Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра системного программирования

Морозов Сергей Валерьевич

Верификатор облачного объектного хранилища данных

Бакалаврская работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
д. ф.-м. н., профессор Нестеров В. М.

Рецензент:
ст. инженер-программист в Санкт-Петербургском Центре Разработок EMC Мицай А. И.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Chair of Software Engineering

Sergey Morozov

Cloud Object Storage Verifier

Bachelor's Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
professor Vyacheslav Nesterov

Reviewer:
senior software engineer at EMC St. Petersburg Development Centre Alexander Mitsai

Saint-Petersburg
2015

Оглавление

Введение

Облачные вычисления, согласно определению Национального института стандартов и технологий США, это модель, обеспечивающая удобный, повсеместный доступ по сети к общим вычислительным ресурсам, которые могут быть введены в пользование с минимальными усилиями как со стороны заказчика, так и со стороны поставщика услуг [14]. Облачная система состоит из одного или нескольких, соединенных между собой, центров хранения и обработки данных. Часто центры хранения и обработки данных одной облачной системы располагаются на разных континентах.

Существует несколько категорий облачных систем: публичные, частные, общественные, гибридные [14]. Компаниям малого и среднего размера выгоднее, с экономической точки зрения, использовать публичные облачные системы как для внутренних целей, так и для предоставления сервисов своим клиентам. Компаниям большого размера дешевле содержать собственную облачную систему.

Облачные вычисления принесли с собой новые парадигмы программирования. Ярким примером является модель распределенных вычислений MapReduce, предложенная компанией Google [13]. Бесплатной реализацией парадигмы MapReduce с открытым исходным кодом является Apache Hadoop[1]. Еще одним примером является ряд подходов к построению баз данных, сильно отличающихся от традиционного подхода к построению реляционных СУБД, обозначаемых одним общим термином NoSQL[2]. Apache Cassandra[3] – распределенная СУБД, разработанная в соответствии с подходами NoSQL и рассчитанная на создание высокомасштабируемых и надёжных хранилищ огромных массивов данных.

Вместе с новыми парадигмами появились и новые проблемы. В 2000 году Эриком Брюером, профессором Калифорнийского университера в Беркли, была сформулирована теорема, согласно которой в распределенной системе, такой как облачная система хранения, возможно обеспечить не более двух из трех следующих свойств: согласованность данных (операция чтения возвращает последнее значение, записанное любым из клиентов распределенной системы), доступность данных (выполнение любых операций разрешено узлам распределенной системы в каждый момент времени), устойчивость к разделению (распределенная система продолжает корректно работать, несмотря на разделение системы на части) [6]. Устойчивость к разделению – важное свойство для облачной системы хранения. Когда связь между двумя центрами хранения и обработки данных прерывается, важно обеспечить автономное обслуживание клиентов обеими частями облачной системы. Поэтому приходится выбирать между согласованностью и доступностью данных. Часто доступность предпочитают согласованности.

[1] http://hadoop.apache.org/
[2] http://martinfowler.com/bliki/NosqlDefinition.html
[3] http://cassandra.apache.org/

Одним из примеров облачной объектной системы хранения, где доступность и устойчивость к разделению предпочли согласованности, является Amazon Simple Storage Service[4] (Amazon S3). Чтобы уменьшить задержку передачи данных, сервис Amazon S3 позволяет выбрать точку доступа в ближайшем регионе мира. Для запросов, поступающих на точку доступа в регион US Standard, сервис Amazon S3 гарантирует свойство «согласованность в конечном счете» (*eventual consistency*) [1]. Свойство «согласованность в конечном счете» означает следующее: если полностью остановить поток записи на облачное хранилище данных, то гарантируется, что в обозримом будущем облачная система придет в согласованное состояние. Существуют и другие облачные хранилища данных, где предпочтение было отдано двум другим свойствам: устойчивости к разделению и согласованности данных. При этом требования к доступности данных ослаблены.

Одним из методов общения клиентов с облачной объектной системой хранения Amazon S3 является общение через REST интерфейс [15]. REST интерфейс – это метод взаимодействия компонентов распределенного приложения, при котором вызов удаленной процедуры представляет собой обычный HTTP-запрос, а необходимые данные передаются в качестве параметров запроса. В документации к сервису Amazon S3 подробно описывается как выполнить то или иное действие (записать объект в облачное хранилище, прочитать объект из облачного хранилища, удалить объект и др.) используя REST интерфейс [1]. Протокол взаимодействия клиента с облачным объектным хранилищем Amazon S3 будем называть протоколом S3. Многие другие облачные объектные системы хранения поддерживают работу в соответствии с этим протоколом.

При тестировании облачного объектного хранилища данных разработчикам необходим инструмент, позволяющий создавать нагрузку на систему хранения потоками запросов. Такой инструмент называется *генератором нагрузки*. Как правило, перед запуском генератора нагрузки разработчик создает сценарий нагрузки, а затем запускает генератор, входным параметром которого является созданный сценарий, и смотрит на поведение хранилища данных. Одним из самых известных генераторов нагрузки на облачное объектное хранилище данных является COSBench, созданный компанией Intel [8].

Несмотря на ослабленное свойство согласованности некоторых облачных объектных хранилищ данных, таких как Amazon S3, пользователь ожидает корректной работы сервиса, в том числе и выполнения свойства «согласованность в конечном счете». В случае частного облачного хранилища проверка на корректность работы может быть выполнена с помощью профилирования. Но в случае публичного облака пользователь не может внедриться в систему облачного хранилища, чтобы проверить действительно ли облачная система работает корректно. В этом случае необходимо проверять кор-

[4]http://aws.amazon.com/s3/

ректность работы системы извне, то есть делать проверку на уровне приложения. На сегодняшний момент уже разработан ряд протоколов, позвляющих удостовериться, что облачная система работает в соответствии с заявленными требованиями, а именно выполняется свойство «согласованность в конечном счете». Если эти требования нарушаются, то программа, реализующая протокол, оповещает об этом пользователя.

Venus – это протокол, описывающий способ взаимодействия клиентов с облачным объектным хранилищем данных и позволяющий выявлять нарушения в работе облачного хранилища [36]. При этом не требуется производить никаких изменений на стороне облачного объектного хранилища данных. Помимо согласованности данных этот протокол позволяет проверять и целостность записанных данных. Большим преимуществом этого протокола является увеличение времени обработки запросов при работе с облачным объектным хранилищем лишь на малое константное время.

1. Постановка задачи

У облачных объектных хранилищ есть множество механизмов защиты данных. Система, позволяющая верифицировать работу облачного хранилища извне, необходима для того, чтобы выявить нарушения в работе сервиса. Пользователь (специалист по тестированию) изначально не вполне доверяет корректности работы облачной системы хранения. Поэтому он не может полагаться на внутренние механизмы верификации корректности работы.

Ни у одного генератора нагрузки на облачное хранилище нет встроенной функции проверки согласованности данных. Целостность данных как правило проверяется, но не всегда. Предлагается вынести функцию проверки целостности и согласованности данных в отдельную «прослойку» между облачной системой хранения и различными генераторами нагрузки, оставив таким образом генераторам только их прямую обязанность – создавать массивные потоки ввода-вывода.

Наиболее приближенным к практике протоколом по верификации целостности и согласованности данных является протокол Venus. Авторы статьи [36] теоритически доказали работоспособность протокола, реализовали его и провели развернутое тестирование на реальном облачном хранилище. К сожалению, исходного кода этой реализации нет в свободном доступе, поэтому было решено создать свою реализацию, адаптированную для тестирования облачных объектных хранилищ.

Целью работы является создание системы, позволяющей тестировать облачное объектное хранилище на выполнение требований целостности и согласованности данных. Для достижения цели были поставлены следующие задачи:

- Анализ проблем, возникающих при разработке облачных объектных хранилищ, и протоколов верификации целостности и согласованности данных.
- Анализ протокола Venus и внесение модификаций, адаптирующих его для тестирования облачных объектных хранилищ.
- Реализация протокола Venus, включающая в себя предложенные модификации и удовлетворяющая следующим требованиям:
  - не вносить изменения в существующие приложения (генераторы нагрузки), работающие с облачной объектной системой хранения по протоколу S3;
  - сохранить свойство оригинального протокола Venus замедлять взаимодействие с облачным объектным хранилищем на небольшую константу.

2. Обзор

2.1. Основные понятия

В этой секции будут даны определения, в основном относящиеся к теории распределенных систем, знание которых облегчит понимание следующих разделов работы. Понятия клиент и процесс следует считать синонимами.

Доступность – свойство распределенной системы, гарантирующее, что процесс, пославший запрос, в конечном счете (через какое-то время) получит ответ [17].

Устойчивость к разделению – свойство распределенной системы, гарантирующее, что распределенная система продолжит корректно работать даже в случае разрыва связи между ее частями. Иными словами, даже в случае потери произвольного количества запросов, распределенная система продолжит корректно работать [17].

Модель согласованности – представляет собой контракт между процессами и хранилищем данных. Он гласит, что если процессы согласны соблюдать некоторые правила, то хранилище соглашается работать правильно [34].

Строгая согласованность (strong consistency) – модель согласованности, где всякое чтение элемента данных x возвращает значение, соответствующее результату последней записи x [34].

Причинная согласованность (casual consistency) – менее строгая модель согласованности, чем строгая согласованность, где операции записи, которые потенциально связаны причинно-следственной связью, должны наблюдаться всеми процессами в одинаковом порядке, а параллельные операции записи могут наблюдаться на разных машинах в разном порядке [34].

Согласованность в конечном счете (eventual consistency) – менее строгая модель согласованности, чем причинная согласованность, где хранилище данных гарантирует, что если полностью прекратится поток запросов на запись, то в конечном счете (через какое-то время) операция чтения, вызванная любым из процессов, вернет последнее записанное значение. Если предположить, что ошибок в системе не происходит, то максимальный промежуток времени, когда данные будут несогласованы, может быть вычислен исходя из задержек связи между процессами и хранилищем данных, общей загрузки системы и прочих факторов [37].

Согласованность чтение-после-записи (read-after-write consistency) – модель согласованности, более сильная, чем согласованность в конечном счете, где чтение данных после операции записи возвращает последнюю версию данных для всех клиентов системы [27]. Аналигично определяются модели согласованности *согласованность чтение-после-обновления* и *согласованность чтение-после-удаления*.

Согласованность с возможностью разветвления (fork-consistency) – менее строгая модель согласованности, чем строгая согласованность, где гарантируется, что ес-

ли при выполнении операции чтения разные процессы получили разные результаты, то они никогда больше не увидят обновлений друг друга без признания ошибки со стороны системы. Такая ошибка может быть найдена с помощью межпроцессного взаимодействия [21].

Слабая согласованность с возможностью разветвления (weak fork-consistency) – менее строгая модель согласованности, чем согласованность с возможностью разветвления, где гарантируется, что если при выполнении операции чтения разные процессы получили разные результаты, то они либо не увидят обновлений друг друга без признания ошибки со стороны системы, либо увидят, но через определенный промежуток времени (в конечном счете) ошибка будет выявлена с помощью межпроцессного взаимодействия [5].

