Факультет вычислительной математики и кибернетики
Московского государственного университета
имени М. В. Ломоносова

И. М. Никольский

РАСПРЕДЕЛЕННАЯ ОБРАБОТКА ДАННЫХ

Учебно-методическое пособие

Издательство Московского университета

Факультет вычислительной математики и кибернетики
Московского государственного университета
имени М. В. Ломоносова

И. М. Никольский

РАСПРЕДЕЛЕННАЯ ОБРАБОТКА ДАННЫХ

Учебно-методическое пособие

Издательство Московского университета • 2023

УДК 004.75(075.8)
ББК 16.262я73
Н64

Печатается по решению Редакционно-издательского совета факультета вычислительной математики и кибернетики МГУ имени М. В. Ломоносова

Рецензенты:

К. А. Жуков — канд. физ.-мат. наук,
сотрудник кафедры суперкомпьютеров и квантовой информатики
факультета вычислительной математики и кибернетики
МГУ имени М. В. Ломоносова

А. Н. Сальников — канд. физ.-мат. наук, ведущий научный сотрудник
кафедры автоматизации систем вычислительных комплексов
факультета вычислительной математики и кибернетики
МГУ имени М. В. Ломоносова

Никольский, И. М.

Н64 Распределенная обработка данных: учебно-методическое пособие. — Москва : Издательство Московского университета, 2023. — 28, [1] с. — Электронное издание сетевого распространения. — (Библиотека факультета ВМК МГУ).

ISBN 978-5-19-011913-8 (e-book)

Пособие посвящено обработке больших объёмов данных, хранимых распределённо на узлах вычислительной системы. Рассмотрены основные подходы к репликации и секционированию данных. Большое внимание уделяется методам синхронизации компонент распределённых систем, изложены основы технологии блокчейн, а также основная технология распределённой обработки данных MapReduce.

Данное пособие предназначено для студентов бакалавриата, обучающихся по направлению 01.03.02. «Прикладная математика и информатика», а также может использоваться в качестве вводного курса для тех, кто планирует специализироваться в сфере аналитики данных, инженерии данных и смежных областях.

УДК 004.75(075.8)
ББК 16.262я73

ISBN 978-5-19-011913-8 (e-book)

Оглавление

1. Предисловие

Индустрия больших данных — это одна из самых быстро развивающихся областей информационных технологий. Анализ больших массивов информации позволяет находить закономерности в данных научных экспериментов, предсказывать отток клиентов компаний, выявлять факты мошенничества и решать многие другие важные задачи. Профессии, связанные с Big Data, привлекательны для выпускников высших учебных заведений.

Однако специализация в сфере работы с данными требует специфических знаний и навыков. И в первую очередь это связано с необходимостью использования распределённых вычислительных систем. В самом деле, современные массивы данных обладают большим объёмом и большой скоростью генерации, что делает невозможным их корректную обработку с помощью отдельного сервера. Требования отказоустойчивости также не позволяют использовать локальную систему — необходимо дублирование и данных, и аппаратной составляющей.

Инженерам, занимающимся построением и развитием инфраструктуры данных, приходится работать с обширным набором специальных программных инструментов. В их число входят аналитические базы данных (GreenPlum, Vertica), фреймворки распределённой обработки данных (Hadoop, Spark), брокеры сообщений (RabbitMQ, Kafka), инструменты трансформации и загрузки данных (Apache Airflow, Prefect), сервисы координации (Consul, ZooKeeper) и многие другие. Ситуация осложняется постоянным изменением стека используемых технологий.

Нельзя сказать, что отрасль распределённой обработки данных испытывает дефицит специализированной литературы — напротив, её весьма много. Пользователям доступно множество лекционных курсов, статей, монографий. Однако для того, чтобы ориентироваться в море технической информации необходимо компактное учебное пособие, которое могло бы служить быстрым стартом для вхождения в предметную область. Именно с этой целью и было написано данное пособие.

Здесь кратко рассматриваются основные сведения о хранении и обработке данных в распределённых системах. Достаточно много места уделено вопросам координации компонентов систем с помощью распределённых алгоритмов. Отличительной особенностью данного издания является описание технологии блокчейн, достаточно редко освещаемой в учебной литературе.

Предлагаемое пособие может быть полезно студентам, планирующим карьеру в качестве инженера данных (data engineer), системного архитектора и т.д. Усвоение материала облегчит чтение более специализированной профильной литературы и технической документации.

2. Введение в распределённые системы

2.1 Определение распределённой системы

В попытке найти общие черты у распределённых систем разных классов, исследователи предложили несколько определений термина «распределённая система». Приведём три из них, подчёркивающие различные грани этого понятия.

Определение Э. Танненбаума:
"Распределенная система — набор независимых компьютеров, представляющийся пользователям единой объединенной системой."

