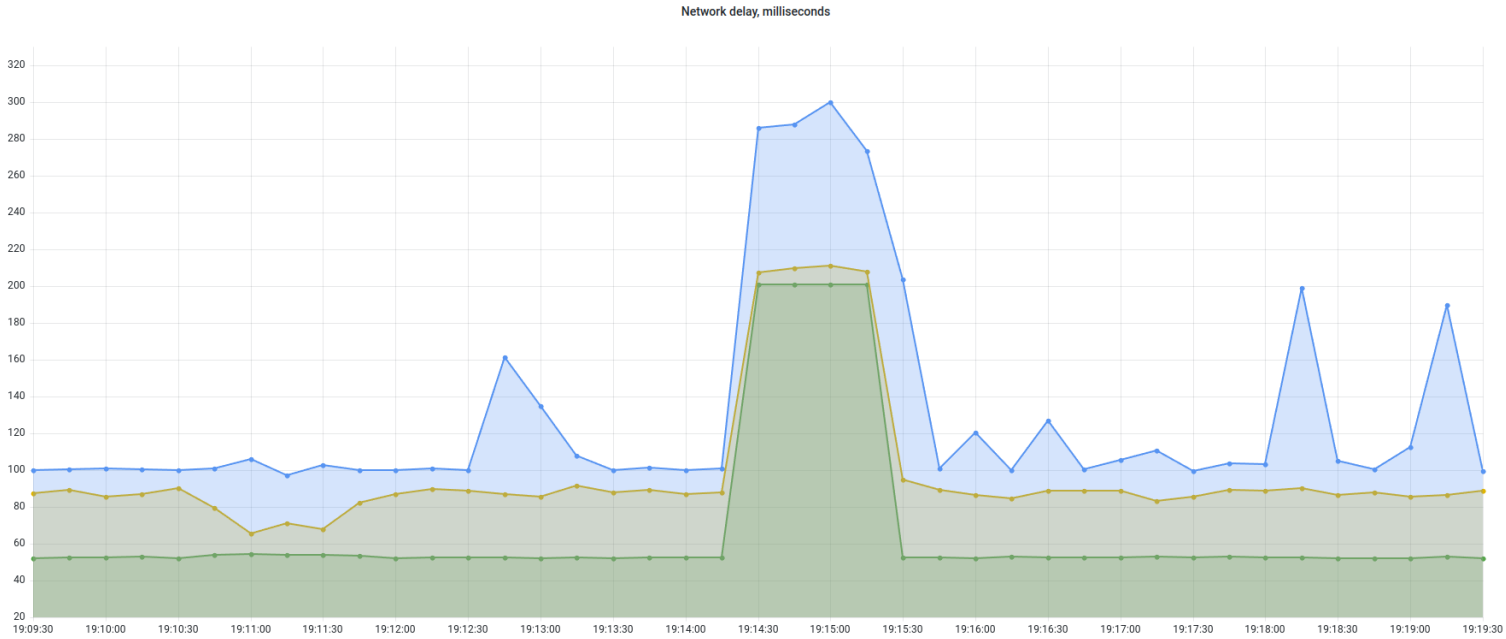

Рис. 26: Network delay, нагрузка на сеть

График сетевой задержки соответствует эксперименту - наблюдаем повышение всех квантилей на 200 миллисекунд.

Вывод:

Сетевая задержка имеет прямой негативный эффект на метрику времени commit-а операции и очень малый эффект на оперативную память. Утилизация процессора снова показывает незначительную обратную зависимость. В отличие от утилизации процессора, время записи на диск показывает очень явную обратную зависимость.

7.4 Нагрузка на встроенную память

Нагрузим встроенную память лидера - запустим эксперимент занимающий 50% пропускной способности диска.