Теорема CAP (теорема Брюера) – эвристическое утверждение, сформулированное профессором Калифорнийского университера в Беркли Эриком Брюером в 2000 году и позднее формализованное и доказанное Сетом Гильбертом и Нэнси Линчем в 2002 году в работе [17]. Приведем неформализованный вариант формулировки этой теоремы. В распределенной системе невозможно гарантировать одновременное выполнение следующих трех свойств: согласованность, доступность и устойчивость к разделению. В теореме CAP понятие доступность следует понимает следующим образом: все узлы распределенной системы, в случае разделения системы на части, имеют возможность обрабатывать запросы на чтение и на запись, причем обработка занимает определенный (не бесконечный) промежуток времени [7].

Кворум – минимальное количество голосов (положительных ответов клиентов, одобряющих выполнение операции), которые клиент, выполняющий операцию, должен получить, чтобы считать операцию выполненной успешно [23].

Отношение "выполняется прежде" (обозначается $\rightarrow$) – это отношение частичного порядка на множестве событий распределенной системы, удовлетворяющее следующим трем условиям: (1) если a и b события, выполненные одним и тем же процессом, и событие a было выполнено до события b, то $a \rightarrow b$; (2) если a событие, обозначающее посылку сообщения одним процессом, а b событие, обозначающее прием этого сообщения другим процессом, то $a \rightarrow b$; (3) если $a \rightarrow b$ и $b \rightarrow c$, то $a \rightarrow c$. Если $a \not\rightarrow b$ и $b \not\rightarrow a$, то события a и b называются *параллельными* [20].

Векторные часы – алгоритм получения частичного упорядочения событий в распределенной системе и обнаружения нарушений причинно-следственных связей. Векторные часы устанавливают взаимосвязь между событиями и соответствующими векторными временны́ми метками. Появляется возможность выявить причинно-следственные связи между событиями путем сравнения их векторных временных меток. Каждый процесс в системе имеет вектор $VC_i[0..n]$ (изначально все элементы проинициализированы нулями), который обновляется следующим образом:

- Каждый раз, когда происходит какое-либо событие (посылка сообщения, полу-

чение сообщения, внутреннее событие), процесс P_i увеличивает значение $VC_i[i]$ ($VC_i[i] := VC_i[i] + 1$).

- Когда процесс P_i посылает сообщение m, он прикладывает к нему текущие векторные часы VC_i. Обозначим это значение векторных часов как $m.VC$.

- Когда процесс P_j получает сообщение m, он обновляет элементы своих векторных часов следующим образом: $VC_j[x] := \max(VC_j[x], m.VC[x])$ для любого x.

Сформулируем *свойство векторных часов*: если $a \to b$, то $a.VC < b.VC$, где a и b различные события. Здесь запись $a.VC < b.VC$ обозначает следующее: $\forall k(a.VC[k] \leq b.VC[k])\&\exists k(a.VC[k] < b.VC[k])$ [3].

2.2. Облачные объектные хранилища данных

Объектом хранения называется набор байтов на устройстве хранения с известными методами доступа, атрибутами, характеризующими хранимые данные, и политиками безопасности, предотвращающими несанкционированный доступ. Устройство, хранящее такие объекты, называется *объектным хранилищем данных* [22]. Объектное хранилище называется *облачным* объектным хранилищем, когда оно развернуто в облачной системе. В этой секции будут рассмотрены три примера облачных объектных хранилищ данных, поддерживающих работу по протоколу S3.

2.2.1. Amazon Simple Storage Service

Amazon S3 – это веб-сервис, предназначенный для хранения и получения любого объема данных в любое время из любой точки сети [1]. С помощью Amazon S3 достигается высокая масштабируемость, надёжность, высокая скорость и недорогая инфраструктура хранения данных. Компания Amazon сама же использует этот веб-сервис для разворачивания глобальной сети веб-сайтов.

Есть несколько ключевых концепций, которые необходимо понимать, чтобы эффективно работать с сервисом Amazon S3:

- *Контейнер bucket* – это контейнер для объектов, хранимых в облачном объектном хранилище Amazon S3. Каждый объект принадлежит какому-либо контейнеру bucket. Контейнеры bucket выполняют несколько функций: на самом общем уровне организуют пространство имен облачного объектного хранилища Amazon S3, отвечают за идентификацию аккаунта, с которого будет списываться плата за используемые ресурсы, являются важными элементами при контроле доступа к данным и рассматриваются как отдельные единицы при сборе статистики использования сервиса [1].

- *Объект* – фундаментальная сущность, хранимая в облачном объектном хранилище Amazon S3. Объект состоит из данных и мета-данных. Данные являются для Amazon S3 «непрозрачными», то есть Amazon S3 может только сохранять и возвращать данные объекта, но не может проследить их внутреннюю структуры. Мета-данные объекта – это множество пар вида ключ-значение, описывающих объект. Каждый объект в облачном объектном хранилище Amazon S3 может быть однозначно определен по идентификатору контейнера bucket, ключу и идентификатору версии объекта [1]. Понятие версии объекта рассматриваться не будет.

- *Ключ* – это уникальный идентификатор объекта в пространстве контейнера bucket. Каждому объекту в контейнере bucket соответствует ровно один ключ [1].

- *Регион* – географическое местоположение хранения данных в контейнере bucket. Выбор определенного региона может быть полезен в следующих случаях: для оптимизиции времени задержки (латентности), минимизации расходов и для обхождения регуляторных норм [1].

Сервис Amazon S3 гарантирует высокую доступность данных путем их репликации по серверам различных центров обработки данных компании Amazon, а также устойчивость к разделению системы на части. Требования к согласованности данных ослабляются: гарантируется согласованность в конечном счете для всех типов запросов в регионе US Standard. Для всех остальных регионов гарантируется согласованность чтение-после-записи для всех запросов типа PUT (запросов на запись), когда создается новый объект, и согласованность в конечном счете для запросов типа PUT, когда обновляется уже существующий объект, и для запросов типа DELETE, когда происходит удаление существующих объектов [1].

2.2.2. Google Cloud Storage

Google Cloud Storage[5] – это веб-сервис, аналогичный сервису Amazon S3, предназначенный для хранения и получения любого объема данных в любое время из любой точки сети. Основные концепции схожи с концепциями Amazon S3, но есть и некоторые различия.

Все данные в Google Cloud Storage принадлежат *проекту*. Проект состоит из набора пользователей и набора интерфейсов программирования приложений (API). К проекту привязана функция выставления счетов, аутентификации и настройки мониторинга интерфейсов программирования приложений. Пользователь Google Cloud Storage может иметь один или несколько проектов [18]. В облачном объектном хранилище Google Cloud Storage объединение концепций проекта и контейнера bucket

[5]https://cloud.google.com/storage/

соответствует концепции контейнера bucket в Amazon S3. Остальные концепции схожи с концепциями, описанными для веб-сервиса Amazon S3.

Сервис Google Cloud Storage предоставляет более высокий уровень согласованности данных, чем сервис Amazon S3. Гарантируется строгая согласованность данных для следующих операций: чтение после записи, чтение после обновления, чтение после удаления объекта. Операция составления списка объектов, содержащихся в контейнере bucket, является согласованной в конечном счете [18].

2.2.3. EMC ViPR Software-Defined Storage

EMC ViPR Software-Defined Storage[6] – это *программно-определяемое хранилище данных* [32]. Одним из сервисов, которые предоставляет EMC ViPR Software-Defined Storage, является сервис объектного хранения данных EMC ViPR Object Storage.

EMC ViPR Object Storage поддерживает работу по протоколу S3, а также в некоторой степени расширяет его, добавляя дополнительную функциональность:

- допускатеся модификация части объекта и запись в конец объекта при помощи запроса типа PUT [16];
- допускается запись в конец объекта без указания смещения (в случаях, когда определение такого смещения является неэффективным) [16];
- имеется возможность примонтировать контейнер bucket к файловой системе при помощи NFS [26] (когда контейнер bucket примонтирован к файловой системе, возможности выполнять операции с объектами, содержащимися в контейнере bucket, с помощью REST интерфейса нет) [16];
- вводится концепция *пространства имен*, которая в какой-то степени является аналогом концепции проекта в Google Cloud Storage, а именно пространства имен изолируют пространства контейнеров bucket друг от друга [16].

EMC ViPR Object Storage гарантирует строгую согласованность данных [35] для всех запросов и устойчивость к разделению. Строгая согласованность достигается с помощью применения специальных техник в зависимости от характера обращения к данным. Тем не менее, в связи с ограничениями налагаемыми теоремой CAP, требования к доступности должны ослабиться. Автор полагает, что требование к доступности может нарушаться в случае, когда объект x располагается в центре хранения и обработки данных DC_1, его копия располагается в центре хранения и обработки данных DC_2, и происходит разрыв связи между DC_1 и DC_2. В этом случае объект x будет доступен на чтение из DC_1 и DC_2, но недоступен на запись (обновление) до тех пор, пока связь между центрами хранения и обработки данных не восстановится.

[6] http://www.emc.com/vipr/

2.3. Протоколы верификации целостности и согласованности данных

В последние годы активно исследуется вопрос «внешней» верификации облачных объектных хранилищ данных, в частности вопрос их соответствия требованиям целостности и согласованности. На сегодняшний день уже существует ряд работ, в которых описываются различные протоколы взаимодействия клиентов с облачными объектными хранилищами данных, позволяющие выявить нарушения в их работе. В этой секции проведен обзор нескольких таких протоколов, а в секции 2.4 будет детально рассмотрен один из них.

Первой работой, в которой была предложена система, гарантирующая согласованность с возможностью разветвления, является система SUNDR [30]. SUNDR – это сетевая файловая система с механизмом защиты, основанным на древовидном хэшировании пользовательских файлов [2]. SUNDR использует «дорогой» протокол, где размер сообщений составляет $\mathcal{O}(n^2)$, где n это количество клиентов в системе [11]. Также как и некоторые другие системы, гарантирующие согласованность с возможностью разветвления, SUNDR имеет некоторые ограничения. К примеру, даже если сервер работает корректно, может случиться ситуация, когда операция клиента блокируется в связи с тем, что другие клиенты параллельно выполняют свои операции [11].

Для того, чтобы предотвратить блокировку операций клиентов, был разработан ряд протоколов (FAUST [9], SPORC [29], Venus [36]), снижающих уровень согласованности данных до слабой согласованности с возможностью разветвления, где согласованность операции устанавливается не сразу, а в конечном счете, когда следующие операции уже могли выполниться. SPORC и родственный с ним протокол Blind Stone Tablet (BST) [38] устроены таким образом, что каждый клиент имеет у себя все данные о состоянии системы, а сервер необходим только для координации клиентов. Такие требования перекрывают все преимущества хранения данных в публичном облачном хранилище. Протоколы BST и COP [10] не требуют блокировки операций клиентов, которые *коммутативны* друг другу.

Протокол VICOS [5] был выпущен в феврале 2015 года. В нем решены многие проблемы, присущие другим протоколам. VICOS гарантирует согласованность с возможностью разветвления. Все *совместимые* операции являются неблокируемыми. Размер сообщений меньше, чем во всех предыдущих протоколах. VICOS гарантирует замедление выполнения операций на константное время, вне зависимости от числа клиентов в системе, тогда как в протоколах FAUST и Venus размер передаваемых сообщений составляет $\mathcal{O}(n)$, где n это количество клиентов в системе, и, соответственно, время замедления операций зависит от количества клиентов в системе. Протокол VICOS гарантирует согласованность с возможностью разветвления для операций чтения, записи, удаления и составления списка объектов, тогда как протокол Venus гарантирует

слабую согласованность с возможностью разветвления только для операций чтения и записи.

