Содержание

1 Введение **2**

2 Обзор литературы **3**
2.1 Формальные спецификации 3
2.2 Внесение неисправностей 4

3 Постановка задачи **5**
3.1 Кипарис 5
3.2 Динамические таблицы 9

4 Предложенный алгоритм **12**
4.1 jepsen 12
4.2 Верификация истории 13
4.2.1 Кипарис 13
4.2.2 Динамические таблицы 15

5 Проведённые эксперименты **20**
5.1 Кипарис 20
5.2 Динамические таблицы 21

Часть 1

Введение

Тестирование распределённых и устойчивых к сбоям баз данных является более сложной задачей чем тестирование монолитных систем

В силу асинхронности сети([4]) и возможности аппаратных сбоев разработка распределённых и устойчивых к сбоям баз данных является более сложной задачей чем разработка монолитных систем. Поэтому, требуется более тщательное тестирование, проверяющее отказоустойчивость и выполнение декларируемых гарантий в случае сбоев отдельных узлов и каналов связи между ними. В данной работе используется метод внесения неисправностей(fault injection) для верификации подсистем Yandex.YT.

Yandex.YT является map-reduce системой, которую так или иначе используют практически все сервисы Yandex. В том числе, многие сервисы реального времени хранят данные в подсистеме YT, которую мы тестируем в данной работе. В нем хранятся такие данные как показы рекламы, данные клиентов,.... Поэтому, нарушение заявляемых гарантий может привести к поломке или простою критически важных процессов. И последующей потере денег

Часть 2

Обзор литературы

В данной главе сделан обзор существующих методов верификации гарантий консистентности распределённых систем.

2.1 Формальные спецификации

Метод формальных спецификаций заключается в построении математической модели системы, формализации требований к ней и последующему доказательству того, что система удовлетворяет поставленным требованиям. Обычной практикой является использование инструментов, позволяющих получить доказательство в полуавтоматическом режиме([5]). Построение формальных спецификаций полезно при доказательстве корректности модели соответствующей системы, но в случае, если модель оказывается достаточно громоздкой, возникают вопросы соответствия этой модели реальной системе, а также проблемы обновления модели при изменении её функциональности. Данные ограничения, как отмечают авторы в [9] сильно повышают стоимость поддержки и делают данный подход оправданным только для критичных систем. Впрочем, достаточно большие компании, такие как Amazon, всё-таки могут позволить себе использование формальных моделей ([14]) наряду с другими методами.

2.2 Внесение неисправностей

Метод внесения неисправностей заключается в искусственном создании неисправностей(аппаратных сбоев), направленном на тестирование отказоустойчивости системы. Применительно к распределённым системам это такие неисправности, как искусственные разрывы сети или отказы вычислительных узлов. В отличие от построения спецификаций данный метод не позволяет доказать корректность, но является гораздо менее накладным, а так же не оперирует с производными от конечного продукта(формальной моделью), что расширяет область применения. Netflix использует этот подход([19]) для тестирования своих сервисов. Многие системы с открытым исходным кодом тестировались при помощи фреймворка jepsen([8], [2]), который упрощает внесение неисправностей(искусственные разрывы сети, и т.д.). Также существуют примеры применения гибридных методик, такие как lineage-driven fault injection([1]) – когда вместо того, чтобы тестировать систему методом чёрного ящика, используется знание протокола, и избирательно теряются сообщения между узлами.

Часть 3

Постановка задачи

В данной работе мы верифицируем подсистемы Yandex.YT, а именно “Кипарис” и “Динамические таблицы”. Существуют следующие типы нарушений консистентности, которые необходимо обнаружить.

- Потеря подтверждённых записей.
- Чтение устаревшего состояния.
- Чтение данных, появившихся в результате неподтверждённых записей.

3.1 Кипарис

“Кипарис” – распределённое хранилище «ключ-значение». Является CP системой в смысле CAP-теоремы и гарантирует линеаризуемость [13].

“Кипарис” использует подход репликации конечного автомата[16] и отказоустойчивость ему придаёт слой репликации называющийся “Hydra”, алгоритм работы которого схож с идеями, применёнными в Zookeeper [3] и Raft [15].

Опишем в общих чертах алгоритм “Hydra.” Кластер состоит из $2n + 1$ машин. У каждого узла есть 3 режима работы: лидер, последователь и режим восстановления.