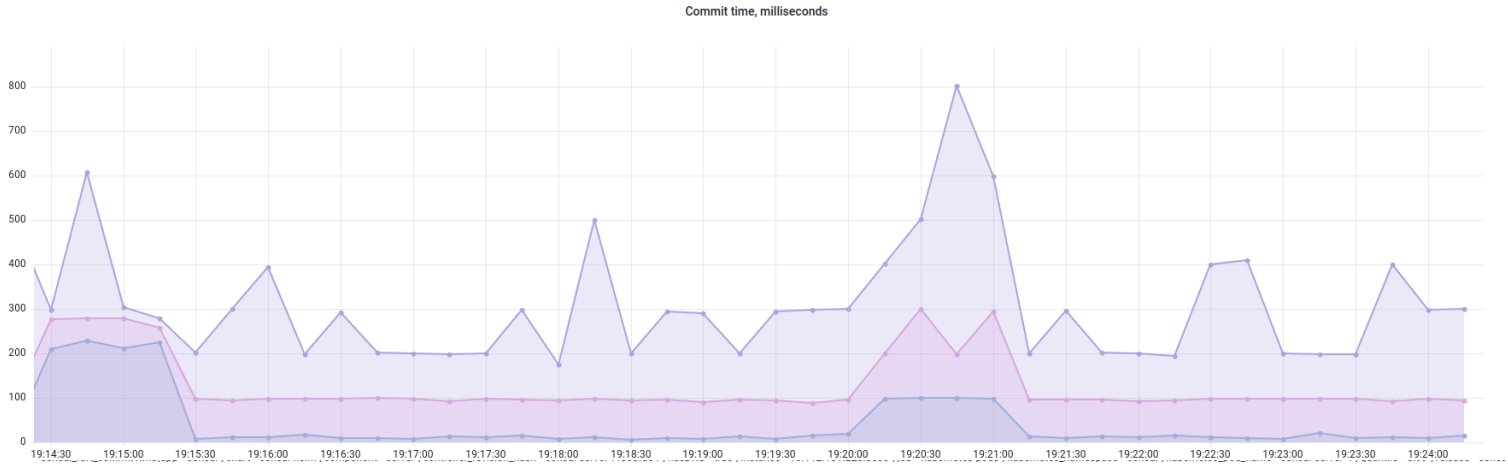

Рис. 27: Commit time, нагрузка на встроенную память

Время commit-а операции возросло, но, в отличие от прошлого эксперимента (который виден на графике слева), квантили увеличились на разное время. Квантиль 0.5 возрос на 100 миллисекунд, квантиль 0.9 на 200 миллисекунд, квантиль 0.99 на 600 миллисекунд. Примечательно, что квантиль 0.99 возрос до 800 миллисекунд - это его самое большое значение из всех экспериментов. В то же время, 0.5 квантиль изменился очень незначительно.

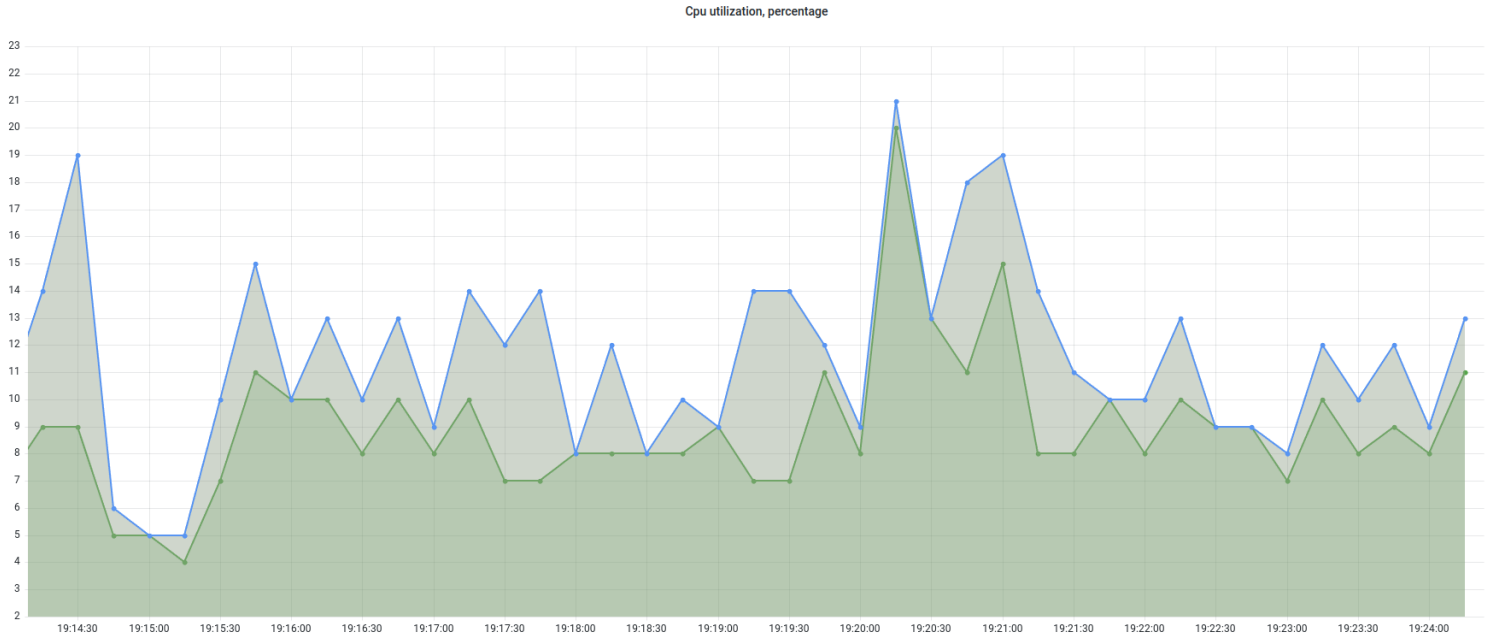

Рис. 28: Cpu utilization, нагрузка на встроенную память

На графике утилизации процессора наблюдаем повышение сразу всех трёх квантилей до уровня 20%. Помимо второго эксперимента, в котором мы

напрямую повышали утилизацию процессора, это самые высокие показатели за все время. Более того, это первый замер, в котором утилизация процессора повышается вместе с другими метриками.