Автор считает протокол VICOS более интересным, чем протокол Venus. Но к моменту выхода протокола VICOS работа над реализацией протокола Venus уже активно велась, и оба этих протокола одинаково хорошо решают поставленную в секции 1 задачу.

2.4. Детальный обзор протокола Venus

В этой секции описывается протокол Venus в оригинальном варианте [36]. Протокол, реализованный автором, является модифицированной версией протокола Venus, адаптированной для тестирования облачных объектных хранилищ с помощью генераторов нагрузки. Отличия реализации от оригинальной версии протокола будут описаны в секции 3.1.

2.4.1. Модель системы

Модель системы включает в себя *облачное объектное хранилище данных*, которое не является доверенным компонентом, *верификатор*, который обрабатывает информацию о согласованности данных, и множество *клиентов*. Модель системы представлена на Рис. 1. Протокол не требует внесения каких-либо модификаций на стороне облачного объектного хранилища. Авторы протокола Venus подчеркивают, что верификатор не является доверенным компонентом системы. Клиенты доверяют верификатору в той же степени, что и облачному хранилищу [36]. Это дает возможность разворачивать верификатор в той же облачной системе, в которой находится облачное объектное хранилище. Верификатор также может быть развернут на каком-либо сервере вне облака.

Протокол Venus предполагает, что клиенты могут часто отключаться от сети. Из множества всех клиентов выделяется так называемое *основное подмножество* клиентов, о которых заведомо известно всему множеству клиентов. Клиенты, принадлежащие основному подмножеству клиентов, выполняют специальные функции, а именно: помогают поддерживать согласованность данных и выявлять нарушения в работе системы, а также управляют членством всех клиентов в системе. Клиенты из основного подмножества могут временно отключаться от сети, но протокол гарантирует корректную работу только в том случае, если кворум из клиентов основного подмножества всегда будет в сети. Все клиенты в системе являются доверенными элементами. Допускается только внезапное отключение клиентов от сети.

Предполагается, что облачная система хранения имеет интерфейс, поддерживающий операции чтения и записи объектов: *read(path)* и *write(path, payload)*. После того, как объект был записан на облачное объектное хранилище операцией

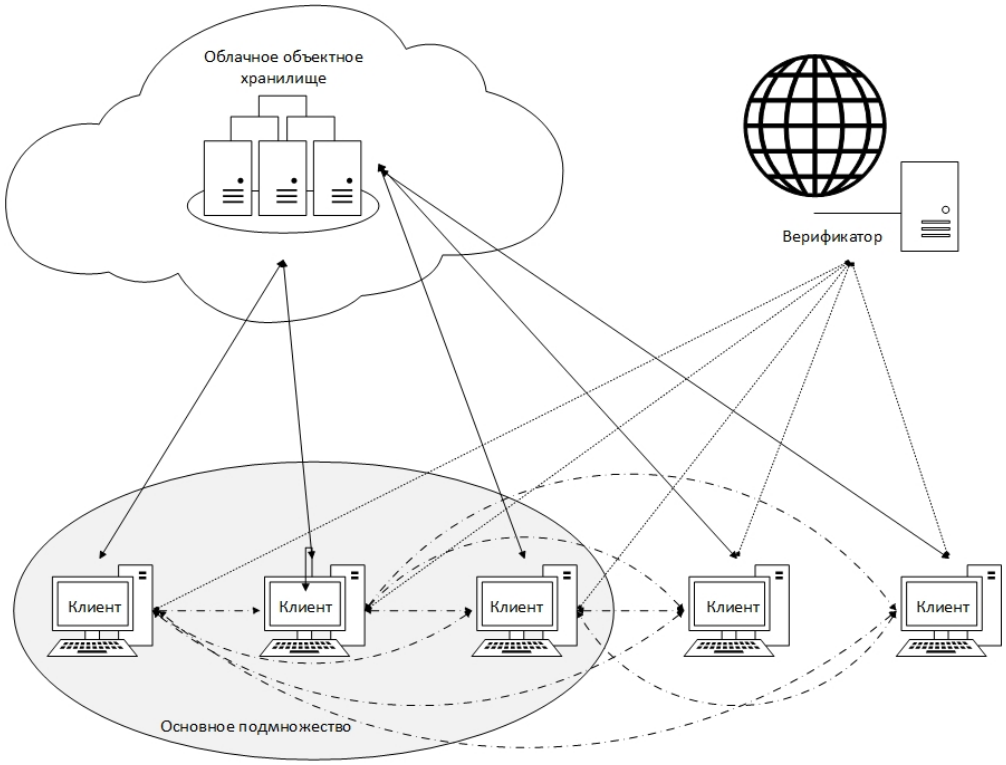


Рис. 1: Модель системы.

write(path, payload), клиенты могут получить объект при помощи операции *read(path)*, по крайней мере через определенный ограниченный промежуток времени. Будем считать, что этот промежуток времени известен заранее, хотя на практике он может быть получен динамически в процессе работы. Если после записи данных через указанный промежуток времени ответ от облачного объектного хранилища не приходит, то поведение системы хранения трактуется как ошибочное.

2.4.2. Интерфейс и семантика

Протокол Venus меняет логику выполнения операций *write(path, payload)* и *read(path)*. В первую очередь, каждой выполненной клиентом операции присваевается своя временная метка, значение которой монотонно возрастает с каждой операцией, выполненной клиентом. Выполнение каждой операции o не зависит от операций, выполняемых другими клиентами. Сразу после выполнения операции ей присваивается *красная* отметка, что значит, что целостность данных была уже проверена, а согласованность еще нет. Тем не менее, для операций помеченных красной отметкой гарантируется причинная согласованность данных. Когда подтверждается, что выполненная операция не нарушила согласованности данных, ей присваивается *зеленая* отметка.

Оригинальный протокол Venus предоставляет асинхронный интерфейс обратных

вызовов, через которые периодически запрашиваются сведения о согласованности данных и об ошибках клиентов. Сведения о согласованности данных включают в себя временную метку последней «зеленой» операции, выполненной клиентом. Все операции, выполненные клиентом до последней «зеленой» операции, также имеют зеленую отметку. Все клиенты видят операции с зеленой отметкой выполненными в одинаковом порядке, то есть протокол Venus гарантирует доступ к так называемой *глобальной последовательности операций*. Эта последовательность также может включать и операции, помеченные красной отметкой. Такое может произойти, к примеру, когда клиент теряет связь в момент совершения операции записи, и «зеленая» операция чтения возвращает записанное значение.

2.4.3. Описание протокола

Протокол Venus поддерживает динамическое присоединение новых клиентов. Тем не менее, в обзоре эта тема рассмотрена не будет, в связи с тем, что эта функция протокола реализована не была. Протокол Venus требует, чтобы были определены следующие параметры:

- R – количество попыток выполнить операцию чтения или записи, прежде чем будет признано, что результат операции неуспешен;
- t_{dummy} – частота фиктивных операций чтения (определение будет дано далее в обзоре);
- t_{send} – время, по прошествии которого клиент запрашивает информацию о последней выполненной другим клиентом операции напрямую, с помощью связи клиент-клиент, если эта информация не была получена клиентом от верификатора;
- $t_{receive}$ – время между последовательными проверками наличия новых сообщений от других клиентов.

Будем различать объекты, которые видят приложения, и объекты, которые создает Venus при записи данных. Объекты, видимые приложением, называются просто *объектами*, а объекты, создаваемые Venus, называются *объектами низкого уровня*. Каждое обновление объекта x приложением влечет создание нового объекта низкого уровня с уникальным идентификатором p_x. Верификатор хранит ссылки на последние записанные объекты низкого уровня, соответствующие каждому объекту x.

Операции чтения и записи. При совершении клиентом операции записи вычисляется h_x – криптографический хэш записываемого объекта x, а так же p_x – идентификатор объекта низкого уровня. Далее клиент посылает запрос на запись $write(p_x, x)$

к облачному объектному хранилищу данных и ждет ответа об успехе операции. Если хранилище данных возвращает позитивный ответ, то клиент посылает верификатору сообщение типа SUBMIT вместе с идентификатором объекта низкого уровня и криптографическим хэшем объекта. После этого клиент ждет от верификатора сообщение типа REPLY. Как только клиент получает это сообщение операция записи считается законченной. Верификатор, в свою очередь, выстраивает все принятые от клиентов сообщения в глобальную последовательность операций $\mathcal{H}$.

При совершении клиентом операции чтения верификатору посылается сообщение типа SUBMIT, для того чтобы получить идентификатор последнего записанного объекта низкого уровня, соответствующего запрашиваемому объекту. Верификатор отвечает сообщением типа REPLY, прикладывая идентификатор объекта низкого уровня p_x и криптографический хэш объекта h_x. Затем клиент посылает облачному хранилищу запрос на чтение $read(p_x)$ и ждет получения объекта x. Как только клиент получил объект x, он проверяет целостность данных, вычисляя криптографический хэш и сравнивая его с полученным от верификатора криптографическим хэшем h_x. Если хэши совпадают, операция чтения считается законченной. В случае если от облачного объектного хранилища приходит ответ, что объекта с идентификатором p_x нет, клиент заново посылает запрос $read(p_x)$ к облачному хранилищу. Если после R повторений запроса на чтение облачное объектное хранилище так и не вернуло объект, то операция чтения считается неуспешной, и клиент оповещает других клиентов об ошибке. Если вычисленный клиентом хэш и хэш h_x, полученный от верификатора, не совпадают, то клиент также оповещает других клиентов об ошибке. Операции записи и чтения представлены на Рис. 2.

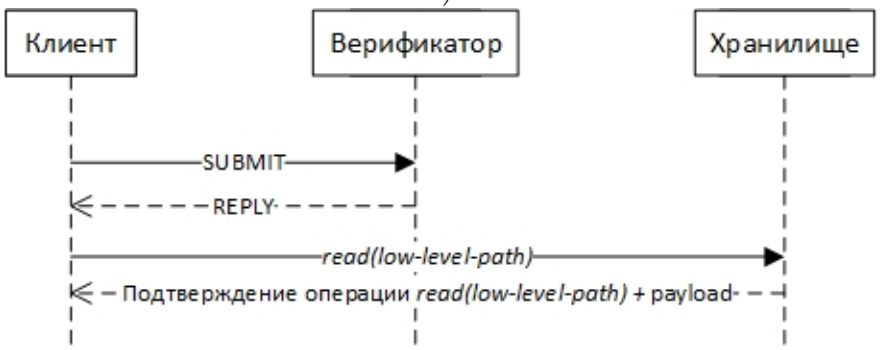
В связи с тем, что верификатор не является доверенным компонентом системы, целостность мета-данных, полученных от верификатора вместе с REPLY сообщением, проверяется. Клиентам приходится подписывать электронной подписью каждое SUBMIT сообщение.