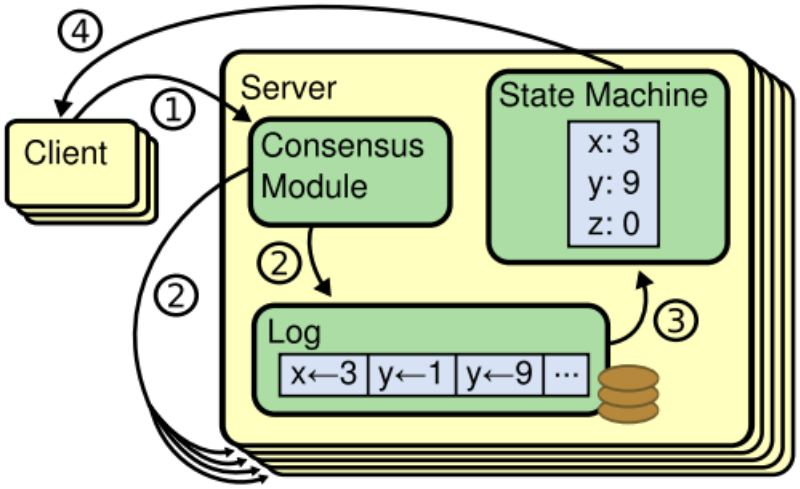

Рис. 3.1: Общая схема системы, построенной по принципу репликации конечного автомата

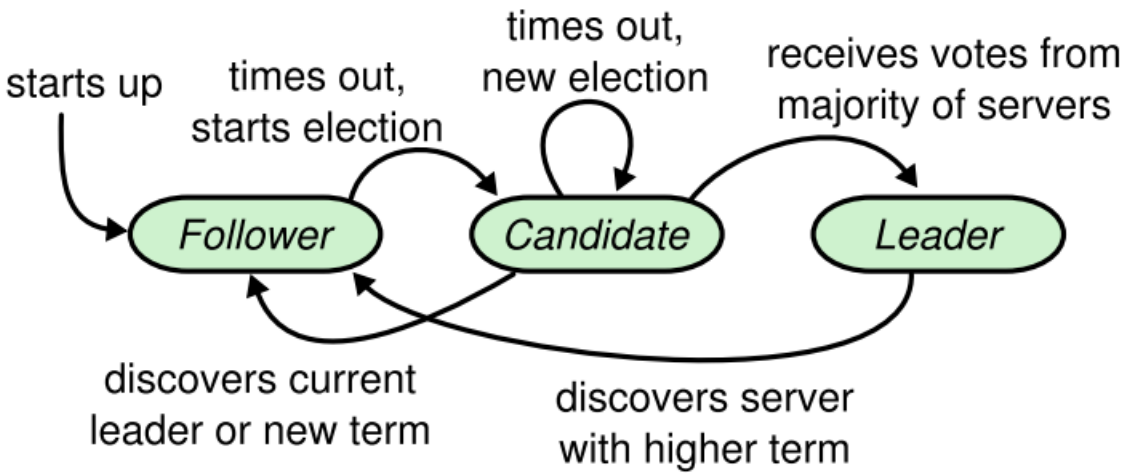

Рис. 3.2: Переходы между состояниями узлов.

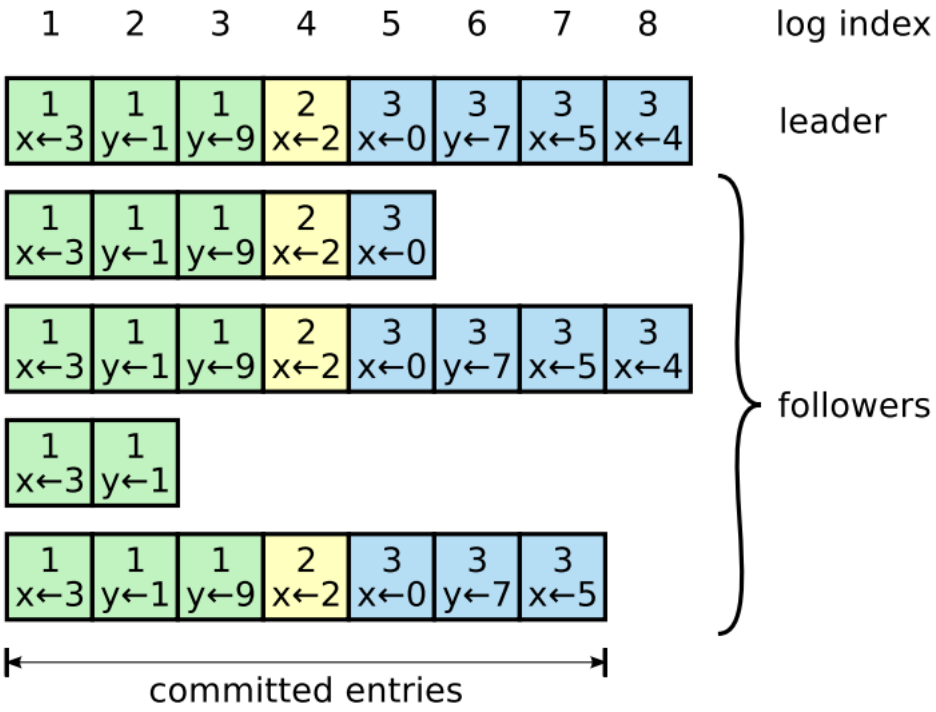

Рис. 3.3: Лог операций, хранящийся на каждом узле, состоит из записей, содержащих номер эпохи и операцию над состоянием автомата.