M. van Steen and A.S. Tanenbaum
«Distributed Systems: Principles and Paradigms»

Определение Дж.Калуриса:
"Распределенная система состоит из аппаратных и программных компонентов, расположенных в сети компьютеров, которые взаимодействуют и координируют свои действия только с помощью посылки сообщений."

G. Coulouris et al «Distributed Systems: Concepts and Design»

Определение Л.Лэмпорта:
"Вы понимаете, что пользуетесь распределенной системой, когда поломка компьютера, о существовании которого вы даже не подозревали, приводит к останову всей системы."

Email of 28 May 1987 https://lamport.azurewebsites.net/pubs/distributed-system.txt

В нашем курсе нам будет достаточно считать распределённой системой некоторый набор вычислительных узлов, объединённых коммуникационной средой (проводной либо беспроводной). Предполагается, что на каждом узле запущен ровно один процесс, каждая из выполняющий некоторую часть функционала системы.

Отметим, что в литературе часто под узлами распределённой системы понимаются процессы, которые могут располагаться как на одном вычислительном устройстве, так и на разных.

2.2 Классификация

Распределённые системы очень разнообразны. Э. Танненбаум в своём монументальном труде [1] классифицирует их следующим образом:

- вычислительные — предназначенные для проведения расчётов;
- информационные — используемые для хранения и обработки информации;
- всепроникающие (pervasive) — объединение устройств, снабжённых датчиками; в наше время подобные системы относят к интернету вещей (internet of things, IoT)

В данном пособии рассматриваются в основном распределённые информационные системы.

2.3 Причины построения распределённых систем

Переход от локальной системы (располагающейся на одном компьютере) к распределённой — трудоёмкий и дорогостоящий процесс. В каких случаях это необходимо? Здесь возможны разные варианты:

- необходимость в обработке больших объёмов данных;
- обслуживание географически распределённых пользователей;
- распределение нагрузки;
- увеличение производительности;
- повышение отказоустойчивости;
- предоставление совместного доступа к некоторому общему ресурсу.

Кроме того, как показывают исследования, горизонтальное масштабирование вычислительной системы (переход от системы с одним узлом к кластеру) может быть экономически выгоднее вертикальной масштабируемости (увеличение мощности имеющегося сервера).

2.4 Основные характеристики распределённых систем

Эффективности распределённой системы характеризуется следующими основными показателями

- время отклика системы (response time): время от отправки пользовательского запроса до получения ответа;
- пропускная способность системы (throughput): количество однотипных запросов, обрабатываемых за единицу времени.

Способность распределённой системы справляться с увеличением нагрузки без потери эффективности называется масштабируемостью (scalability). Выделяют следующие виды масштабируемости:

- нагрузочная масштабируемость — способность справляться с ростом количества запросов;
- географическая масштабируемость — способность справляться с увеличением расстояния между узлами;
- административная масштабируемость — способность справляться с увеличением количества административных делений системы.

Одним из факторов, позволяющих достичь хорошей масштабируемости, является *открытость* (openness) системы. Под открытостью понимается использование в системе открытых стандартов на интерфейсы, службы и форматы данных. В открытой системе новые узлы подключаются легко, если они используют общепринятые протоколы и форматы данных.

Масштабируемость позволяет достичь *прозрачности*. Прозрачность (transparency) — способность системы скрывать свою распределённую природу. В идеале пользователю должно казаться, что он работает с одной машиной.

При этом отказы части узлов также желательно скрывать от пользователя, т. е. система должна быть отказоустойчивой. *Отказоустойчивость* (fault tolerance) — способность системы вести себя четко определенным образом при возникновении неисправностей.

Эффективность распределенной системы ограничивается следующими двумя основными факторами:

- задержка сети (latency): время, необходимое для передачи сообщения из одного местоположения в другое в распределенной системе;
- пропускная способность сети (bandwidth): объем данных, который может быть передан за единицу времени в стабильном состоянии (когда свойства потока данных не меняются со временем).

Ещё одним важным понятием является *доступность* системы (availability, иногда high availability), под которым понимается способность системы давать ответ на запрос клиента (пусть даже возвращённые данные будут не самыми свежими). На практике часто пользуются метрикой SLA (service level agreement) — доля времени, в течение которого система находится в рабочем состоянии.

2.5 Модель распределённой системы

Под моделью распределённой системы обычно понимают совокупность предположений о её свойствах.

Одним из важнейших является предположение о синхронности либо асинхронности системы. Оно оказывает существенное влияние на выбор алгоритмов, которые будут положены в основу функционирования данной системы.

Синхронной мы называем такую систему, для которой известна верхняя оценка на

- время передачи сообщения,
- время обработки сообщения,
- скорость расхождения показаний часов на узлах.

Асинхронной же называется система, для которой такие оценки отсутствуют.

Ещё один момент, который обычно рассматривается при проектировании систем, это наличие либо отсутствие свойства FIFO каналов связи, т. е. гарантий доставки сообщений от узла A узлу B в том же порядке, в котором они были отправлены узлом A.