Структура данных version. Необходимо проверять, что верификатор поддерживает корректную глобальную последовательность операций $\mathcal{H}$. Протокол Venus обязывает верификатор присылать *контекст* каждой операции вместе с сообщением типа REPLY. Контекст операции o – это префикс глобальной последовательности операций, выполненных до операции o. Эта информация может быть компактно представлена с помощью *структуры данных version*.

Каждая выполняемая клиентом операция o, имеет свою временную метку $ts(o)$. До того, как операция o завершена, клиент определяет векторные часы $vc(o)$. Элемент $vc(o)[j]$ является временной меткой последней операции, совершенной клиентом C_j в контексте операции o.

Для поддержания согласованности данных информации о временных метках не

a)



b)

Рис. 2: Операции: a) операция записи, b) операция чтения. *low-level-path* – идентификатор объекта низкого уровня, *payload* – объект (последовательность байтов).

достаточно. Обозначим через $vh(o)$ массив, состоящий из криптографических хэшей префиксов глобальной последовательности операций для последних операций, совершенных клиентами системы в контексте операции o. То есть $vh(o)[j]$ является хэшом префикса $\mathcal{H}$ до операции o_j, являющейся последней выполненной клиентом C_j операцией в контексте операции o, выполняемой текущим клиентом. Этот хэш вычисляется итеративным хэшированием всех операций префикса $\mathcal{H}$ с помощью устойчивой к коллизиям криптографической хэш функции. Клиент вычисляет вектор хешей в процессе выполнения операции o из информации, полученной от верификатора вместе с REPLY сообщением.

Структура данных version(o) – это структура, состоящая из векторных часов $vc(o)$ и вектора хэшей $vh(o)$. На множестве структур данных version определяется порядок: $\forall o \forall o' (version(o) \leq version(o')) \Leftrightarrow \forall o \forall o' \forall k (version(o).vc[k] \leq version(o').vc[k]) \&$ $\forall o \forall o' \exists k ((version(o).vc[k] = version(o').vc[k]) \Rightarrow (version(o).vh[k] = version(o').vh[k]))$.

Проверяется, что контекст операции o' включает в себя все операции, находящиеся в контексте операции o, а также что префикс глобальной последовательности операций $\mathcal{H}$ до какой-либо операции o_k в контексте операции o' совпадает с префиксом этой же операции в контексте операции o. Будем говорить, что две структуры данных version *сравнимы*, если одна из них меньше либо равна другой. Существование несравнимых структур данных version свидетельствует об ошибке верификатора.

Операции в деталях. Каждый клиент хранит структуру данных version, соответствующую последней выполненной операции o_{prev}. Более того, если o_{prev} является операцией чтения, то клиент также хранит идентификатор объекта низкого уровня p_{prev} и криптографический хэш h_{prev}, полученные от верификатора. Клиент не знает $version(o)$ текущей операции o в момент посылки SUBMIT сообщения, потому что $version(o)$ вычисляется только после получения REPLY сообщения от верификатора. Поэтому, при выполнении операции o, к SUBMIT сообщению прикладывается структура данных $version(o_{prev})$ последней выполненной клиентом операции.

Если текущая операция o, выполняемая клиентом, является операцией чтения, то к сообщению типа SUBMIT прикрепляется временная метка $ts(o)$ и стурктура данных $version(o_{prev})$. Предположим, что клиент C_i выполняет операцию o. $version(o_{prev}).vh[i]$ определяет префикс глобальной последовательности операций для o_{prev}. Введем обозначение *proof* для электронной подписи от $version(o_{prev}).vh[i]$. Если текущая операция o является операцией записи, то к сообщению типа SUBMIT прикрепляется кортеж $(p_x, h_x, ts(o_{prev}))$. В случае, когда o является операцией чтения, и o_{prev} тоже была операцией чтения, к сообщению типа SUBMIT прикрепляется кортеж $(p_{prev}, h_{prev}, ts(o_{prev}))$. Вся информация, передаваемая в сообщении типа SUBMIT, подписана электронной подписью, за исключением proof, которая сама по себе является электронной подписью.

Верификатор создает глобальную последовательность операций $\mathcal{H}$. Поддерживается массив Ver, где каждый элемент является последней полученной от клиента с соответствующим номером структурой данных version. Верификатор также хранит номер клиента, от которого пришла максимальная структура данных version в переменной c. Иными словами $Ver[c]$ – максимальный элемент массива Ver. Операцию, которой соответствует $Ver[c]$, обозначим o_c. Верификатором поддерживается список $Pending$, содержащий операции из глобальной последовательности операций $\mathcal{H}$, выполненные после o_c, а также массив $Proofs$, содержащий электронные подписи proof, полученные в последних SUBMIT сообщениях от клиентов с соответствующими номерами. Массив $Proofs$ позволяет клиентам проверять согласованность данных с клиентом C_j, вплоть до последней выполненной им операции o_{prev_j}, до того как они примут решение о включении следующей операции клиента C_j в свой контекст.

Верификатор хранит массив $Paths$, содержащий кортежи вида $(p_x, h_x, ts(o))$, относящиеся к последним выполненным клиентами операциям. Кортежи такого вида приходят вместе с сообщениями типа SUBMIT, посланными клиентами при выполнении операции записи, и верификатор обновляет соответствующее значение в массиве $Paths$. При выполнении операции чтения к сообщению типа SUBMIT не прикрепляются идентификатор объекта низкого уровня p_x и криптографический хэш h_x. Соответственно, верификатор в этом случае не обновляет массив $Paths$. Верификатор обрабатывает все сообщения типа SUBMIT атомарно, обновляя необходимые пере-

менные, массивы и списки.

После обработки SUBMIT сообщения, посланного клиентом при выполнении операции записи, верификатор отвечает клиенту сообщением типа REPLY, включающим в себя c, $version(o_c)$, $Pending$ и часть значений массива $Proofs$, соответствующих клиентам, чьи операции находятся в списке $Pending$. Для операций чтения, помимо вышеописанного, к REPLY сообщению прикрепляется кортеж (p_x, h_x, t_x), соответствующий либо последней операции записи в списке $Pending$, либо, если операций записи в $Pending$ нет, кортеж $Paths[c]$ (здесь t_x обозначает временную метку соответствующей операции).

Когда клиент C_i, выполняющий операцию o, в ответ на посланное верификатору SUBMIT сообщение получает сообщение типа REPLY, он проверяет всю подписанную электронными подписями информацию, а затем выполняет следующие проверки:

1. Структура данных $version(o_c)$ больше либо равна структуре данных $version(o_{prev})$, соответствующей последней выполненной клиентом C_i операции.

2. Временная метка последней выполненной клиентом C_i операции o_{prev} равна временной метке $version(o_c).vc[i]$, так как операция o_{prev} должна быть последней операцией в контексте операции o_c.

3. Кортеж (p_x, h_x, t_x) соответствует последней операции записи, находящейся в списке $Pending$, или операции o_c, в случае если операций записи в $Pending$ нет. Это соответствие может быть проверено сравнением временной метки t_x и временной метки соответствующей операции в $Pending$ или $ts(o_c)$, в случае если операций записи в $Pending$ нет.

После этого клиент C_i вычисляет $version(o)$, вызывая функцию, представленную на Рис. 3. Следующие проверки должны выполняться во время обхода списка $Pending$, поэтому они выполняются в процессе вычисления $version(o)$.

4. В списке $Pending$ не может находится более одной операции, посланной клиентом C_j, где $j \neq i$, и не может находится ни одной операции, посланной клиентом C_i.

5. Для каждой операции o', выполненной клиентом C_j, находящейся в $Pending$, временная метка $ts(o')$ равна $version(o_c).vc[j] + 1$.

6. Для каждого клиента C_j, операция которого находится в $Pending$, $Proofs[j]$ является электронной подписью от $version(o_c).vh[j]$.

Если хотя бы одна из вышеописанных проверок провалилась, все клиенты основного подмножества оповещаются об ошибке.

```
1  function compute-version-and-check-pending(o)
2  begin
3      (vc, vh) ← version(o_c);
4      histHash ← vh[c];
5      for q = 1, ..., |Pending| do
6          let client j be the client executing Pending[q];
7          vc[j] ← vc[j] + 1;
8          histHash ← hash(histHash||Pending[q]);
9          vh[j] ← histHash;
10         perform checks 4, 5 and 6 (see the text);
11     end
12     version(o) = (vc, vh);
13     return version(o);
14 end
```

Рис. 3: Функция, вычисляющая $version(o)$. Операция || означает конкатенацию строк.

Согласованность и обнаружение ошибок. Протоколом Venus поддерживается два вида оповещений: оповещения о согласованности, которые показывают, что какие-то операции отмечены как «зеленые», то если их согласованность проверена, и оповещения об ошибках, посылаемые, когда замечена ошибка на стороне облачного хранилища или на стороне верификатора.

Каждый клиент C_i хранит массив $CVer$. Элемент $CVer[j]$ является максимально известной клиенту C_i структурой данных version клиента C_j из основного подмножества клиентов. Значения элементов массива $CVer$ могут быть не актуальными, к примеру клиент C_j уже имеет структуру данных version старше, чем $CVer[j]$, но клиент C_i об этом еще не знает. Вместе с элементами массива $CVer$ клиент хранит локальные временные метки последнего обновления каждого элемента массива.

Каждый раз, когда клиент C_i завершает выполнение операции o, вычисляется $version(o)$ и записывается в $CVer[i]$. Клиент собирает структуры данных version других клиентов, чтобы понять являются ли его собственные операции согласованными. Клиенту необходимо получать информацию о структурах данных version от кворума клиентов из основного подмножества. Обычно, эта информация приходит от верификатора, но также может быть получена с использованием межклиентского взаимодействия (по связи клиент-клиент).

Для того, чтобы получить структуру данных version другого клиента, клиент C_i расширяет каждое сообщение типа SUBMIT подзапросом VERSION-REQUEST, указывая номер k интересующего клиента из основного подмножества. Верификатор, в свою очередь, прикладывает структуру данных version $Version[k]$ к сообщению типа REPLY. Как только клиент C_i получил сообщение типа REPLY, он обновляет элемент массива $CVer[k]$, в случае, если структура данных version из REPLY сооб-

щения больше, чем предыдущее значение $CVer[k]$. Подзапрос VERSION-REQUEST прикрепляется к каждому сообщению типа SUBMIT. Следующий клиент, к которому обращен VERSION-REQUEST, выбирается в соответствии с алгоритмом round-robin [19] по всем клиентам из основного подмножества. В случае, если операции не выполнялись клиентом более, чем t_{dummy} секунд, клиент начинает выполнение *фиктивной* операции чтения. В этом случае, сообщение типа SUBMIT также содержит подзапрос VERSION-REQUEST. Отличие фиктивной операции чтения от операции чтения состоит в том, что при выполнении операции фиктивного чтения, после получения от верификатора сообщения типа REPLY, не происходит обращения к облачному хранилищу. Выполнение операции фиктивного чтения клиентом C_i влечет обновление элемента массива $Version[i]$ на стороне верификатора, также, как это происходит и при обычной операции чтения. Таким образом, клиенты, периодически запрашивающие у верификатора структуру данных version клиента C_i, будут видеть увеличивающуюся последовательность.