Лидер определяется в процессе голосования, которое устроено следующим образом:

1. Узлы рассылают всему кластеру длину записанной истории и свой идентификатор,
2. Узлы выбирают лидера (узел с наименьшим идентификатором среди имеющих наибольшую длину истории) и рассылают остальным идентификатор выбранного лидера.
3. Узел, за которого проголосовали не менее $n + 1$ узлов становится лидером.

Далее каждый узел хранит идентификатор лидера. Периодически лидер опрашивает последователей, и если не

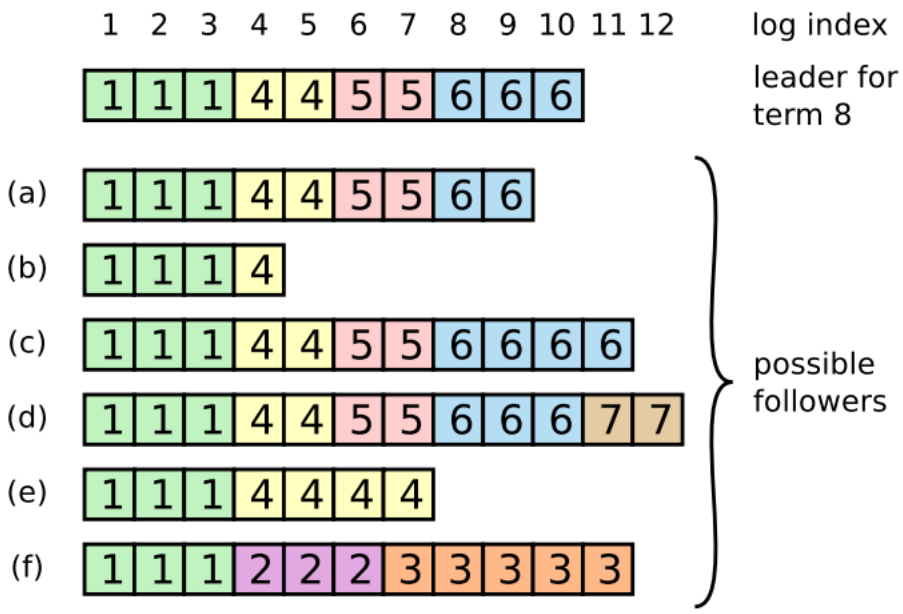

Рис. 3.4: Выборы лидера. Каждый квадрат обозначает запись в логе. (a-f) – возможные состояния последователей.

набирается n узлов, для которых верно, что идентификатор их лидера совпадает с опрашивающим узлом, то начинаются выборы. Также выборы начинаются, если в течение заранее заданного промежутка времени какой-либо из последователей не был опрошен лидером.

При записи (любом мутирующем запросе) клиент обращается к лидеру, который в свою очередь опрашивает последователей и реплицирует на них запрос. Лидер отвечает успехом если по крайней мере n последователей подтвердили что их хранимый идентификатор лидера совпадает с опрашивающим узлом, и они записали изменение на диск.

На запросы чтения лидер отвечает без подтверждения от последователей. Последователи отвечают на запрос чтения только после того как получено подтверждение от лидера о том, состояние узла не отстаёт от лидера. Узлы в режиме восстановления на запросы чтения не отвечают. Более того, узлы в режиме восстановления не голосуют при выборах, а только асинхронно

забирают изменения с лидера.

3.2 Динамические таблицы

“Динамические таблицы” – это также CP-хранилище «ключ-значение», в свою очередь являющееся Snapshot-сериализуемым[6]. В отличие от линеаризуемости, которая даёт гарантии для операций над одним объектом, Snapshot-сериализуемость даёт гарантии относительно групп объектов и групп операций над ними, объединённых в транзакции. Также отметим отличия от сериализуемости и строгой сериализуемости, которые дают гарантии атомарности транзакций. В случае snapshot-сериализуемости мы гарантируем что все запросы оперируют с состоянием базы на момент старта транзакции и то что изменения сделанные внутри транзакции не уничтожат записи сделанные с начала старта из других транзакций. Чтобы показать что snapshot-сериализуемость отличается от сериализуемости рассмотрим следующий пример истории:

$$
\begin{gathered}
1 : start \\
1 : read(2) \\
2 : start \\
2 : read(1) \\
1 : write(1) \\
1 : commit \\
2 : write(2) \\
2 : commit
\end{gathered}
$$

Здесь операции сделанные внутри транзакции x помечены как x :, $read(x)$ – операция чтения из регистра x, $write(x)$ – некоторая запись в x. $start$ и $commit$ – начало и завершение транзакции соответственно. Очевидно, что данная история не

является сериализуемой, так как тразакция 2 оперирует с значением в регистре 1 до записи сделанной в транзакции 2, а чтение внутри транзакции 1 сделано до записи внутри транзакции 2. Подобное ослабление гарантий позволяет добиться повышения производительности и применяется в таких базах данных как Oracle, Postgres[17], Microsoft SQL Server, Google Spanner[7], Cockroach DB.