В любую распределённую систему закладываются механизмы отказоустойчивости. Однако нельзя бороться с теми отказами, которые не были предусмотрены на этапе проектирования. Таким образом, механизмы безопасности создаются в расчёте на возможность возникновения отказов следующих типов:

1. отказы узлов:
   - стандартный отказ узла (fail-stop) — узел перестаёт работать (окончательно), причём все остальные узлы узнают об этом событии;
   - отказ с возможностью восстановления (fail-recover) — в некоторый момент узел может вернуться в рабочее состояние;
   - византийские отказы — непредсказуемое поведение узлов; такую ошибку невозможно воспроизвести — разные узлы получают разные ответы на идентичные запросы;
2. отказы каналов связи:
   - частичная потеря сообщений в канале;
   - разделение сети (network partition) — полная потеря сообщений в некотором канале связи.

2.6 Проблемы построения распределённых систем

Проблемы, связанные с сетью. Любая распределённая система включает в себя некоторую коммуникационную среду. Неидеальности этой среды — потеря сообщений, задержки и т. д. — должны учитываться в программах, однако зачастую это не делается. На невозможность написания программ для распределённых систем без учёта наличия коммуникаций по сети указывали известные специалисты П.Дойч и Дж.Гослинг из Sun Microsystems. В 1994 году они сформулировали так называемые «восемь заблуждений о распределённых системах» («8 fallacies of distributed computing»):

1. сеть надёжна;
2. нулевая задержка;
3. бесконечная ширина пропускания;
4. сеть безопасна;
5. топология не меняется;
6. администратор только один;
7. передача данных бесплатна;
8. сеть однородна.

Проблемы, связанные с распределённой природой системы. В системах большого масштаба трудно рассчитывать на наличие *глобального времени*. В связи с тем, что физические часы на различных узлах не являются идентичными, их показания могут расходиться. Простейшее решение — регулярная синхронизация часов — не всегда приемлемо в связи с большими накладными расходами.

В распределённой системе (особенно больших масштабов) трудно собирать и поддерживать в актуальном состоянии *глобальную информацию* — данные о состоянии всех узлов системы. В связи с этим алгоритм каждого узла, должен действовать на основе только локальной информации. В частности, необходимо учитывать, что данные о текущем

состоянии других узлов может быть устаревшей. Все эти обстоятельства приводят к усложнению алгоритмов функционирования узлов системы.

Отметим, что отказы в распределённой системе являются *частичными* (partial failures). Это означает, что отказ одного узла не приводит к остановке работы всей системы. Несмотря на плюсы, связанные с этим обстоятельством, обработка частичных отказов является непростой задачей в связи со сложностями обнаружения таких отказов.

Одновременная работа компонент распределённой системы также является источником проблем. Необходимо предусмотреть механизмы координации работы узлов (например, механизм взаимного исключения). Разнообразные задачи координации могут быть сведены к задаче консенсуса, которую часто иллюстрируют с помощью аллегории, называемой *проблемой генералов*.

Допустим, что две армии, каждая под командованием своего генерала, подошли к городу. Армии расположились достаточно далеко друг от друга, генералы могут общаться только путём посылки гонцов. Командующим необходимо договориться о времени штурма. Это важно для успеха кампании — если армии атакуют порознь, они будут разбиты по отдельности.

Существует разновидность этой задачи, в которой некоторые генералы могут быть предателями и намеренно пытаться разрушить консенсус. Этот вариант называется *задачей о византийских генералах* и иллюстрирует ситуацию, когда некоторые узлы подвержены непредсказуемым сбоям (например, вследствие перегрева), либо захвачены мошенниками.

2.7 Реализация распределённых систем

При создании распределённой системы, как правило, необходимо реализовать следующий функционал:

- хранение данных;
- координация работы узлов;
- коммуникации;
- обработка данных.

Существует множество решений (в том числе свободно распространяемых) для каждого пункта этого списка. Таким образом зачастую создание распределённой системы состоит в объединении и настройке готовых компонент.

2.8 Вопросы для самопроверки

1. Какие существуют определения понятия “распределённая система” ?
2. Какие предположения могут делаться при проектировании распределённых систем?
3. Дайте определения терминам масштабируемость, прозрачность, отказоустойчивость

3. Хранение данных в распределённых системах

Эффективная работа с большими массивами ценной информации невозможна без правильной организации хранения данных. Большие объёмы (терабайты и даже петабайты) этих данных делают невозможным выполнение их обработки на одном сервере, их приходится разбивать на части (*секционировать*) и распределять по разным узлам системы.

При этом необходимо решить целый ряд проблем — надёжное хранение метаинформации (карты расположения частей массива данных в системе), обеспечение равномерности

распределения данных по узлам, а также отказоустойчивость — отказ одного из серверов не должен приводить к полной потере секции данных. Для решения последней проблемы применяется *репликация* — хранение на различных узлах копий одного и того же массива данных.