Тем не менее, может случиться ситуация, когда клиент C_k отключится от сети, и клиент C_i не получит новую структуру данных version клиента C_k и, соответственно, не обновит значение $CVer[k]$. Более того, если верификатор работает некорректно, новые структуры данных version клиента C_k могут не приходить вместе с REPLY сообщением клиенту C_i. Клиентом C_i оба описанных случая воспринимаются одинаково. В случае, если клиентом C_i не было замечено изменений структуры данных version клиента C_k в течение t_{send} секунд, то по прошествии этого времени клиент C_i запрашивает текущую версию клиента C_k напрямую, по связи клиент-клиент. Когда клиент C_k подключен к системе, каждые $t_{receive}$ секунд он проверяет наличие новых сообщений от других клиентов. Таким образом, если клиент отключился от сети не навсегда, то в конечном счете он получит посланные ему сообщения и выполнит проверку на сравнимость структуры данных version, полученной в сообщении, и максимальной структуры данных version из массива $CVer$. В случае, если ошибок выявлено не было, клиент C_k отвечает клиенту C_i, прикладывая к сообщению максимальную структуру данных version из массива $CVer$, как показано на Рис. 4(а). Заметим, что максимальная структура version массива $CVer$ клиента C_k не обязательно должна соответствовать операции, выполненной клиентом C_i. Все сообщения типа клиент-клиент посылаются в виде email сообщений, подписанных электронной подписью для предотвращения внешних угроз со стороны сети.

Всякий раз когда массив $CVer$ обновляется, клиент C_i выполняет проверку наличия дополнительно появившихся «зеленых» операций. Информация об операциях, которые из «красных» стали «зелеными», может быть получена из массива $CVer$ следующим образом. Для проверки того, что операция o стала «зеленой», клиентом C_i вызывается функция, представленная на Рис. 5. Эта функция вычисляет *множество согласованности* $\mathcal{C}(o)$, состоящее из номеров определенных клиентов основного

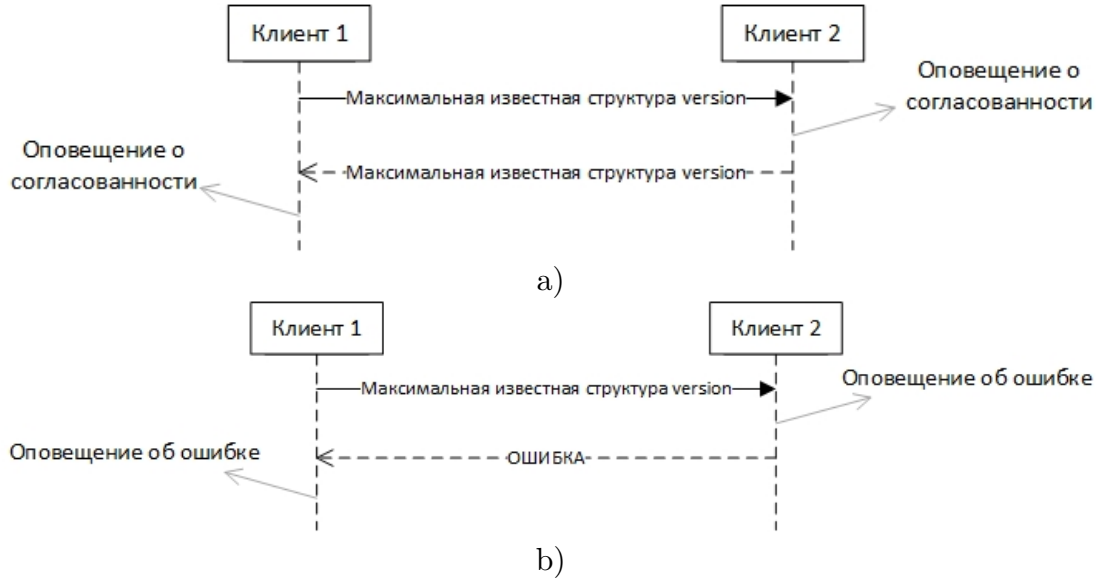


Рис. 4: Проверка согласованности с использованием связи клиент-клиент: a) если проверка согласованности проходит успешно, то посылается ответное сообщение и оповещение о согласованности, b) если проверка согласованности не проходит, то посылается оповещение об ошибке.

подмножества. Если $\mathcal{C}(o)$ содержит кворум клиентов из основного подмножества, то результатом функции будет утверждение о том, что операция o «зеленая». В противном случае, операция o «красная».

```
1  function check-consistency(o)
2  begin
3      C(o) ← ∅;
4      foreach client k in the core set do
5          if CVer[k].vc[i] ≥ ts(o) then
6              add k to C(o);
7          end
8      end
9      if C(o) contains a quorum of the core set then
10         return green;
11     else
12         return red;
13     end
14 end
```

Рис. 5: Функция, выполняющая проверку статуса операции o («красная» или «зеленая»).

Клиент C_i начинает проверку статуса операций с последней «красной» операции, которая не являлась фиктивной. Проверка «красных» операций выполняется в порядке, обратном порядку их выполнения. Проверка выполняется до тех пор, пока какая-нибудь «красная» не фиктивная операция o не будет признана «зеленой». Ес-

ли такая операция o была найдена, клиент C_i признает все операции с временной меткой меньшей либо равной $ts(o)$ «зелеными».

Если на каком-то этапе проверки согласованности клиентом происходит ошибка, клиент посылает всем клиентам основного подмножество оповещение об ошибке. Когда клиент основного подмножества в первый раз получает такое сообщение, он посылает это же оповещение об ошибке всем другим клиентам основного подмножества. Когда клиентом была обнаружена ошибка или клиент получил оповещение об ошибке, он останавливает выполнение всех операций. Клиент основного подмножества, знающий об ошибке, отвечает на все сообщения клиентов сообщением об ошибке, как показано на Рис. 4(b).

Оптимизации и сборка мусора. Скорость работы системы может быть оптимизирована, путем распараллеливания действий при выполении операций чтения и записи.

При выполнении операции чтения, как только было получено сообщение типа REPLY от верификатора, немедленно посылается запрос на чтение к облачному объектному хранилищу. Пока клиент ждет ответа от облачного хранилища, выполняются все необходимые проверки в соответствии с информацией, полученной в REPLY сообщении. Помимо этого клиент подготавливает информацию о текущей выполняемой операции для того, чтобы послать ее верификатору вместе со следующим сообщением типа SUBMIT.

Оптимизировать операцию записи немного сложнее, потому что сообщение типа SUBMIT невозможно послать верификатору раньше, чем будет получен ответ от верификатора, так как оно содержит информацию о текущей операции. Это связано с тем, что если сообщение типа SUBMIT послано верификатору раньше, чем будет получено подтверждение от облачного объектного хранилища, и операция записи завершится неудачно, то у верификатора будет информация об операции, которая на самом деле завершилась провалом. При этом ни одна операция чтения в дальнейшем не сможет завершиться успехом.

Эта проблема решается путем неблокируемого выполнения операций клиентов. А именно, состояния клиентов и верификатора меняются не сразу после выполнения операции записи на облачное объектное хранилище. Поскольку каждый записываемый объект обладает уникальным идентификатором низкого уровня, операция записи происходит без задержек. Оптимизиция операции записи происходит следующим образом. Клиент посылает верификатору сообщение типа DUMMY-SUBMIT, которое, в отличие от SUBMIT сообщения, не содержит никакой информации, а соответственно верификатор безошибочно сможет отличить DUMMY-SUBMIT сообщение от SUBMIT сообщения. После получения DUMMY-SUBMIT сообщения, верификатор посылает в ответ REPLY сообщение, ничем не отличающееся от ответа на сообщения

типа REPLY (содержимое REPLY сообщения для операции записи никак не зависит от содержимого SUBMIT сообщения). После получения REPLY сообщения клиент выполняет все необходимые вычисления и проверки. Когда клиент получает подтверждение об успешном выполнении операции записи от облачного хранилища, он посылает верификатору сообщение типа SUBMIT и ждет ответ. Если содержимое REPLY сообщения не изменилось, результаты поизведенных вычислений могут быть использованы в дальнейшем. В противном случае все проверки и вычисления производятся заново.

Venus создает новый объект с идентификатором низкого уровня каждый раз при выполнении операции записи. Во многих облачных объектных хранилищах данных операции создания объектов и обновления объектов не отличаются. Появляется необходимость в сборщике мусора, удаляющего устаревшие объекты из облачного хранилища. Каждый клиент периодически производит удаление своих устаревших объектов с идентификаторами низкого уровня.

3. Реализация протокола Venus

Главное достоинство представляемой реализации – это разделение логики работы по протоколу S3 и по протоколу Venus. Такой подход позволяет использовать существующие генераторы нагрузки (S3 приложения), не обладающие встроенными функциями верификации согласованности данных, для проверки выполнения облачным объектным хранилищем требований к целостности и согласованности. Наиболее вероятным сценарием тестирования облачного объектного хранилища с использованием реализации протокола Venus является запуск нескольких генераторов нагрузки, работающих с одним набором данных, где предлагаемая реализация протокола Venus является промежуточным слоем между генераторами нагрузки и облачным объектным хранилищем, как показано на Рис. 6.

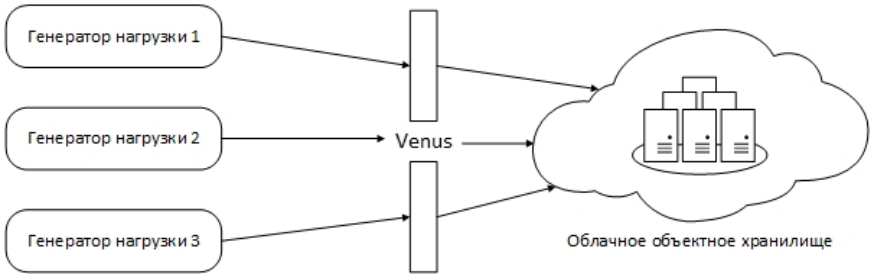


Рис. 6: Наиболее вероятный сценарий тестирования облачного объектного хранилища данных на выполнение требований к целостности и согласованности данных.

Благодаря тому, что протокол Venus замедляет взаимодействие с облачным объектным хранилищем лишь на небольшую константу (что будет показано в секции 3.7), появляется возможность выполнять тестирование облачного объектного хранилища на нагрузку, получая при этом информацию о том, выполняются ли операции согласованно или нет, при этом практически не меняя скорость выполнения операций чтения и записи.

Имеется возможность включать и отключать обработку запросов в соответствии с протоколом Venus прямо в процессе тестирования. Если было зафиксировано нарушение согласованности данных, то Venus автоматически перезапускается и начинает заново верифицировать корректность работы облачного хранилища.