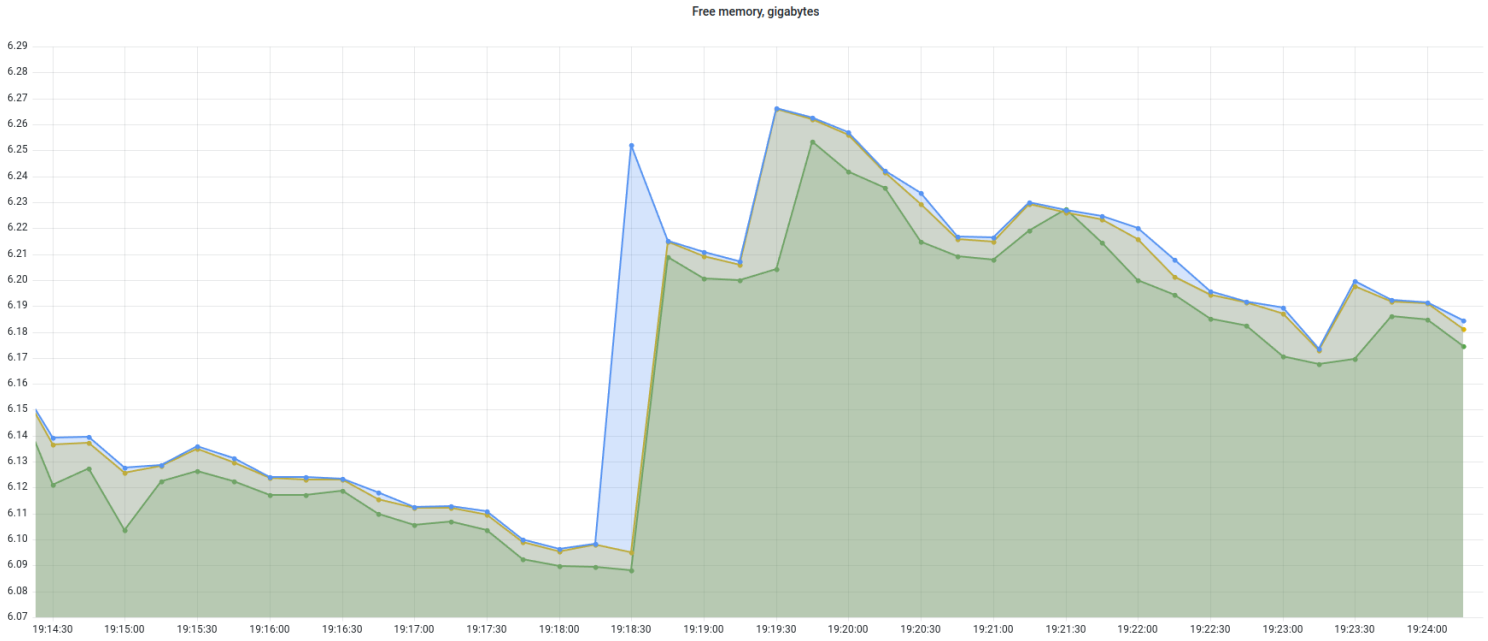

Рис. 29: Free memory, нагрузка на встроенную память

График оперативной памяти не отображает изменений в замерах метрики. Наблюдаем все то же характерное волнообразное поведение.

