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 ([6]) 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([10]) 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 [11].

“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.

Snapshot-сериализуемость понимается как гарантия того, что каждая транзакция оперирует с консистентным состоянием хранилища на некоторый момент времени, а также что выполняются условия последовательной консистентности.

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. После этого собирается история ответов, которая проверяется на соответствие заявленным гарантиям следующим образом. Ищется произвольное "правильное" упорядочение запросов. Для проверки линеаризуемости используется библиотека knossos([9]).

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 knossos

Данная библиотека помогает найти возможное упорядочение запросов, удовлетворяющее выбранной модели, или убедиться в его отсутствии. Поиск реализован с помощью перебора с отсечениями. Здесь и далее называем состоянием состояние базы и набор не применённых операций. Храня состояния в очереди с приоритетами, будем в n потоков доставать те из них, для которых ожидаемое количество продолжений истории наименьшее, и делать попытку продолжить ещё на один шаг. Для ускорения общая очередь разбивается на несколько очередей, привязанных к потокам. Кроме того, используется мемоизация – если состояние уже было исследовано, то второй раз его исследовать не имеет смысла.

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.

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

Пока не тестировались.

Список литературы

[1] Joseph M. Hellerstein Alvaro Peter Joshua Rosen. “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] Fan Zhang Bogdan Munteanu Marc Brooker Michael Deardeuff Chris Newcombe Tim Rath. “Use of Formal Methods at Amazon Web Services”. In: (2014).

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

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

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

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

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

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