3.1. Модификация протокола для целей тестирования

Оригинальный протокол Venus поддерживает временное отключение клиентов от сети [36]. Когда клиент снова подключается, он обрабатывает информацию, поступившую ему с помощью автоматизированных электронных сообщений от других клиентов в период его отсутствия. Такой подход является разумным, если клиентами являются обычные пользователи или программы, работающие с важной информацией. В случае с генераторами нагрузки степень уверенности, что клиент не отключится

от сети, достаточно велика. Ведь, как правило, генераторы нагрузки запускаются на серверах локальной сети компании, или вообще запускаются несколько экземпляров генераторов нагрузки на одном сервере. При такой конфигурации вероятность потери связи с облачным хранилищем для всех клиентов одновременно намного больше, чем вероятность потери связи с одним из клиентов. Даже если происходит отключение от системы одного из клиентов (генератора нагрузки), то при подключении его обратно можно просто перезапустить слой Venus, что никак не отразится на работе генераторов нагрузки, которые от системы не отключались. Поэтому в случае тестирования облачных объектных хранилищ данных генераторами нагрузки, разумно предположить, что клиенты не отключаются от сети, а соответственно получают все сообщения, посланные другими клиентами.

Из предположения о том, что клиенты системы не могут отключаться от сети, следует, что все клиенты принадлежат основному подмножеству клиентов системы. А значит можно отказаться от концепции основного подмножества и определить одинаковое поведение для всех клиентов системы. Помимо этого, появляется возможность заменить способ общения между клиентами на более быстрый. В предлагаемой реализации рассылка оповещений о согласованности и оповещений об ошибках осуществляется с помощью *широковещания* (*all-to-all broadcast*). Этот метод передачи данных схематично представлен на Рис. 7. Обмен сообщениями между двумя клиентами осуществляется по протоколу *TCP* [24].

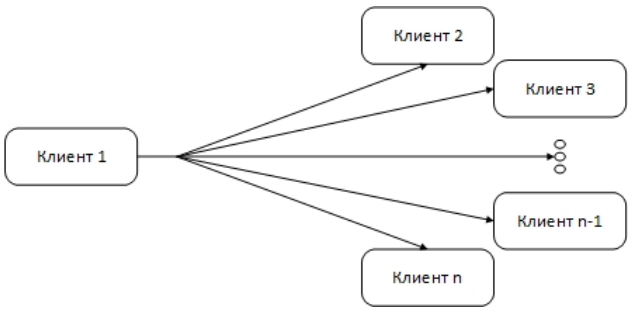


Рис. 7: Широковещание – метод передачи данных от одного клиента всем клиентам системы.

Функция динамического присоединения клиентов к системе была убрана. При тестировании облачных объектных хранилищ данных необходимости динамически добавлять еще один генератор нагрузки нет. Знание точного количества клиентов системы позволяет определить максимальные размеры всех сообщений в системе и точнее спрогнозировать время задержки выполнения операций. Максимальный размер сообщений, в частности SUBMIT и REPLY сообщений, зависит только от количества подключенных клиентов [36].

3.2. Представление Venus, как отдельного слоя

В предлагаемом методе тестирования облачных объектных хранилищ данных на выполнение требований целостности и согласованности генераторы нагрузки (создающие потоки запросов на чтение и запись данных) отделены от реализации протокола Venus. Такое разделение возможно при использовании концепции прокси-сервера [31].

Большинство известных S3 приложений имеют функцию работы через прокси-сервер. В частности, комплект средств разработки AWS SDK[7] компании Amazon включает в себя эту возможность. Для работы вместе с реализацией протокола Venus, генератор нагрузки обязательно должен поддерживать взаимодействие с облачным объектным хранилищем данных через прокси-сервер. Помимо этого, генератор нагрузки должен уметь вычислять хэш передаваемого (принимаемого) объекта. В реализации протокола Venus, при внесении небольших изменений, хэш передаваемого (принимаемого) объекта тоже может вычисляться, но свойство протокола Venus замедлять взаимодействие генератора нагрузки и облачного хранилища на небольшую константу пропадет и время выполнения операции будет зависеть от размера передаваемого (принимаемого) объекта. В комплект средств разработки AWS SDK функция проверки хэша объекта включена по умолчанию.

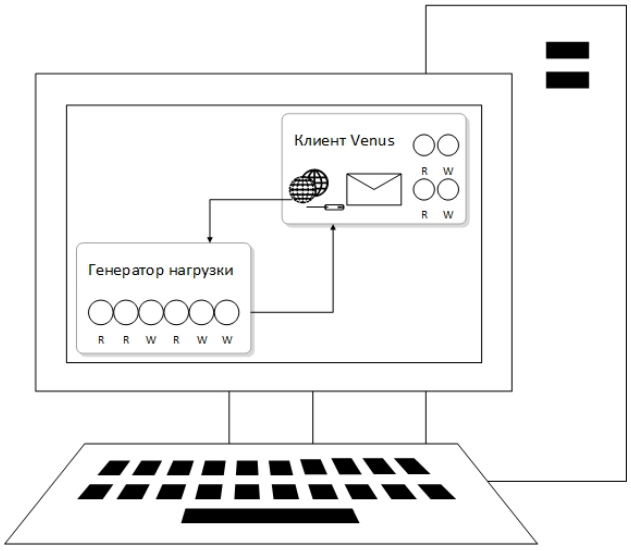


Рис. 8: Сервер, с запущенным на нем генератором нагрузки на облачное объектное хранилище данных и клиентом Venus.

Для того, чтобы отличать сущность клиента оригинального протокола Venus от реализованной сущности клиента, введем понятие *клиента Venus*. Можно сказать, что генератор нагрузки и клиент Venus соответствуют понятию клиента оригинального протокола. Под *слоем Venus* понимается реализация клиента Venus и верификатора. Клиент Venus запускается на том же сервере, что и генератор нагрузки на облачное

[7]https://aws.amazon.com/tools/

хранилище. Клиент Venus включает в себя функцию прокси-сервера. Прокси-сервером перехватываются все запросы, поступающие от генератора нагрузки. Обработка запросов на чтение и запись происходит в соответствии с оригинальным протоколом Venus. Все остальные запросы не изменяются. Клиент Venus реализует всю логику работы клиента, описываемую в оригинальном протоколе, за исключением ограничений, описанных в секции 3.1. Сервер с запущенным на нем генератором нагрузки на облачное объектное хранилище данных и клиентом Venus представлен на Рис. 8.

3.3. Модель системы

Модель реализованной системы, с учетом вышеуказанных модификацией, представлена на Рис. 9. В отличие от оригинальной модели системы, представленной на Рис. 1, все клиенты выполняют одинаковые функции (выполнен отказ от концепции основного подмножества). Более того, в оригинальной модели клиент изначально работает в соответствии с протоколом Venus, то есть умеет формировать запросы по протоколу S3 к облачному объектному хранилищу, а также должным образом обмениваться сообщениями с верификатором и другими клиентами. В реализованной модели системы запросы по потоколу S3 формирует генератор нагрузки, а клиент Venus занимается только обменом сообщениями между членами системы и модификацией отдельных заголовков перехваченных от генератора нагрузки запросов.

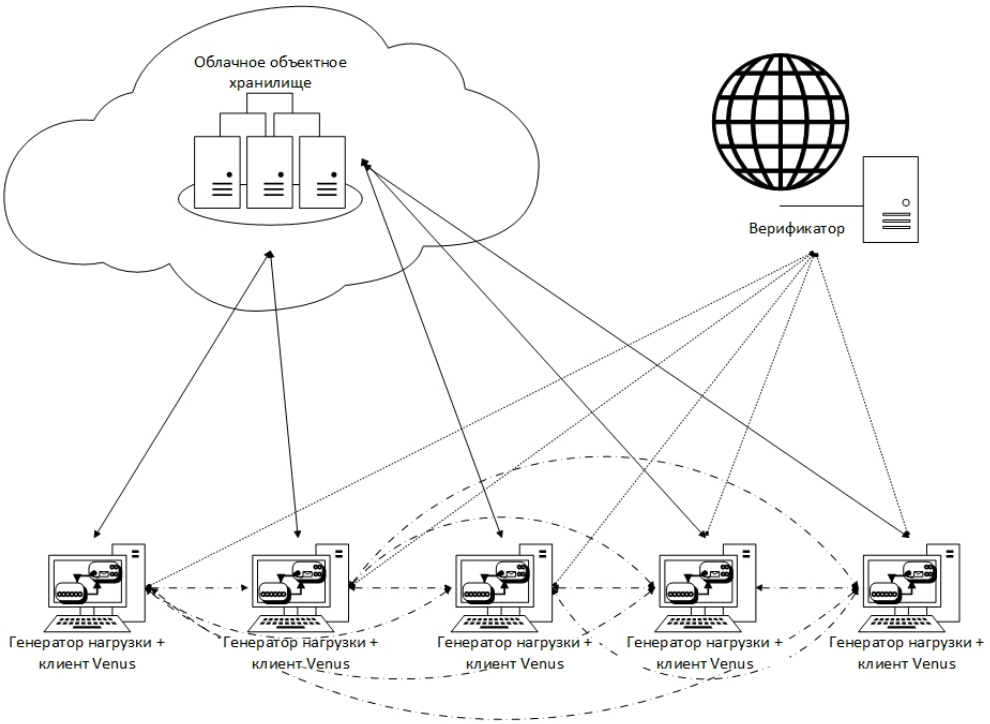


Рис. 9: Реализованная модель системы.

3.4. Технические особенности реализации

3.4.1. Подмена идентификатора объекта

Каждый запрос генератора нагрузки перехватывается прокси-сервером, являющимся частью реализации клиента Venus. Протокол S3 является REST интерфейсом для доступа к облачному объектному хранилищу (например, к Amazon Simple Storage Service). При использовании REST интерфейса любой запрос к облачному хранилищу представляется в виде HTTP запроса [25]. Для контроля доступа к облачному объектному хранилищу добавляются несколько дополнительных заголовков, в частности заголовоки *Authorization* и *Content-MD5* [1]. Запрос на чтение данных является GET запросом, а запрос на запись данных PUT запросом. Эти типы запросов могут использоваться и для работы с контейнером для объектов, называемым *bucket*. Прокси-сервер определяет, относится ли запрос к bucket или к объекту с помощью регулярных выражений, применяемых к URL локатору запроса [4].

Каждая операция создания и обновления объекта влечет за собой создание нового объекта с уникальным идентификатором низкого уровня на облачной объектной системе хранения [36]. Если прокси-сервером клиента Venus был перехвачен PUT запрос, относящийся к объекту, то клиент вычисляет уникальный идентификатор объекта низкого уровня, исходя из идентификатора клиента в системе, идентификатора объекта, MD5 хэша объекта [28]. Предположим, что идентификатор клиента в системе *id*, идентификатор объекта *path* и MD5 хэш объекта *hash*. Тогда идентификатор низкого уровня будет иметь вид *id-path-hash*. Прокси-сервер получает идентификатор объекта и хэш MD5 из перехваченного HTTP запроса.