Опишем в общих чертах схему работы “динамических таблиц”.

Пространство ключей статически шардировано. Для координации используется “Hydra”. А именно, для хранения информации о шардах и предоставления монотонных timestamp-меток. В "рабочем" режиме у каждый шард обслуживает некоторый узел. Через него проходят все операции на соответствующем диапазоне ключей. Для обеспечения отказоустойчивости, обслуживающий узел реплицирует историю операций на кворум из некоторых вторичных для этого шарда узлов.

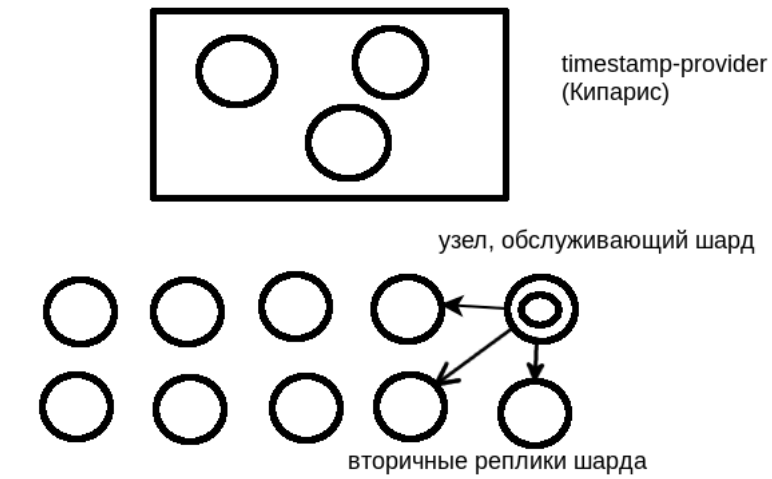

Рис. 3.5: Общая схема кластера

В данной работе мы тестируем транзакции, которые устроены следующим образом: В начале транзакции клиент получает timestamp и все чтения в рамках транзакции производит по нему же. Чтение возвращает наиболее свежую версию данных до

полученного timestamp. При завершении транзакции возможно два принципиально разных случая:

- В случае если записи затрагивают 1 шард, этот же шард просто выполняет записи.

- В случае если записи затрагивают несколько шардов, используется алгоритм двухфазного коммита [18]. А именно,

  Клиент отправляет запрос всем затронутым шардам, содержащий информацию о множестве ключей и timestamp, соответствующий началу транзакции.

  После получения положительного ответа от всех участников, случайный шард назначается координатором и на него же клиент отправляет запрос на коммит.

  Координатор отправляет запрос на подготовку коммита всем участникам.

  После подтверждения всеми участниками, генерируется timestamp и отправляется подтверждение всем участникам.

  После получения подтверждения от всех участников, отправляется ответ клиенту.

Часть 4

Предложенный алгоритм

В данной работе мы пользуемся фреймворком jepsen([8]). Общая схема процедуры верификации такова: В несколько потоков делаются серии запросов к базе, развёрнутой на кластере размера 3-5. Запросы и ответы собираются в единую последовательную историю событий, которая в дальнейшем проверяется на соответствие заявленным гарантиям.

4.1 jepsen

Jepsen представляет собой библиотеку для написания тестов и состоит из следующих частей:

- **core** – основной модуль, управляющий установкой и настройкой тестируемой базы, запуском потоков, инициирующих запросы к базе. Запросы, а также моделируемые неисправности генерируются при помощи данного на вход генератора и инициируются данным модулем. Также передаёт полученную историю модулю **checker**, и генерирует отчёт.

- **gen** – модуль, предназначенный для генерации запросов, неисправностей и т.п.. Представляет из себя

набор элементарных генераторов и операторы для их комбинирования.

- **nemesis** – модуль для внедрения неисправностей, предназначен для запуска отдельного рабочего потока, который создаёт искусственные разрывы сети между узлами, имитирует сбои отдельных узлов, манипулирует со временем на узлах и т.п..

- **model** – Предоставляет модель состояния базы, предназначен для проверки правильности результатов запросов.

- **checker** – Модуль для интеграции пользовательских анализаторов истории с jepsen.

4.2 Верификация истории

4.2.1 Кипарис

Для верификации линеаризуемости используется классический алгоритм J. Wing, C.Gong [20] с дополнениями G. Lowe [12] и A. Horn [10]. Приведём краткую схему работы алгоритма.