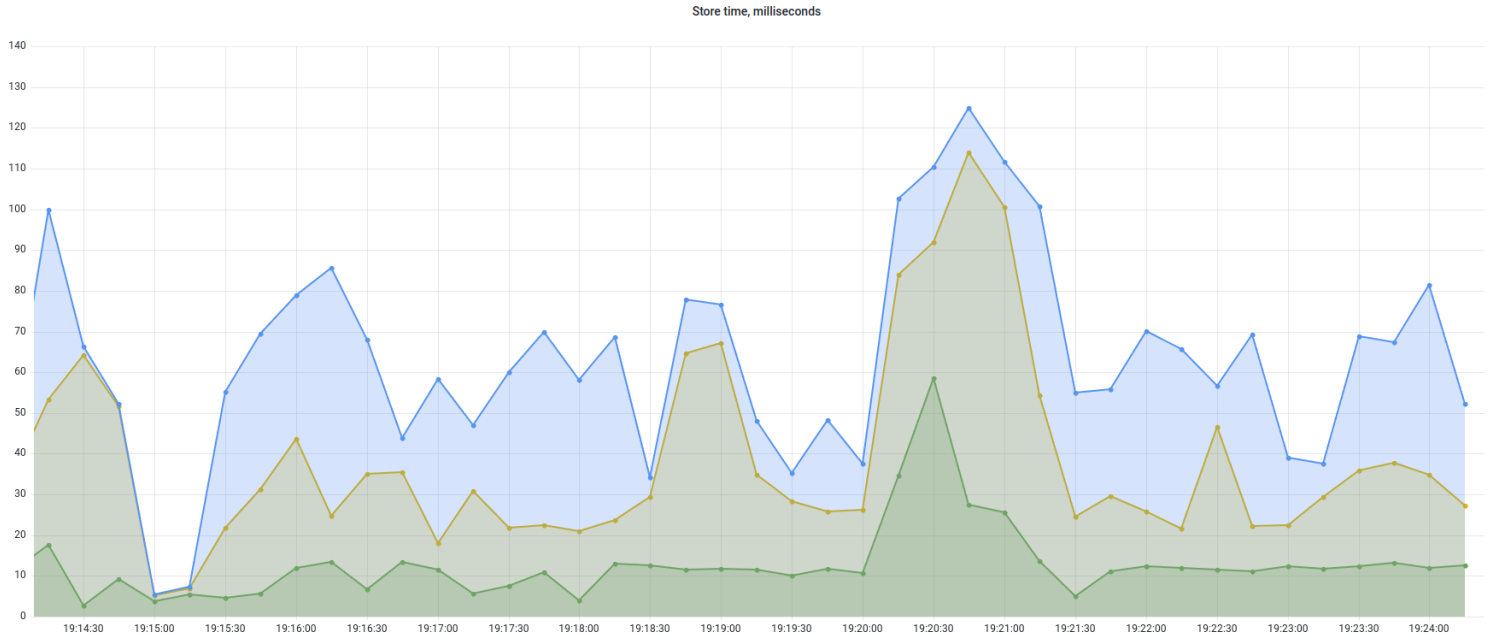

Рис. 30: Store time, нагрузка на встроенную память

Соответственно эксперименту график времени записи операции на диск увеличился, в частности заметно возрос 0.5 квантиль.

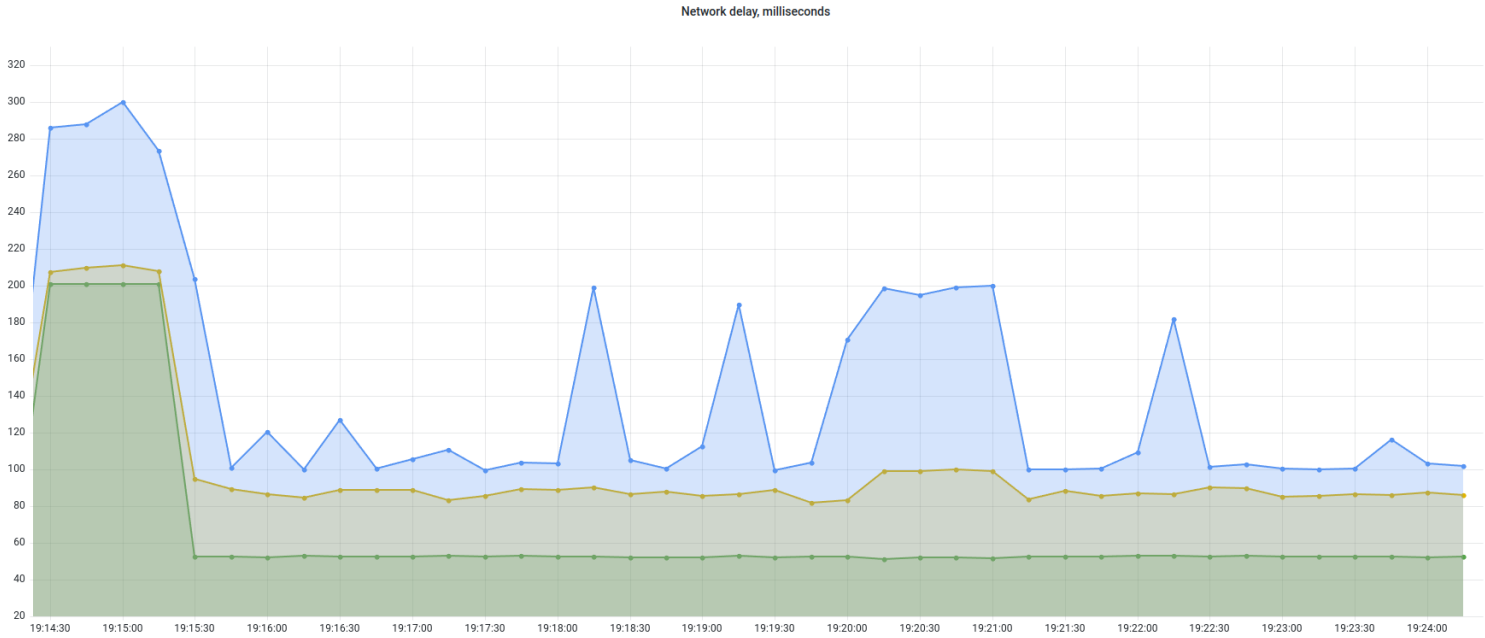

Рис. 31: Network delay, нагрузка на встроенную память

На графике сетевой задержки также наблюдается ухудшение, и, так же как и в первом эксперименте, в основном только 0.99 квантиля.

