Содержание

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. В частности, при тестировании распределённых баз данных возникает потребность в верификации гарантий, которые даёт тестируемая система. В данной работе используется метод внесения неисправностей(fault injection) для верификации подсистем Yandex.YT.

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 [8] 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 ([13]) 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([15]) для тестирования своих сервисов. Многие системы с открытым исходным кодом тестировались при помощи фреймворка jepsen([7], [2]), который упрощает внесение неисправностей(искусственные разрывы сети, и т.д.). Также существуют примеры применения гибридных методики, 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 [12].

Схема репликации “Кипариса” схожа с идеями, применёнными в Zookeeper [3] и Raft [16].

“Кипарис” запущен на кластере из $2n+1$ машин. У каждого узла “Кипариса” есть 3 режима работы: лидер, последователь и режим восстановления.

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 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]. Пространство ключей статически шардировано. Для координации

используется “Кипарис”. А именно, в нем хранится информация о шардах. Также “Кипарис” служит timestamp-поставщиком.

Каждый шард устроен следующим образом: некоторый узел объявляется главным, статически назначаются вторичные узлы на которые будут реплицироваться изменения главного Далее при каждой операции на главном узле собирается кворум вторичных узлов.

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 [14]. А именно,

  Клиент отправляет запрос всем затронутым шардам.

  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([7]). 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 [17] с дополнениями G. Lowe [11] и A. Horn [9]. Приведём краткую схему работы алгоритма.

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

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 [17].

При постановке экспериментов использовалась реализация из библиотеки knossos([10]), запоминающая исследованные состояния в хеш-таблице. Также, перед началом работы алгоритма явно строится граф состояний и переходов для модели, что сильно уменьшает время работы.

4.2.2 Динамические таблицы

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

Данный алгоритм работает с конкретной моделью базы Ξ. А именно, для некоторого количества регистров n и множества значений S $\Xi = S^n$

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 [12]. 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**-записью.
- 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 [9] 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, что сильно уменьшает время работы.

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 **??**. 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 кипариса, 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.

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

[8] A. Hall. "Seven myths of formal methods". In: (2002).

[9] 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.

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

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

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

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

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

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

[16] *The Raft Consensus Algorithm*. URL: `https://raft.github.io/`.

[17] 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.