Определение 4.1 (Модель)**.** Для множества Ξ, некоторого множества Δ, множества операций $\chi \subset ((\{\bot\} \cup \Xi) \times \Delta)^{\Xi}$ будем называть $\langle \Xi, \Delta, \chi \rangle$ моделью.

Неформально – модель это набор состояний разделяемого объекта и поддерживаемые операции над ним. $\bot$ означает что операция над данным объектом не поддерживается.

Определение 4.2 (История)**.** Для модели $\langle \Xi, \Delta, \chi \rangle$ будем называть историей последовательность событий вида

- *call.process.op*, где process – идентификатор процесса, $op : \Xi \to \langle \Xi, \Delta \rangle$ – производимая операция над состоянием разделяемого объекта.

- *ret.process.response* process – идентификатор процесса, $response = \langle status, value \rangle, value \in \Delta, status \in \{\mathbf{fail}, \mathbf{ok}, \mathbf{info}\}$ Здесь и далее

  fail – детерминировано отвергнутая операция,

  ok – успешная операция.

  info – неизвестно, отвергнута операция или нет(моделирует превышение таймаута и проч.).

Определение 4.3 (Полная история)**.** История является полной, если история, ограниченная на любой процесс, удовлетворяет следующим требованиям:

- История начинается с *call*-записи.
- За каждой *call*-записью которая не является последней в истории следует соответствующая *ret*-запись, за *ret* записью может следовать только другая *call*-запись.
- Для всех *ret*-записей $status = \mathbf{ok}$.

Алгоритм оперирует с полной историей, которая может быть получена из реальной удалением детерминировано неудачных операций и **info**-записей и представляет из себя обход некоторого графа состояний.

Также отметим что при проведении экспериментов в случае если операция завершилась с **info** то идентификатор процесса данной операции далее использоваться не может и соответствующий логический процесс необходимо считать завершённым.

Определение 4.4 (Состояние)**.** Будем называть состоянием $\langle state, history \rangle$ где $state \in \Xi$, *history* – полная история.

Для $\langle state, history \rangle$ будем считать смежными состояния вида $op[state], history \backslash op$ где op – *call* запись принадлежащая *history* которой не предшествует ни одна *ret* запись, $history \backslash op$ – история с удалённой op и соответствующей *ret*-записью (если таковая есть).

История считается линеаризуемой при достижимости состояния с *history* без *ret*-записей из начального состояния, в котором *history* – полная исследуемая история. Корректность данного утверждения доказана в [20].

При постановке экспериментов использовалась реализация из библиотеки knossos([11]), запоминающая исследованные состояния в хеш-таблице. Также, перед началом работы алгоритма явно строится граф состояний и переходов для модели. Это уменьшает потребление памяти и избавляет алгоритм от обработки специфичной логики для переопределённой пользователем модели. Также, в хеш-таблице сохраняются не истории а множества линеаризованных операций. Это также ускоряет алгоритм, так как множества представлены последовательностью бит, в силу особенностей современных процессоров такие структуры обрабатываются быстрее чем односвязные списки.

4.2.2 Динамические таблицы

Для верификации snapshot-сериализуемости использовалась модификация вышеупомянутого алгоритма J. Wing, C.Gong, G. Lowe.

При описании алгоритма состояние базы рассматривается как один разделяемый объект со специфичными операциями чтения и записи. Соответственно мы объединяем все чтения и старт соответствующие транзакции в одну операцию, а также все записи и коммит. В данной постановке необходимо верифицировать отсутствие конфликтующих записей из других транзакций между соответственными чтением и записью.

Далее оперируем с моделью $\Xi = S^n$, для некоторого количества регистров n и множества значений S.

Поддерживаемые операции:

- $start(S)$ Начать транзакцию, прочитать значения регистров из S.

- *commit*(S, V) Завершить транзакцию, записать в регистры из S значения V.

Верифицируемые истории при ограничении на любой процесс должны являться чередующимися последовательностями *start* и *commit* операций. Более того, мы требуем чтобы ограниченные истории начинались с *start* операции.

Сформулируем определение snapshot-сериализуемости для описанной модели.

Определение 4.5. Будем называть историю H snapshot-сериализуемой если существует линеаризация [13]. В которой для любой пары (u, v) соответствующих *start* и *commit* операций, между u и v нет ни одной *commit* операции которая записывает в регистры, записываемые v.

Первым шагом алгоритм преобразует исходную историю в некоторое множество историй для другой модели $\Xi = (S \times \{0, 1\})^n$.

Неформально, транзакции будут блокировать регистры в которые планируют писать и соответственно состояние разделяемого объекта теперь включает в себя информацию о заблокированных ячейках. Далее будем говорить о регистрах, которые могут быть заблокированы.