3.1 Секционирование

Этот метод хранения состоит в разбиении массива данных на части и размещении каждой части на отдельном узле системы. Рассмотрим варианты секционирования табличных данных.

Существует два основных варианта — горизонтальное и вертикальное разбиение. При *горизонтальном* разбиении таблица делится на наборы строк. При этом, как уже было сказано, важно обеспечить приблизительное равенство объёма хранимой информации на узлах системы. Предполагается, что каждая строка имеет поле с уникальным значением, которое называется *ключом*. В результате горизонтальное разбиение сводится к разбиению множества ключей. Для решения этой задачи существует целый ряд стандартных методов:

- диапазонный (range): множество ключей разбивается на фиксированные диапазоны (например, по первой букве);
- списочный (list): явно указывается список значений для каждой части;
- основанный на хэшировании (hash): разбиение в соответствии с хэшем ключа.

Отметим, что горизонтальное секционирование таблиц иногда называют *шардингом* (от shard – осколок).

Существует также *вертикальное* разбиение таблиц, используемое реже, при котором разрезание производится по столбцам, в результате чего происходит разделение схемы данных. Другими словами, из исходной таблицы создаётся несколько новых с меньшим количеством столбцов. Новые таблицы будут храниться на разных узлах, поэтому при запросах, в которых требуются все столбцы исходных строк, время отклика может оказаться достаточно большим из-за обмена сообщениями с несколькими узлами и объединении полученной информации. Однако если большинство запросов затрагивают лишь несколько столбцов исходной таблицы вертикальное разбиение становится эффективным.

3.2 Репликация

При работе с ценным массивом информации необходимо хранить одну или несколько копий этого массива. Это позволяет не только повысить отказоустойчивость, но и уменьшить время отклика системы (например, за счёт распределения запросов на чтение). Хранение нескольких копий одного и того же массива данных на разных узлах распределённой системы называется *репликацией*.

При всей простоте идеи репликации существует множество вариантов её реализации. Приведём несколько способов систематизации этих вариантов.

Классификация по количеству узлов, способных принимать запросы на запись:

- репликация с одним ведущим узлом (master-slave) — в системе есть один выделенный узел, который принимает запросы на запись; запросы на чтение принимают все узлы;
- репликация с несколькими ведущими узлами (master-master) — в системе есть несколько узлов, которые принимают запросы на запись; запросы на чтение принимают все узлы;
- репликация без выделенных узлов (masterless) — все узлы принимают запросы на запись и чтение.

Классификация по принципу фиксации операции записи:

- синхронная — операция записи завершается после того, как обновления достигнут всех узлов;
- асинхронная — операция записи завершается сразу после того, как обновления зафиксированы на основном узле (который иногда называют *мастер-узлом*); мастер-узел подтверждает клиенту завершение операции, не дожидаясь, пока подчинённые узлы подтвердят мастер-узлу применение обновления на своих локальных копиях;
- кворум (majority) — операция записи завершается после того, как обновления достигнет большинства реплик, и они сообщат об этом мастер-узлу.

Классификация по принципу передачи данных:

- физическая — обновление базы формулируется в терминах изменения страниц памяти;
- логическая — обновление базы формулируется в терминах изменения строк таблиц;
- репликация в виде потока команд.

Стоит подчеркнуть, что существует различие между репликацией и резервированием данных (бэкапом). При репликации основная и подчинённые копии обновляются после каждой записи. Резервирование же происходит периодически по расписанию.

3.3 Классификация СУБД

При проектировании распределённой системы необходимо определить, как будут храниться данные. С точки зрения удобства доступа предпочтительнее хранение в некоторой *системе управления базами данных* (СУБД), однако вариант работы с сырыми данными, хранящимися в виде набора файлов, также возможен. В больших проектах присутствуют как сырые данные, так и данные в СУБД, причём может использоваться несколько СУБД — каждая для своей подзадачи.

Среди СУБД традиционно доминировали реляционные, в которых информация хранится в таблицах, связанных между собой посредством внешних ключей. Эти СУБД имеют ряд неоспоримых достоинств, которые обеспечивают им популярность и в настоящее время. Данные в них хорошо структурированы и подчиняются заранее заданной схеме отношений (под схемой понимается набор имён атрибутов с указанием соответствующих типов). Используемый для запросов язык SQL имеет солидную теоретическую базу в виде реляционной алгебры. Имеется механизм транзакций — группировки операций с данными, хранящимися в данных, в одну логическую единицу, которая либо полностью выполняется, либо полностью откатывается. Реляционные СУБД предоставляют ряд гарантий на выполнение транзакций, известных как ACID (по-английски acid — «кислота»). Акроним ACID состоит из первых букв следующих понятий:

- atomicity (атомарность) — невозможность частичного выполнения группы операций, входящих в транзакцию; при возникновении ошибки в какой-либо операции, система откатывается к тому состоянию, в котором она была перед транзакцией;
- consistency (согласованность) — гарантия того, что результат любой успешно выполненной транзакции не нарушает ограничений, заложенных при проектировании базы данных;
- isolation (изолированность) — отсутствие влияния параллельных транзакций друг на друга;
- durability (надёжность) — свойство, означающее, что даже в случае сбоя аппаратного обеспечения результаты зафиксированной транзакции не пропадут.