Вывод:

Снова наблюдаем прямую зависимость между метриками времени commit-а и времени записи на диск операции. В то же время, впервые наблюдаем рост утилизации процессора вместе с повышением других метрик. Сетевая задержка увеличилась менее явно и поведение оперативной памяти не изменилась.

7.5 Выводы по поведению метрик

В ходе трёх из четырёх хаос-экспериментов основная характеристика системы - время commit-а входящих операций - показала явное ухудшение производительности системы. При этом в разных экспериментах этому увеличению сопутствовало увеличение разных метрик. В первом более заметно ухудшилось время записи на диск, менее заметно ухудшились сетевая за-

держка и оперативная память. В третьем заметно ухудшилась только сетевая задержка. В четвертом заметно ухудшились сетевая задержка и время записи на диск, менее заметно ухудшилась утилизация процессора.

Таким образом, все представленные метрики полезны при оценке производительности распределённой системы. Более того, различные негативные сценарии иллюстрируются разным подмножеством метрик. Поэтому в общем случае выгодно конфигурировать узлы кластера на использование разных метрик.

7.6 Consul с оригинальной реализацией протокола Raft

Рассмотрим Consul с оригинальной реализацией протокола Raft. Сначала проведем замер обычной работы системы. На рис.32 представлены графики времени commit-а операции в секундах. Более конкретно, на нём изображены графики квантилей 0.5, 0.9 и 0.99. Из графиков следует, что время commit-а операции в 90% не превышает 0.1 секунду и в худших случаях (0.99 квантиль) занимает 0.2 — 0.4 секунды.

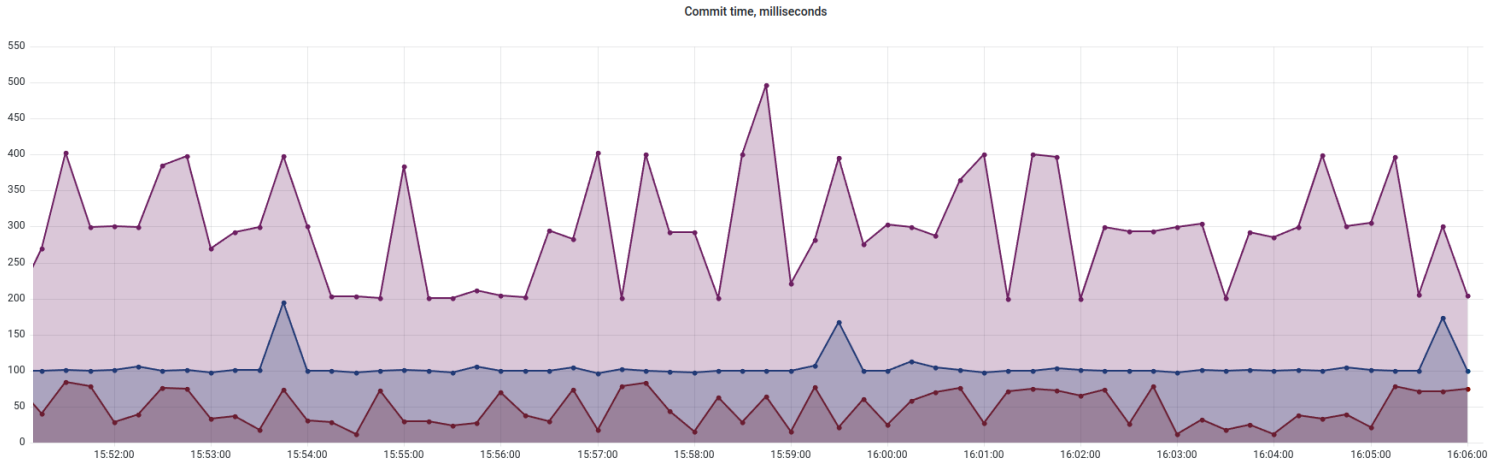

Рис. 32: Commit Time, Raft

Рассмотрим теперь Consul с оригинальной реализацией протокола Raft с внедрением хаос-экспериментов. Повысим нагрузку на оперативную память узла - займем 70 мегабайт из доступных 200. На рис.33 представлены

результаты - также графики квантилей 0.5, 0.9 и 0.99 времени commit-а операции. На графике видим ожидаемое ухудшение показателей: 0.9 квантиль возрос до 300 миллисекунд, а 0.99 квантиль два раза достиг уровня в 600 — 700 миллисекунд. Таким образом показатели кластера по этой характеристике ухудшились в примерно в 3 раза.