Для каждой пары соответствующих *start* и *commit* операций

- Если *commit* операция завершилась **fail**-записью, обе записи соответствующие *commit* операции удаляются. *start* операция преобразуется в *read* операцию для того-же множества регистров. То есть не меняющее состояние объекта чтение.

- Если *commit* операция завершилась **ok**-записью, то *start*-операция преобразуется в *read* операцию, блокирующую регистры, в которые пишет *commit* операция. Соответственно операция определена только на объектах, в которых соответствующие регистры не заблокированы. *commit* операция преобразуется в *write* операцию, при этом

разблокирующую регистры в которые пишет. Соответственно определена только на объектах, в которых соответствующие регистры заблокированы.

- Если *commit* операция не завершилась или завершилась **info**-записью, то рассматриваем два варианта истории, преобразованной одним из вышеупомянутых способов.

Таким образом получаем некоторое множество историй.

Лемма 4.1. *Snapshot-сериализуемость исходной истории равносильна линеаризуемости одной из историй, полученных приведённым преобразованием.*

Доказательство. В одну из сторон приведённое утверждение очевидно. То есть из snapshot-сериализуемости очевидна линеаризуемость одной из полученных историй. Пусть T получена из H и L – линеаризация T. Покажем как построить линеаризацию H удовлетворяющую 4.5. Рассмотрим L' соответствующую L. В том смысле, что операции в L' получаются из операций в L игнорированием блокировки регистров. *read*-записям соответствуют *start*-записи, *write* переходят в *commit*. Рассмотрим ограничение L на один из регистров. Также удалим неблокирующие чтения. Из определения операций очевидно, что чтения и записи чередуются, и история начинается с чтения. Покажем, что история устроена следующим образом: каждой записи предшествует соответствующее чтение(с тем же идентификатором процесса), устанавливающее блокировку и за каждым чтением следует соответствующая запись. От противного: пусть пара u, v – последовательные чтение и запись с разными идентификаторами процесса. Тогда в истории должно присутствовать соответствующее v чтение w. Более того, оно должно быть расположено перед u. Тогда после него должна быть запись с другим идентификатором процесса. Получаем противоречие с тем что пара u, v первая. Пусть теперь пара u, v в L' – соответствующие *start* и *commit* операции. Пусть между u и v есть w которая пишет в один из регистров, в которые пишет

v. Тогда ограничивая L на этот регистр получаем противоречие вышедоказанному. □

Алгоритм устроен следующим образом:

- Сначала строится преобразование истории: при неоднозначности преобразования элементом истории является кортеж альтернатив. у *ret*-записей альтернатив нет. Для некоторых записей помечаем что при выполнении следующую операцию данного процесса надо игнорировать.
- Удаляются все **fail** и **info** записи. Для **fail**-записей необходимо удалить также **call**-запись.
- Производится поиск в пространстве состояний, схожий с описанным в разделе 4.2.1.

Состоянием так же является $\langle state, history \rangle$. При вычислении смежных состояний мы рассматриваем все альтернативы для каждой **call**-записи, и удаляем из истории следующую операцию процесса(если помечено что надо её проигнорировать). При поиске запоминаем пройденные состояния, в том смысле что запоминаем пару (множество удалённых операций, состояние). В [10] запоминали именно пару $\langle state, history \rangle$, что создавало некоторое замедление из-за необходимости сравнения историй, которые хранились в неизменяемом односвязном списке. Авторы предлагают кешировать результаты сравнения списков для ускорения работы алгоритма. Мы же, поддерживая множество удалённых операций(bitset) добились сравнимой производительности при меньшей сложности алгоритма.

Как и в алгоритме, описанном в 4.2.1, при постановке экспериментов, перед началом работы алгоритма явно строится граф состояний и переходов для модели, мемоизируется именно множество удалённых операций, но в отличие от [11], мы используем неизменяемые списки, что с одной стороны замедляет алгоритм, но с другой позволяет эффективно распределить вычисления на несколько процессорных ядер и результирующее время работы (на синтетических тестах) получается меньше.

Номер теста	Длина истории	knossos	наша реализация
1	20	0.2с	2с
2	200	40с	10с
3	800	1200с	100с

Все тесты производились на 16-ядерном процессоре, с доступным объёмом памяти 60G.

Часть 5

Проведённые эксперименты

5.1 Кипарис

Было исследовано поведение системы при следующих условиях.

- Использовалась модель атомарного регистра, со следующими доступными типами операций:
  - **read** – Чтение значения регистра, возможны исходы:
    - **ok** – Удалось прочитать значение регистра.
    - **fail** – Не удалось прочитать значение.
  - **write** – Запись в регистр, возможны исходы:
    - **ok** – Удалось записать значение.
    - **fail** – Хранилище отклонило операцию.
    - **unknown** – Возможно удалось записать значение (хранилище не отклонило и не подтвердило операцию).