Позиции реляционных СУБД были незыблемы до конца 90х годов XX века, когда распространение интернета стало повсеместным и начался бум электронной торговли. Появилась необходимость работы с мощными потоками данных и оказалось, что необходимы новые механизмы работы с информацией. Стали набирать популярность альтернативные СУБД, которые часто объединяют под аббревиатурой NoSQL. Изначально она означала именно отказ от SQL, однако позднее получила более мягкую расшифровку «not only SQL».

Важную роль в подъёме NoSQL СУБД послужило представление Эриком Брюером, профессором Калифорнийского университета в Беркли, так называемой *CAP-теоремы* на конференции в 2000м году. Эта теорема гласит, что невозможно построить систему, которая одновременно обладала бы тремя свойствами:

- consistency — (строгая) согласованность (система в ответ на любой запрос возвращает самую свежую версию данных),
- availability — доступность (это понятие описано в п.2.4),
- partition tolerance — устойчивость к разделению сети (разрыву одного из соединений коммуникационной сети).

Позднее эта теорема многократно критиковалась, однако её влияние на всю отрасль распределённых систем бесспорно. Она легла в основу классификации по манере поведения при возникновении разделения в сети. Если в этой ситуации система делает выбор в пользу сохранения согласованности данных (останавливает работу до возобновления нормальной работы сети), то она называется CP-системой. Если же происходит выбор в пользу согласованности (система продолжают отвечать на запросы, несмотря на то, что ответ может содержать не самые свежие данные), то её относят к AP-системам.

Базы данных NoSQL предлагают различные модели хранения данных, отличающиеся от традиционных таблиц. Как правило эти модели более гибкие, чем жёстко заданные схемы в реляционных СУБД. Кроме того, предлагаются различные языки запросов.

Приведём одну из классификаций баз данных NoSQL по модели хранения данных:

- ключ-значение — данные хранятся в виде пар КЗ, где ключом как правило является строка, а значением может быть практически любая структура данных; примеры – Redis, memcache;
- документарные — данные хранятся в виде документов форматов JSON, XML и т. д.; пример — MongoDB;
- графовые — данные хранятся в виде графа; пример – Neo4j;
- колоночные базы данных(wide column, column-based) — представляют собой набор таблиц с более гибкой схемой, чем в случае реляционных СУБД; в этих таблицах есть несколько «широких» колонок, внутри которых в разных строках может быть разное количество ячеек; пример — HBase;
- движки полнотекстового поиска — они хранят корпусы текстов, внутри которых осуществляется поиск, схожий по функционалу с интернет-поисковиками; пример — ElasticSearch;
- блокчейн и другие технологии распределённого реестра (distributed ledger).

Базы данных NoSQL придерживаются набора более мягких требований, чем классический ACID. Этот набор требований был предложен Э.Брюером и получил название BASE (слово "base" означает "щёлочь"):

- Basically Available — во главу угла ставится доступность;
- Soft state — изменчивое состояние (т. е. даже при прекращении потока обновлений пользователь может увидеть изменение состояния БД);
- Eventually consistent — в качестве модели согласованности используется

согласованность в конечном счёте

Победное шествие NoSQL поставило под сомнение дальнейшее существование самого популярного языка запросов SQL. Однако со временем стало ясно, что отказаться от него невозможно — он очень удобен, накоплена большая кодовая база и количество программистов, владеющих этим языком, больше, чем специалистов по альтернативным языкам запросов. В результате стали появляться так называемые NewSQL решения, сочетающие в себе привычный язык запросов SQL и эффективость работы с распределёнными данными, присущие NoSQL.

3.4 Вопросы для самопроверки

1. Перечислите методы разбиения множества ключей
2. Какие существуют варианты репликации?
3. Перечислите разновидности СУБД

4. Координация в распределённых системах

Для того, чтобы распределённая система работала корректно (могла продолжать функционировать при отказе одного из узлов, не допускала бы потери данных и т. д.) приходится решать такие типовые задачи, как выбор лидера, взаимное исключение и некоторые другие, реализуя их в виде сервисных алгоритмов.

В целях отказоустойчивости желательно решать перечисленные задачи децентрализованным образом, без использования выделенного узла-координатора. Такие алгоритмы существуют, их называют *распределёнными*.