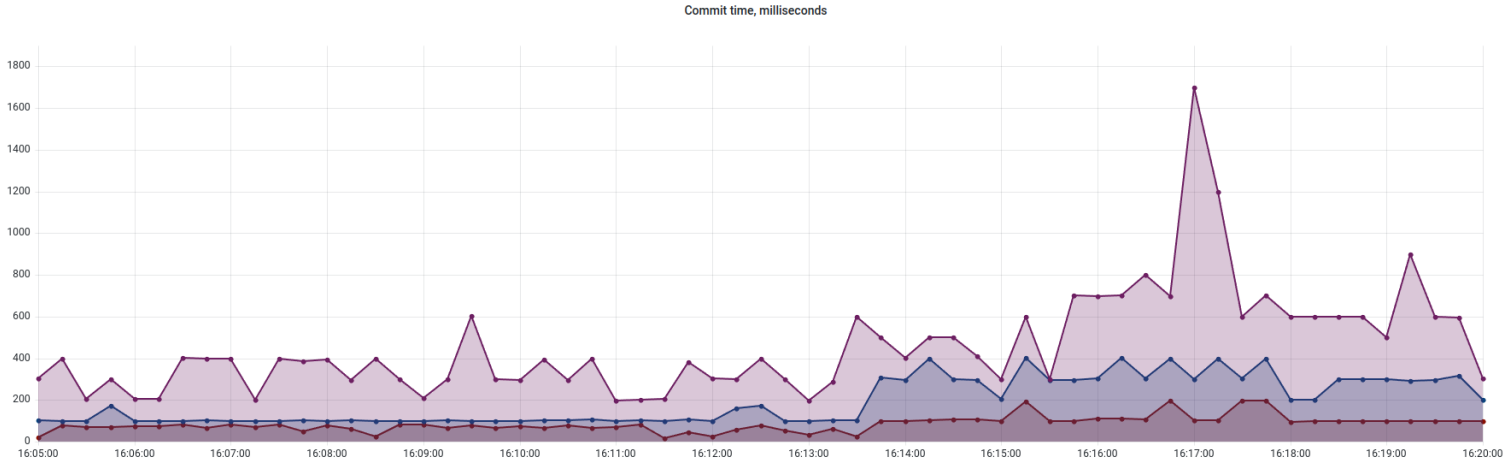

Рис. 33: Commit Time, Raft

Проведем еще один похожий замер с ухудшением сетевого соединения - добавим задержку в 300 миллисекунд. В результате на рис.35 наблюдаем соответственное увеличение всех квантилей характерстики. Эти два эксперимента подтверждают описанные ранее недостатки протокола Raft. Время выполнения ключевой операции, commit-а записи в реплицированный лог, сильно увеличилось, в данном тесте в 3 — 4 раза, но лидер кластера не отказал и кластер продолжил работать с пониженной производительностью.

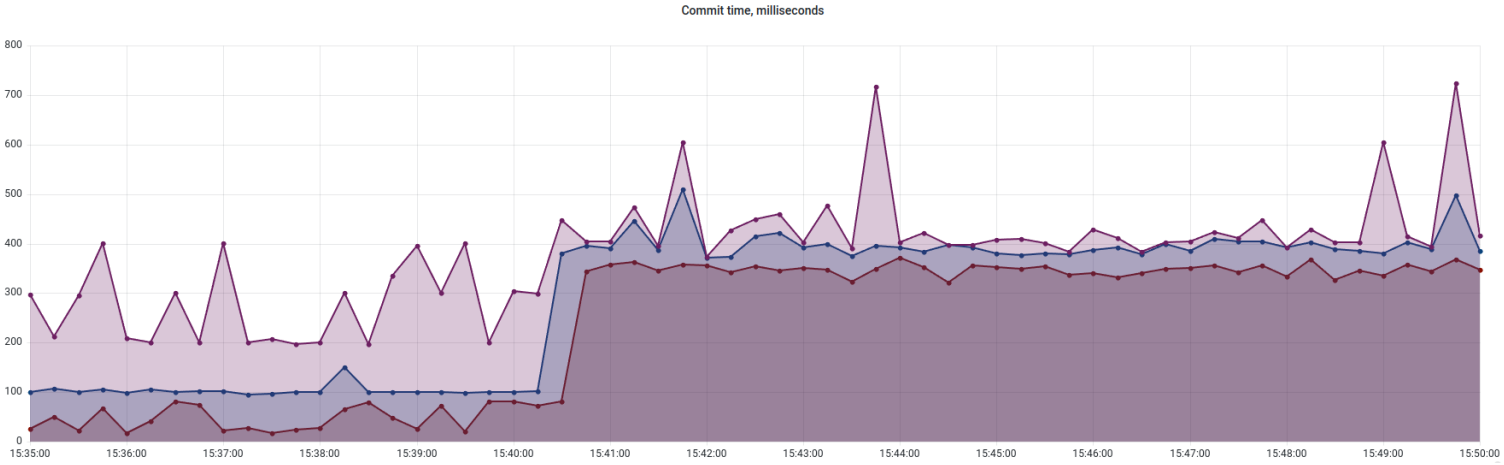

Рис. 34: Commit Time, Raft

7.7 Consul с реализацией протокола Raft-PLUS

В соответствии с результатами тестирования поведения метрик настроим кластер следующим образом. Один из девяти узлов будет использовать утилизацию процессора, два оперативную память. Метрики сетевой задержки и времени записи операции на диск будут использовать по три сервера.