  Соответственно, все операции с хранилищем производились по одному и тому же ключу.

- Запросы осуществлялись в 8 потоков, чтение с последователей разрешено.

- Для обнаружения ошибок в сетевую конфигурацию вносились сбои следующего вида:

  Кластер разбивался на две примерно равные части, также было исследовано поведение при разбиении на две равные части с одним узлом “перемычкой”, который видит остальные.

  Также было исследовано поведение алгоритма при периодических обрывах сети между текущим лидером и остальными узлами.

В исходной постановке считалось что исход любой неудавшейся записи – **fail**. После ряда экспериментов, были обнаружены аномалии вида "чтение неподтверждённых записей". После анализа исходного кода, было обнаружено, что запросы записи нельзя однозначно считать отклонёнными системой. В скорректированной постановке, ошибок линеаризации на данный момент обнаружено не было.

5.2 Динамические таблицы

Использовался алгоритм, описанный в 4.2.2. Так как начало транзакции в исследуемой системе скорее логическое(клиент получает timestamp, и в дальнейшем все чтения осуществляет по нему, то *start* операции, завершившиеся по таймауту можно считать завершившимися с **fail** статусом. Было исследовано поведение системы при следующих условиях.

- Размер кластера – 5 узлов, 1 узел – timestamp-provider, 5 шардов.

- Запросы осуществлялись в 6 потоков. Было исследовано поведение системы при записях на 1 шард и на 2 шарда(распределённые коммиты).

- Для обнаружения ошибок в сетевую конфигурацию вносились сбои следующего вида:

  Кластер разбивался на две примерно равные части.

  Избирательно обрывалась сеть между узлами, обслуживающими случайный шард и мастером.

В каждом шарде находится по одному ключу, по которому в дальнейшем и делаются запросы чтения/записи.

Все эксперименты проводились с rpc_timeout=3000 на всех узлах. Общий таймаут на транзакцию – 6.3 секунд.

Тест 1 – успешно верфицировано соблюдение гарантий. С перерывами по 8 секунд обрывается связь с узлом обслуживающим один шард на 8 секунд. Каждая транзакция состоит из двух чтений и двух записей. Запросы делаются с промежутками в 0.2 секунды в 5 потоков. Время сбора истории – 80 секунд, время верификации 11 минут.