4.1 Распределённые алгоритмы

В распределённых алгоритмах каждый узел принимает решение на основе своей локальной информации о текущем состоянии системы. Эта информация может быть частично устаревшей, что не должно сказываться на корректности работы алгоритма.

Основными свойства, которыми должен обладать распределённый алгоритм, являются живость и безопасность.

Свойство *живости (*liveness) означает, что алгоритм движется вперёд и задача будет в конце концов решена. Иногда это свойство формулируют более общо: «что-то хорошее обязательно произойдёт»

Свойство *безопасности* (safety) подразумевает, что в процессе выполнения алгоритма некая нежелательная ситуация не возникнет ни при каких условиях («что-то плохое никогда не произойдёт»).

4.2 Обнаружение отказов

Прежде чем обрабатывать отказ его необходимо обнаружить. Будем считать, что в нашей системе реализован *детектор отказа* (failure detector) – некий абстрактный функционал, который обеспечивает обнаружение отказа узла. Это может быть алгоритм, выполняемый на узлах распределённой системы, либо внешний сервис.

Основные механизмы контроля работоспособности узлов:

• контроль со стороны системы: узлы шлют наблюдаемому узлу эхо-запросы (ping);
• пульсация (heartbeat): наблюдаемый узел сам шлёт другим узлам контрольные пакеты.

Детектор отказа может ошибочно определить факт отказа узла (из-за длительной задержки ответа). Для уменьшения вероятности ошибки узел, обнаруживший отказ, может «попросить» другие узлы послать контрольный пакет на подозрительный узел. Таким образом, определение факта отказа станет предметом консенсуса группы узлов.

В синхронной системе для определения факта отказа узла обычно используется предельное время ожидания (таймаут). Асинхронная система требует других подходов, например, использование счётчиков контрольных пакетов.

4.3 Выбор лидера

Многие распределённые системы используют выделенный узел-координатор (его также называют *лидером*) для общей организации работы. В случае отказа координатора система перестаёт функционировать. В связи с этим для повышения отказоустойчивости необходимо предусмотреть механизм, который позволил бы сделать один из оставшихся узлов новым лидером. Такой механизм называется *алгоритмом выбора лидера*.

Алгоритм выбора лидера должен обладать следующими свойствами:

- завершаемость: процесс выборов должен завершаться за конечное время,
- единственность: выбранный лидер должен быть только один,
- согласие: все узлы системы признают лидером один и тот же узел.

Здесь завершаемость является свойством живости, а единственность и согласие — свойствами безопасности.

Отметим, что задачу выбора лидера можно рассматривать как разновидность задачи консенсуса. Допустим, что у каждого узла есть свой идентификатор (id) и переменная leader_id. После отказа лидера все оставшиеся узлы должны договориться о том, кто из них станет новым лидером. Другими словами, переменная leader_id на всех узлах должна принять значение id одного из узлов.

Алгоритм Гарсии-Молины

Рассмотрим алгоритм, который в разных вариациях используется до сих пор — алгоритм Гарсии-Молины или алгоритм громилы (bully algorithm). Он послужил родоначальником целого класса алгоритмов поиска экстремума (в результате выборов лидером становится узел с наибольшим идентификатором).

Алгоритм работает в предположениях наличия у каждого узла уникального идентификатора, синхронности системы, надёжности сети, возможности отказа узлов. Каждый узел знает общее количество узлов и их идентификаторы, но не знает их текущего состояния (работает/не работает).

Схема работы алгоритма следующая:

- узел P запускает выборы, если подозревает падение лидера;
- P рассылает сообщение ELECTION узлам с б**о**льшими id;
- если P получает в ответ OK, он становится пассивным, ждёт окончания выборов;
- если сообщение ELECTION остаётся без ответа, P становится лидером, рассылает всем сообщение LEADER;
- узел, получивший ELECTION от узла с меньшим id, шлёт в ответ OK, запускает свой выборный процесс;

- в случае если бывший лидер перезапустится, он инициирует новые выборы.

Описанный алгоритм прост в реализации, однако имеет ряд недостатков. Основным из них является сложность в терминах сообщений (message complexity) — для проведения выборов необходимо $O(N^2)$ служебных сообщений, где N — количество узлов в распределённой системе.

Кольцевой алгоритм

Схожая идея процесса выбора с максимальным идентификатором используется в алгоритмах, предназначенных для работы на (логической) кольцевой топологии.

Простейший из них выглядит следующим образом:

- кольцо однонаправлено — сообщения двигаются либо только по часовой стрелке, либо против часовой;
- узел, заметивший падение лидера, запускает по кольцу сообщение ELECTION; в это сообщение инициатор добавляет свой идентификатор;
- предполагается, что сообщение может передвигаться по кольцу и после отказа лидера (т. е. физически сеть остаётся связной);
- узел, получив ELECTION, добавляет в него свой идентификатор в случае если этот идентификатор отсутствует в сообщении; затем пересылает сообщение дальше;
- если узел, получив ELECTION, обнаруживает в нём свой идентификатор, он выбирает максимальный из всех идентификаторов, содержащихся в сообщении; узел с максимальным идентификатором объявляется лидером, сообщение LEADER с этим идентификатором запускается по кольцу;
- узел, получив LEADER, обновляет свою переменную leader_id, пересылает сообщение дальше;
- после того как сообщение LEADER сделает полный круг, оно изымается из обращения.