Проведённые раньше эксперименты показали, что на время commit-а операции влияют хаос-эксперименты нагружающие оперативную память, сеть и диск. Рассмотрим сценарий, в котором лидер начинает испытывать нагрузку на оперативную память и на процессор. На рис.35 представлен результат.

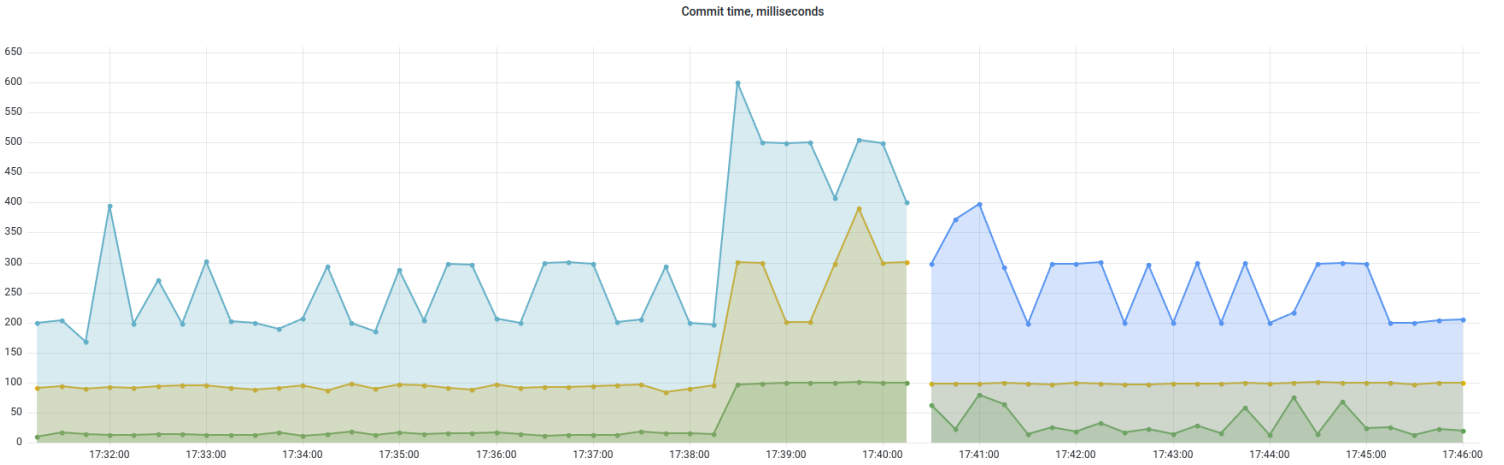

Рис. 35: Commit Time, Raft-PLUS

Рассмотрим ещё один сценарий, в котором лидер испытывает проблемы с оперативной памятью, а также имеет загруженный диск. Результат представлен на рис.36. В обоих экспериментах (рис.35 и рис.36) производительность системы ухудшилась из-за возникших у лидера проблем, но кластер смог оперативно сменить лидера.