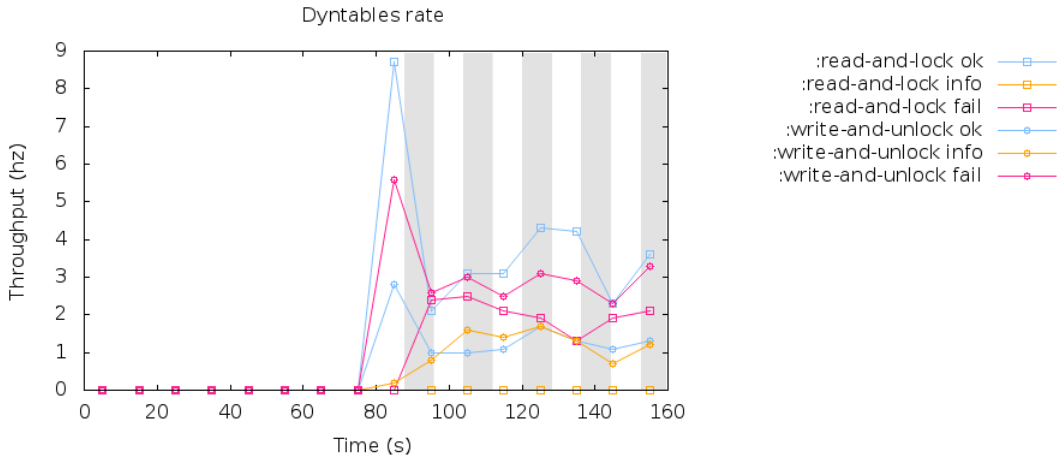

Рис. 5.1: Частота запросов, тест 1

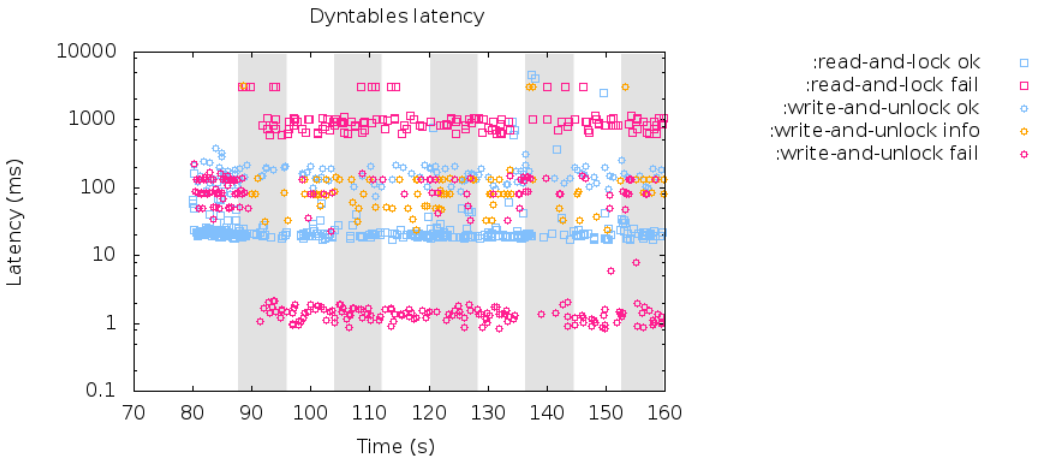

Рис. 5.2: Время отклика, тест 1

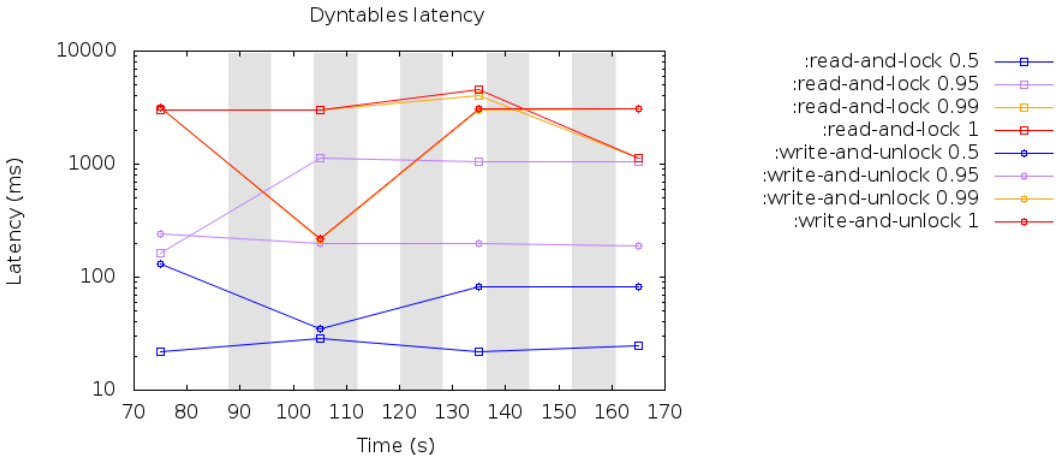

Рис. 5.3: Квантили времён отклика, тест 1

Список литературы

[1] Peter Alvaro, Joshua Rosen, and M. Hellerstein Joseph. “Lineage-driven fault injection”. In: *ACM* (2015). Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.

[2] *Analyses*. URL: `http://jepsen.io/analyses`.

[3] *Apache Zookeeper*. URL: `https://zookeeper.apache.org/`.

[4] Peter Bailis and Kyle Kingsbury. “The network is reliable”. In: (2014).

[5] Bruno Barras. *The Coq proof assistant reference manual: Version 6.1*. 1997.

[6] Michael J Cahill, Uwe Röhm, and Alan D Fekete. “Serializable isolation for snapshot databases”. In: *ACM Transactions on Database Systems (TODS)* 34.4 (2009), p. 20.

[7] James C Corbett et al. “Spanner: Google’s globally distributed database”. In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[8] *Distributed Systems Safety Research*. URL: `http://jepsen.io/`.

[9] A. Hall. “Seven myths of formal methods”. In: (2002).

[10] Alex Horn and Daniel Kroening. “Faster linearizability checking via p-compositionality”. In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer. 2015, pp. 50–65.

[11] Kyle Kingsbury. *Computational techniques in Knossos*. 2014. URL: https://aphyr.com/posts/314-computational-techniques-in-knossos.

[12] Gavin Lowe. "Testing for linearizability". In: *Concurrency and Computation: Practice and Experience* 29.4 (2017).

[13] J. Wing M. Herlihy. "Linearizability: a correctness condition for concurrent objects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* (3 1990), 463—492.

[14] Chris Newcombe et al. "Use of formal methods at Amazon Web Services". In: (2014).

[15] Diego Ongaro and John K Ousterhout. "In search of an understandable consensus algorithm." In: *USENIX Annual Technical Conference*. 2014, pp. 305–319.

[16] Nuno Filipe de Sousa Santos. "State Machine Replication". In: (2012).

[17] Michael Stonebraker and Lawrence A Rowe. *The design of Postgres*. Vol. 15. 2. ACM, 1986.

[18] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[19] *The Netflix Simian Army*. 2011. URL: http://techblog.netflix.com/2011/07/netflix-simian-army.html.

[20] Jeannette M. Wing and Chun Gong. "Testing and verifying concurrent objects". In: *Journal of Parallel and Distributed Computing* 17.1-2 (1993), pp. 164–182.