Основным недостатком данного алгоритма является рост размера сообщения ELECTION по мере его движения по кольцу. Существует модификация (алгоритм Чанга-Робертса), в котором сообщение ELECTION всегда содержит один идентификатор.

4.4 Глобальное упорядочивание событий

Во время работы распределённой системы на узлах происходят различные события: чтение, запись, завершение какого-либо вычисления и т. д. Для многих задач бывает полезно ввести глобальное упорядочивание на множестве всех событий системы. Так, например, если мы выполняем изменения реплицированных (хранящихся в нескольких экземплярах) данных, необходимо, чтобы все узлы управляющие копиями этих данных, применяли обновления в одинаковом порядке.

Естественным решением было бы упорядочить события в соответствии с физическим временем. Для этого достаточно приписать каждому событию временну́ю метку. Но здесь мы сталкиваемся с проблемой неидеальности компьютерных часов — они могут рассинхронизироваться. Необходим механизм, который заменил бы физическое время и не зависел бы от разницы в скорости часов различных узлов. В качестве такого механизма может выступать так называемое *логическое время*.

Причинно-следственные связи

Не любое упорядочивание событий можно считать удовлетворительным. Например, если в этом упорядочивании событие отправки некоторого сообщения располагается после события приёма этого сообщения, то это будет нелогичным с точки зрения стороннего наблюдателя. Другими словами, произойдёт нарушение *причинно-следственных связей*.

Этот тип связей между событиями (которые также называют *каузальными*) были формализованы Л.Лэмпортом в виде отношения «произошло до». Будем говорить, что события *a* и *b* связаны отношением «произошло до» ($a \rightarrow b$), если

- это два внутренних события одного узла, причём *a* произошло раньше *b* по локальным физическим часам этого узла;
- событие *a* — событие отправки некоторого сообщения, событие *b* — событие приёма этого сообщения;
- между *a* и *b* есть цепочка событий, связанных соотношением «произошло до».

Логическое время

Первенство в разработке теории логического времени также принадлежит Л. Лэмпорту. В его модели распределённой системы рассматривалось три типа событий:

- внутреннее событие,
- событие отправки сообщения,
- событие приёма сообщения.

Пусть у нас есть функция $F(e)$, приписывающая каждому событию *e* метку, выражающую некоторое условное (логическое) время. Такую функцию будем называть *часовой*. Эта функция должна удовлетворять условию $F(a)<F(b)$ при $a \rightarrow b$.

Первой конкретной реализацией часовой функции были скалярные часы Л. Лэмпорта. Скалярные часы по сути представляют переменную-счётчик, работа с которой ведётся по следующим правилам:

- у каждого узла P_i свои локальные часы T_i; изначально значение часов всех узлов равно нулю;
- перед наступлением внутреннего события или события отправки значение T_i увеличивается на 1, событию приписывается обновлённая метка T_i;
- метка события отправки вкладывается в сообщение ;
- при приёме сообщения msg значение T_i обновляется по правилу $T_i = \max(T_i, T_{msg})+1$, событию приёма приписывается метка T_i .

Приведённый алгоритм позволяет соблюдать причинно-следственные связи, т.е. при $a \rightarrow b$ выполняется $T(a)<T(b)$. Таким образом, мы получаем возможность упорядочивать события по отметкам событий, состоящим из скалярной отметки и номера узла. Отметим, что в отличие от привычных физических часов, логические часы продвигаются вперёд только при наступлении некоторого события.

Снимок мгновенного состояния

Рассмотрим использование скалярных часов на примере задачи сохранения состояния распределённой системы. Под состоянием системы мы понимаем совокупность состояний узлов и каналов.

Проиллюстрируем проблему получения снимка с помощью так называемой *банковской*

задачи. Предположим, что некоторый банк представляет собой систему взаимосвязанных филиалов. Время от времени филиалы пересылают друг другу некоторые суммы денег. Для простоты будем считать, что деньги не выводятся из системы и не заводятся в систему извне. Необходимо в некоторый момент определить общее количество средств в системе.

При наличии идеально синхронизированных часов в филиалах можно использовать физическое время. Единственной проблемой является лишь вероятность наличия средств в процессе пересылки в момент фиксации. Однако здесь можно применить следующий приём: в момент фиксации записывается наличная сумма в данном филиале, а затем записываются все приходящие суммы, у которых время отправления меньше либо равно времени фиксации. Процесс заканчивается, когда приходит первое сообщение со временем отправления больше времени фиксации. Отметим, что данный критерий остановки работает только при наличии свойства FIFO у каналов передачи данных.