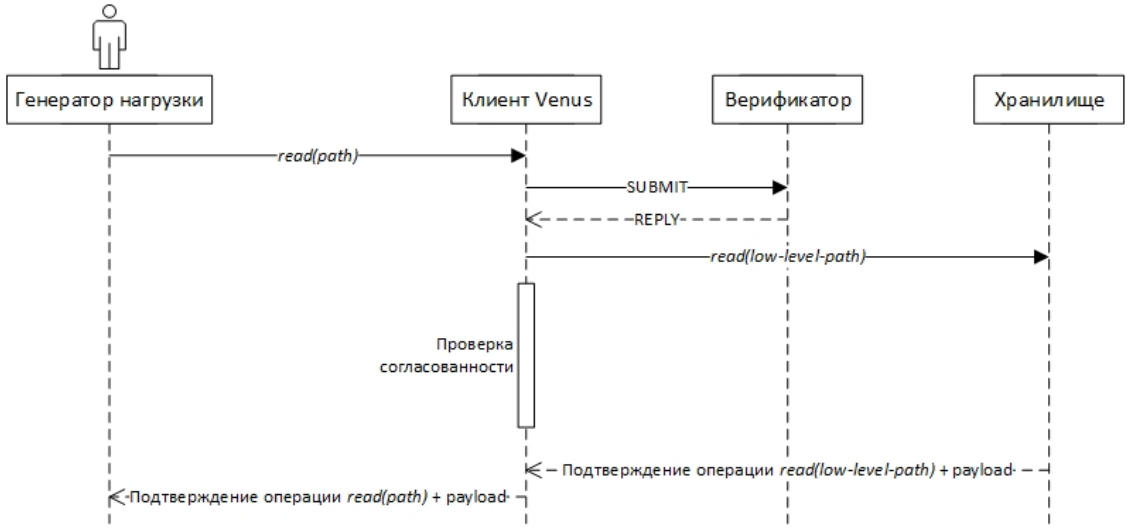
В перехваченном HTTP запросе заменяется идентификатор объекта на идентификатор объекта низкого уровня. Если отослать такой видоизмененный запрос к облачному объектному хранилищу, то вернется ошибка, потому что заголовок HTTP запроса Authorization, являющийся электронной подписью, используемой облачным объектным хранилищем для верификации целостности мета-данных, передаваемых в HTTP запросе, был посчитан от других мета-данных, когда HTTP запрос еще содержал старый идентификатор объекта. Поэтому заголовок Authorization пересчитавается от новых мета-данных запроса в соответствии с процедурой, описанной в [1], и только после этого отсылается на облачное объектное хранилище.

Если прокси-сервером клиента Venus был перехвачен GET запрос, относящийся к объекту, то клиент Venus получает идентификатор объекта низкого уровня из REPLY сообщения от верификатора. Затем, в HTTP запросе необходимо заменить идентификатор запрашиваемого объекта на идентификатор объекта низкого уровня, а соответственно и обновить заголовок Authorization, позволяющий облачному объектному хранилищу удостовериться в целостности передаваемых в HTTP запросе мета-данных. В этом случае алгоритм расчета электронной подписи другой и он

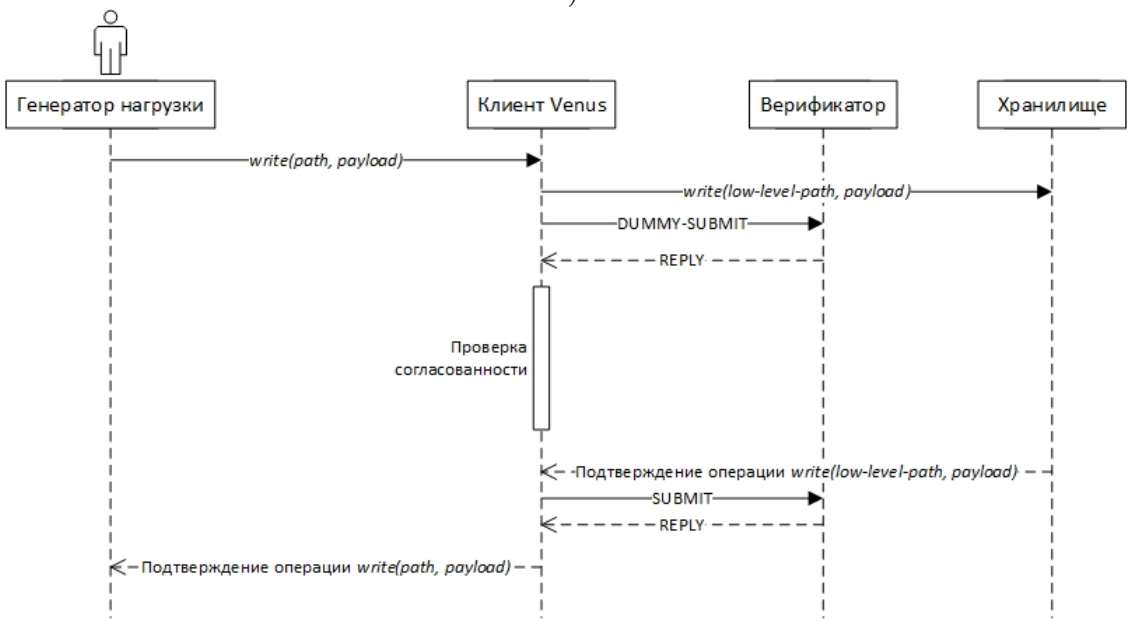
также описан в [1].

3.4.2. Операции чтения и записи

Операции чтения и записи были реализованы с учетом оптимизаций, предлагаемых авторами оригинального протокола Venus. Отличие заключается в том, что добавился дополнительньный обмен сообщениями между генератором нагрузки и клиентом Venus. Несмотря на дополнительный обмен сообщениями общее время выполнения операций чтения и записи практически не изменилось, так как генератор нагрузки и клиент Venus запускаются на одном и том же сервере. Реализованные операции чтения и записи представлены на Рис. 10.



a)



b)

Рис. 10: Операции: a) оптимизированная операция чтения, b) оптимизированная операция записи.

3.4.3. Реализация клиентской части и верификатора

Клиент Venus и верификатор являются многопоточными приложениями. Клиент Venus включает в себя поток, перехватывающий запросы от генератора нагрузки и от облачного хранилища, поток, осуществляющий обмен сообщениями между клиентом Venus и верификатором, и поток, осуществляющий взаимодействие между клиентами. Взаимодействие между потоками осуществляется путем передачи сообщений через синхронные очереди.

Связь между клиентами Venus и верификатором осуществляется с помощью сетевых сокетов [33]. Функцию транспортного уровня выполняет TCP [24]. Верификатор обрабатывает входящие соединения асинхронно. Когда сообщение от клиента получено, в отдельном потоке запускается обработка этого сообщения. Поскольку количество клиентов известно еще до начала тестирования, число потоков верификатора, обрабатывающих принятые в REPLY сообщениях данные, строго определено и равно числу клиентов в системе.

3.4.4. Журналирование

При отладке многопоточных приложений важную роль играет журналирование. В предлагаемой реализации роль журналирования очень велика, так как только с помощью сообщений в журнале можно удостовериться, что реализация действительно работает в соответствии с модифицированным протоколом Venus.

Используется несколько уровней журналирования. Пользователю реализации протокола Venus будут видны только сообщения уровня INFO и выше, такие как оповещения об ошибках, оповещения о согласованности, критические ошибки в работе программы и другие. Разработчику будет удобно использовать уровень журналирования DEBUG, чтобы отследить всю необходимую информацию о работе системы, возникающие исключения, посмотреть мета-данные, передаваемые в сообщениях и прочее. Для отладки критических мест удобно использовать уровень журналирования TRACE. В каждом методе, критическом для корректной работы протокола, для уровня TRACE сообщения о вызове метода и возврата из метода записываются в журнал. Подробнее о концепции уровней журналирования можно узнать в [12].

3.5. Генератор нагрузки, используемый для тестирования реализации

Для тестирования реализации протокола был написан свой небольшой генератор нагрузки на облачное объектное хранилище данных по протоколу S3. Поддерживается набор готовых сценариев нагрузки на облачное хранилище для создания нагрузки разной степени интенсивности. Набор сценариев легко может быть расширен. Имеется

два режима работы – напрямую с облачным объектным хранилищем и через прокси-сервер.

Генератор нагрузки может быть вызван из командной строки с указанием названия сценария, который необходимо выполнить. В качестве примера можно привести сценарий, который принимает на вход время выполнения сценария и соотношение запросов на чтение и запись. Далее, в течение указанного времени, генерируются запросы на чтение и запись в соответствии с указанной пропорцией.

3.6. Технологии, используемые в реализации

Реализация протокола Venus выполнена на языке Java[8]. Выбор пал в пользу этого языка связи с тем, что огромное количество корпоративных приложений написано на языке Java. Имеется много библиотек, позволяющих не «изобретать велосипед», а сразу использовать готовые решения.

Выбор языка Java связан еще и с тем, что большинство известных автору генераторов нагрузки на облачное объектное хранилище написаны на языке Java (в частности, генератор нагрузки COSBench). Реализация клиента Venus выполнена таким образом, что логика протокола Venus и логика передачи данных (обмена сообщениями) отделены друг от друга. При внесении небольшого количества модификаций можно превратить исходный код в библиотеку и интегрировать ее в один из существующих генераторов нагрузки.

Прокси-сервер клиента Venus реализован с помощью библиотеки LittleProxy[9]. Она предоставляет удобный интерфейс для определения действий, совершаемых при поступлении различных видов HTTP запросов. Механизм сериализации данных Protocol Buffers[10], созданный компанией Google, был использован для формирования всех сообщений системы. В качестве библиотеки журналирования была выбрана Logback[11] и фасад для систем журналирования SLF4J[12]. Для сборки проекта используется система автоматической сборки Gradle[13].

Генератор нагрузки, используемый для тестирования реализации протокола Venus, также написан на языке Java. Запросы к облачному хранилищу формируются при помощи комплекта средств разработки AWS SDK for Java[14].

[8] http://docs.oracle.com/javase/
[9] https://github.com/adamfisk/LittleProxy/
[10] https://developers.google.com/protocol-buffers/
[11] http://logback.qos.ch/
[12] http://www.slf4j.org/
[13] https://gradle.org/
[14] http://aws.amazon.com/sdk-for-java/

3.7. Оценка производительности

Существует несколько возможных вариантов развертывания верификатора: в той же самой облачной системе, в которой находится хранилище данных, и вне облачной системы. Возможность разворачивать верификатор в облачной системе имеется в связи с тем, что верификатор не является доверенным компонентом системы [36]. Вне облачной системы верификатор может быть развернут разными способами: на удаленном сервере (в этом случае скорость выполнения операций будет схожей со скоростью выполнения операций, когда верификатор развернут в облачной системе) и на сервере в локальной сети.

Оценка производительности реализации протокола Venus была выполнена для следующих случаев:

- пять клиентов одновременно работают с одним и тем же набором данных по протоколу S3 без промежуточного слоя Venus (все клиенты находятся в одной локальной сети);
- пять клиентов одновременно работают с одним и тем же набором данных по протоколу Venus (все клиенты и верификатор находятся в одной локальной сети);
- пять клиентов одновременно работают с одним и тем же набором данных по протоколу Venus (все клиенты находятся в одной локальной сети, верификатор развернут в облачной системе).

Тестировалось облачное объектное хранилище Amazon Simple Storage Service с точкой доступа в регионе EU (Ireland). Сервис Amazon Elastic Compute Cloud[15] использовался для разворачивания верификатора в облачной системе. Измерения в каждом случае проводились для файлов размером 10 KB и 1 MB. Каждый из пяти клиентов, участвующих в тестировании, выполнил по 25 операций записи и по 25 операций чтения для каждого варианта развертывания верификатора. Результаты измерений представлены на Рис. 11.