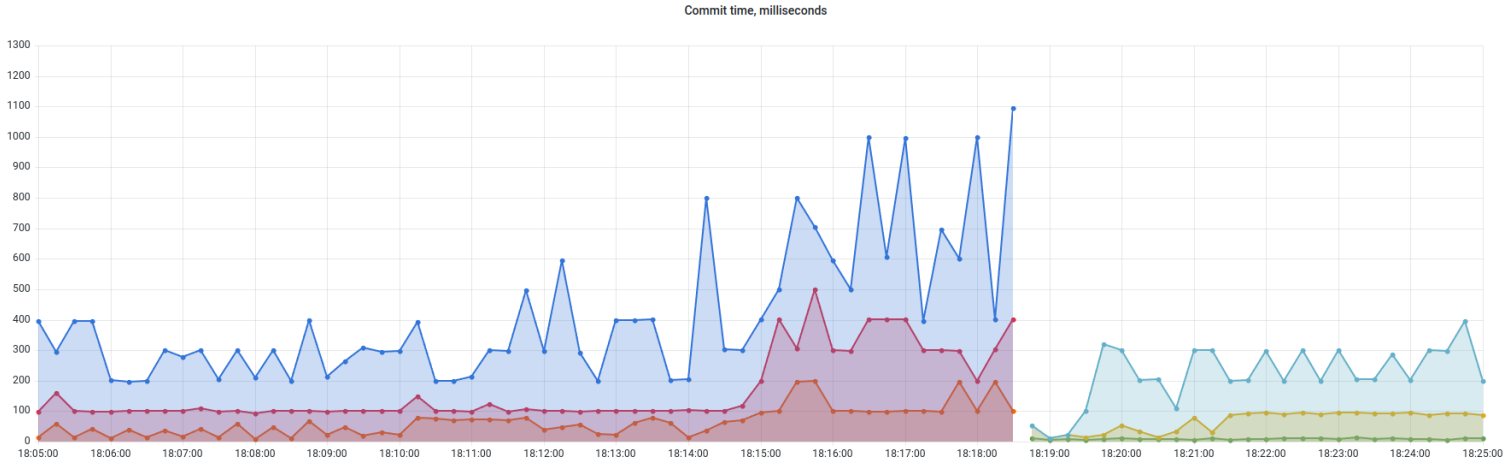

Рис. 36: Commit Time, Raft-PLUS

Проведём ещё один эксперимент, в котором характер возникающих неполадок будет более резким. В новом сценарии лидер будет испытывать сетевую задержку в 400 миллисекунд, а также повышенную до 80% утилизацию процессора. Результат на рис.37 - произошла быстрая смена лидера.

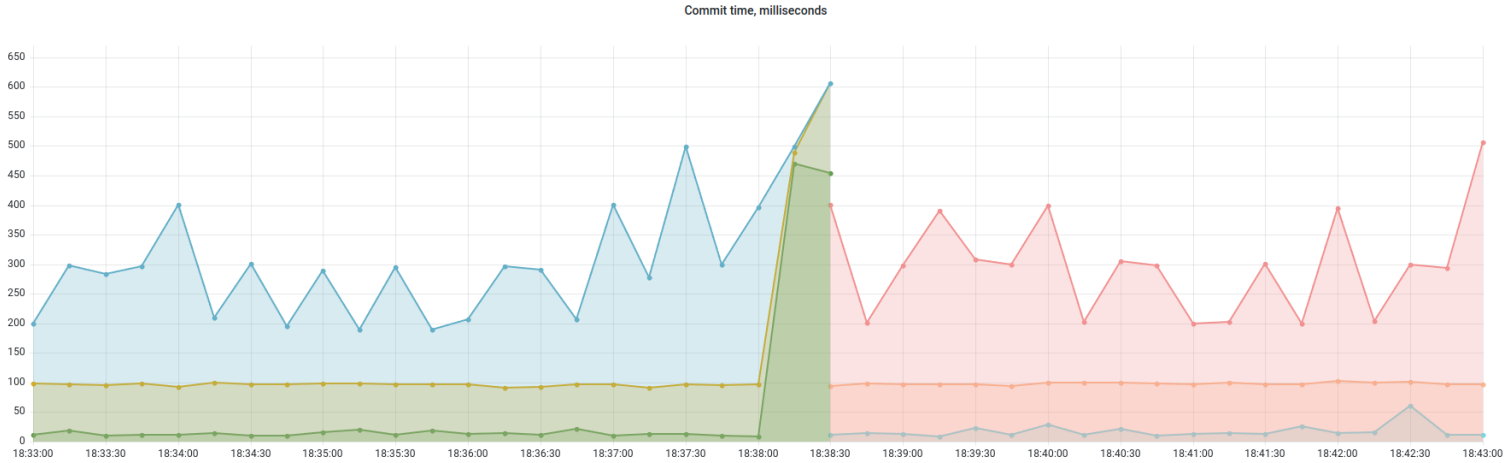

Рис. 37: Commit Time, Raft-PLUS

7.8 Выводы по работе Raft-PLUS

При сравнении графиков работы Consul-a с оригинальным Raft-ом и с подготовленной в рамках данной работы реализацией Raft-PLUS не наблюдаем замедления работы модифицированной версии. В то же время в результате экспериментов было показано, что производительность стандартного протокола может быть понижена в 3 — 4 раза на длительный промежуток времени. Модифицированная версия позволяет избежать данный эффект за счёт оперативной смены лидера. Выводы по использованию конкретных метрик в Raft-PLUS представлены выше.

Список литературы

1. *Ongaro D.*, *Ousterhout J.* In Search of an Understandable Consensus Algorithm. — 2014. — URL: https://raft.github.io/raft.pdf.
2. Raft-PLUS: Improving Raft by Multi-Policy Based Leader Election with Unprejudiced Sorting / J. Xu [и др.] // Multidisciplinary Digital Publishing Institute. — 2022. — URL: https://doi.org/10.3390/sym14061122.
3. Identity-based networking with Consul. — https://www.consul.io/.
4. Test repository. — https://github.com/TheNeonLightning/raft.
5. Open SourceChaos Engineeringplatform. — https://litmuschaos.io/.