Измерения показали, что увеличение времени выполнения операций не зависит от размера передаваемых данных, а зависит только от того, где развернут верификатор. Средняя задержка выполнения операций в случае развертывания верификатора в локальной сети составила 5.5 миллисекунд. Средняя задержка выполнения операций в случае развертывания верификатора в облачной системе составила 258 миллисекунд. Авторы оригинального протокола Venus [36] утверждают, что их реализация замедляет выполнение каждой операции на 450 миллисекунд для случая, когда верификатор развернут в облачной системе. К сожалению, невозможно утверждать выигрывает

[15] http://aws.amazon.com/ec2/

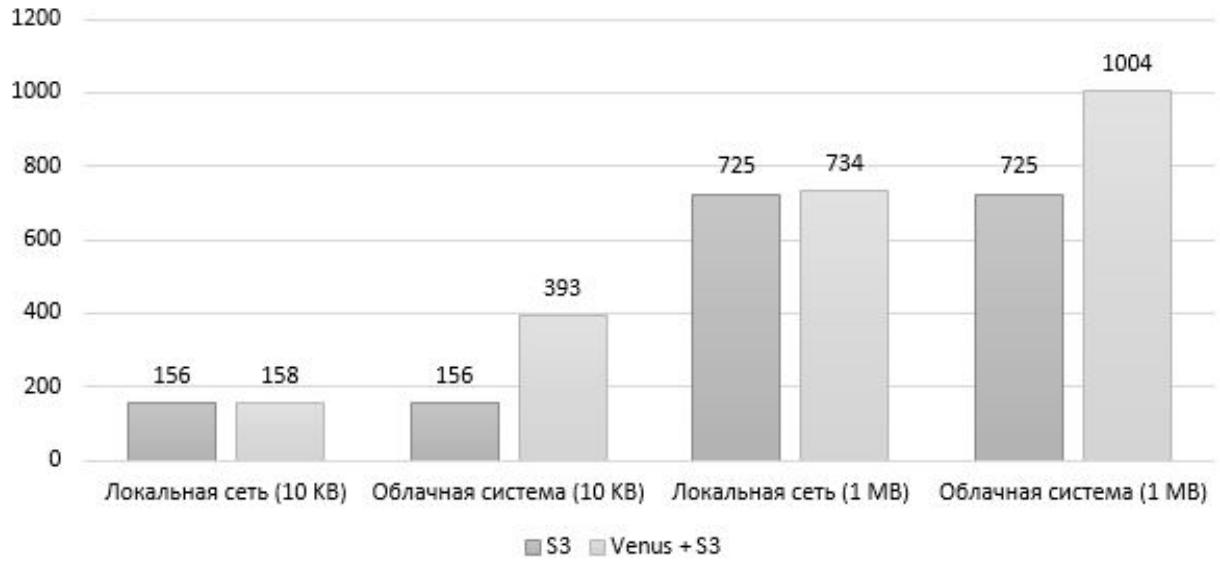


Рис. 11: Гистограмма, демонстрирующая зависимость времени выполнения операции от размера передаваемых данных и расположения верификатора. Время указано в миллисекундах.

ли предлагаемая реализация протокола Venus по производительности относительно реализации авторов оригинального протокола Venus, так как для сравнения необходимо выполнить замеры времени для обеих реализаций в одинаковых условиях, что невозможно, так как исходного кода реализации авторов оригинального протокола Venus нет в свободном доступе. Полученные результаты измерений позволяют лишь утверждать, что увеличение времени выполнения каждой операции происходит лишь на константу, не зависящую от размера передаваемых данных, а зависящую только от расположения верификатора. Также результаты измерений доказывают практическую пригодность предлагаемой реализации протокола Venus для тестирования облачных объектных хранилищ на выполнение требований целостности и согласованности данных с точки зрения производительности, в особенности для случая, когда верификатор развернут в локальной сети, где увеличение времени выполнения каждой операции пренебрежимо мало.

Заключение

В ходе работы были достигнуты следующие результаты:

- Проанализированы проблемы, возникающие при разработке облачных объектных хранилищ, и протоколы верификации целостности и согласованности данных.
- В протокол Venus внесены модификации:
  - заменен способ общения между клиентами на широковещание;
  - введено ограничение на количество клиентов, которое позволяет точно спрогнозировать время задержки выполнения операций.
- Реализован прототип, включающий в себя предложенные модификации, и адаптированный для тестирования облачных объектных хранилищ.
- Прототип протестирован на корректность работы и соответствие требованиям. Выполнены замеры производительности.

В качестве продолжения работы можно предложить реализацию альтернативных методов межклиентского взаимодействия, а также исследование возможности восстановления корректной работы системы после сбоя.

Список литературы

[1] Amazon Simple Storage Service Developer Guide. — API Version 2006-03-01.

[2] Ideal Hash Trees : Rep. ; Executor: Phil Bagwell : 2001.

[3] Baldoni Roberto, Raynal Michel. Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems // IEEE Distributed Systems Online. — 2002. — Vol. 3, no. 2. — P. 12.

[4] Berners-Lee Tim, Masinter Larry, McCahill Mark. RFC 1738: Uniform Resource Locator // Internet Engineering Task Force. — 1994.

[5] Brandenburger Marcus, Cachin Christian, Knežević Nikola. Don't Trust the Cloud, Verify: Integrity and Consistency for Cloud Object Stores // arXiv preprint arXiv:1502.04496. — 2015.

[6] Brewer Eric A. Towards Robust Distributed Systems // PODC. — Vol. 7. — 2000.

[7] The CAP Theorem. — URL: `https://foundationdb.com/key-value-store/white-papers/the-cap-theorem` (online; accessed: 28.05.2015).

[8] COSBench: A Benchmark Tool for Cloud Object Storage Services / Qing Zheng, Haopeng Chen, Yaguang Wang et al. // Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on Cloud Computing / IEEE. — 2012. — P. 998–999.

[9] Cachin Christian, Keidar Idit, Shraer Alexander. Fail-Aware Untrusted Storage // SIAM Journal on Computing. — 2011. — Vol. 40, no. 2. — P. 493–533.

[10] Cachin Christian, Ohrimenko Olga. Verifying the Consistency of Remote Untrusted Services with Commutative Operations // Principles of Distributed Systems. — Springer, 2014. — P. 1–16.

[11] Cachin Christian, Shelat Abhi, Shraer Alexander. Efficient Fork-Linearizable Access to Untrusted Shared Memory // Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing / ACM. — 2007. — P. 129–138.

[12] Ceki Gulcu, Sebastien Pennec, Carl Harris. Logback's Architecture. — URL: `http://logback.qos.ch/manual/architecture.html` (online; accessed: 16.05.2015).

[13] Dean Jeffrey, Ghemawat Sanjay. MapReduce: Simplified Data Processing on Large Clusters // Communications of the ACM. — 2008. — Vol. 51, no. 1. — P. 107–113.

[14] EMC Education Services. Information Storage and Management: Storing, Managing and Protecting Digital Information in Classic, Virtualized and Cloud Environments /

Ed. by Somasundaram Gnanasundaram, Alok Shrivastava. — Second edition. — John Wiley & Sons, Inc., 2012. — P. 528.

[15] Fielding Roy Thomas. Architectural Styles and the Design of Network-Based Software Architectures : Ph. D. thesis / Roy Thomas Fielding ; University of California, Irvine. — 2000.

[16] Getting Started with ViPR - Java + S3 (ViPR SDK). — URL: `https://community.emc.com/docs/DOC-27907` (online; accessed: 27.05.2015).

[17] Gilbert Seth, Lynch Nancy. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services // ACM SIGACT News. — 2002. — Vol. 33, no. 2. — P. 51–59.

[18] Google Cloud Storage. Concepts and Techniques. — URL: `https://cloud.google.com/storage/docs/concepts-techniques` (online; accessed: 26.05.2015).

[19] H. Remzi, C. Andrea. Operating Systems: Three Easy Pieces. — Arpaci-Dusseau Books, 2014.

[20] Lamport Leslie. Time, Clocks and the Ordering of Events in a Distributed System // Communications of the ACM. — 1978. — Vol. 21, no. 7. — P. 558–565.

[21] Mazieres David, Shasha Dennis. Building Secure File Systems Out of Byzantine Storage // Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing / ACM. — 2002. — P. 108–117.

[22] Mesnier Mike, Ganger Gregory R, Riedel Erik. Object-Based Storage // Communications Magazine, IEEE. — 2003. — Vol. 41, no. 8. — P. 84–90.

[23] Özsu M Tamer, Valduriez Patrick. Principles of Distributed Database Systems. — Springer Science & Business Media, 2011.

[24] Postel Jon. RFC 793: Transmission Control Protocol, September 1981 // Status: Standard. — 2003. — Vol. 88.

[25] RFC 2616: Hypertext Transfer Protocol – HTTP/1.1 / Roy Fielding, J Gettys, J Mogul et al. // The Internet Engineering Task Force. — 1999.

[26] RFC 3530: Network File System (NFS) version 4 Protocol / S Shepler, B Callaghan, D Robinson et al. // IETF, April. — 2003.

[27] Read-After-Write Consistency in Amazon S3. — URL: `http://shlomoswidler.com/2009/12/read-after-write-consistency-in-amazon.html` (online; accessed: 28.05.2015).

[28] Rivest Ronald L et al. RFC 1321: The MD5 Message-Digest Algorithm // Internet activities board. — 1992. — Vol. 143.

[29] SPORC: Group Collaboration Using Untrusted Cloud Resources. / Ariel J Feldman, William P Zeller, Michael J Freedman, Edward W Felten // OSDI. — Vol. 10. — 2010. — P. 337–350.

[30] Secure Untrusted Data Repository (SUNDR) / Jinyuan Li, Maxwell N Krohn, David Mazières, Dennis Shasha // OSDI. — Vol. 4. — 2004. — P. 9–9.

[31] Shapiro Marc. Structure and Encapsulation in Distributed Systems: the Proxy Principle // ICDCS. — 1986. — P. 198–204.

[32] Software Defined Storage / Mark Carlson, Alan Yoder, Leah Schoeb et al. // SNIA. — 2015. — P. 11.

[33] Stevens W Richard, Fenner Bill, Rudoff Andrew M. UNIX Network Programming. — Addison-Wesley Professional, 2004. — Vol. 1.

[34] Tanenbaum Andrew, Van Steen Maarten. Distributed Systems. — Pearson Prentice Hall, 2007.

[35] VIPR 2.1 - EMC VIPR Data Services: Geo-Protection and Multisite Access. — URL: `http://www.emc.com/techpubs/vipr/geo_overview-2.htm` (online; accessed: 27.05.2015).

[36] Venus: Verification for Untrusted Cloud Storage / Alexander Shraer, Christian Cachin, Asaf Cidon et al. // Proceedings of the 2010 ACM workshop on Cloud computing security workshop / ACM. — 2010. — P. 19–30.

[37] Vogels Werner. Eventually Consistent // Communications of the ACM. — 2009. — Vol. 52, no. 1. — P. 40–44.

[38] Williams Peter, Sion Radu, Shasha Dennis. The Blind Stone Tablet: Outsourcing Durability to Untrusted Parties // NDSS / Citeseer. — 2009.