Описанное решение не работает при рассинхронизации часов филиалов, оно может приводить к некорректному подсчёту общей суммы. Поскольку мы не можем предсказать момент, когда расхождение часов станет значительным, целесообразным является переход к использованию скалярных часов.

В этом случае временем фиксации должен быть некоторый момент логического времени, который не пройден ни одним филиалом. Поскольку логическое время продвигается только при наступлении некоторого события, каждый филиал должен регулярно отсылать сообщения (это может быть пустое сообщение, если нет необходимости в посылке средств), чтобы логические часы каждого филиала гарантированно прошли момент фиксации .

В остальном алгоритм аналогичен случаю использованию физических часов. Каждый узел фиксирует наличную сумму перед выполнением первого события с меткой больше времени фиксации. После этого фиксируется приход всех сообщений с временем отправки меньше либо равным времени фиксации. Вновь для корректной работы алгоритма необходимо свойство FIFO каналов передачи данных.

4.5 Глобально упорядоченная рассылка

Рассмотрим систему, в которой каждый узел рассылает остальным узлам системы широковещательные сообщения. Например, это может быть реплицированная база данных, в которой каждая реплика может принимать запросы на запись, которые потом необходимо разослать всем остальным репликам.

Необходимо обеспечить, чтобы все сообщения на всех узлах были бы упорядочены одинаково. Такая рассылка называется глобально упорядоченной или *атомарной*.

Будем считать, что

- каналы обладают свойством FIFO,
- каждый узел имеет локальную очередь (буфер) сообщений.

Каждый узел, получив сообщение, помещает его в свою очередь, в которой поддерживается упорядоченность сообщений по комбинированным меткам времени $(L(m),i)$. Здесь $L(m)$ – метка времени, вложенная в сообщение m, а i — идентификатор узла, отправившего данное сообщение. Первым сообщением в очереди будет сообщение с минимальной меткой.

Узел доставляет сообщение m, стоящее в его локальной очереди первым, при условии, что от всех остальных узлов получены сообщения с меткой больше, чем у m. Это означает, что ни в одном канале нет сообщений с метками меньшими, чем у m (поскольку каналы обладают свойством FIFO).

Для того, чтобы гарантировать приход сообщений с меткой больше, чем у m, используются *подтверждения* (acks). Узел, получив сообщение, рассылает на него

подтверждение всем остальным узлам системы. Подтверждение будет иметь логическую метку больше, чем сообщение, на которое оно было послано.

Использование логических часов и подтверждений гарантирует доставку сообщений в одинаковом порядке на всех узлах, несмотря на то, что в некоторые моменты времени набор сообщений в очередях разных узлов будет различен.

Отметим, что атомарная рассылка (реализованная в протоколе ZAB — ZooKeeper atomic broadcast) используется в распределённом сервисе координации ZooKeeper, который обеспечивает отказоустойчивость кластеров Hadoop.

4.6 Взаимное исключение

Задача взаимного исключения исследуется с 60х годов XX века, и за это время было разработано достаточно большое количество подходов к её решению — семафоры, мониторы Хоара и т. д. Все они предполагают доступ узлов к некоторым разделяемым переменным. В распределённых системах такой возможности нет, в связи с чем возникает необходимость в разработке новых подходов.

Простейшим из них является централизованный алгоритм, в котором один из узлов выполняет роль координатора. Все остальные узлы будем называть *рабочими*.

Рабочий узел работает с разделяемым ресурсом по следующей схеме:
- посылает сообщение REQUEST координатору;
- ожидает разрешения (сообщение OK) от координатора;
- после выхода из критической секции посылает сообщение RELEASE координатору.

Узел-координатор, получив заявку, отвечает отправителю OK если с разделяемым ресурсом не работает ни один узел, в противном случае помещает задачу в очередь. Получив RELEASE от узла, вышедшего из критической секции, координатор удаляет его заявку из очереди, посылает сообщение OK узлу, чья заявка находится в голове очереди.

Данный алгоритм обладает рядом достоинств:
- прост в реализации,
- на каждый цикл работы с разделяемым ресурсом одного рабочего узла требуется всего три служебных сообщения (REQUEST, RELEASE, OK).

К недостатком данного подхода следует отнести:
1. наличие узкого места и единой точки отказа (координатор);
2. возможно нарушение причинно-следственных связей.

Проиллюстрируем второй пункт следующим примером. Допустим, система состоит из координатора K и двух рабочих узлов — W_1 и W_2. Узел W_1 посылает заявку req_1. Заявка идёт долгое время, и за это время последовательно происходят следующие события:
1. узел W_1 посылает сообщение m узлу W_2 ,
2. узел W_2 получает m,
3. узел W_2 посылает заявку req_2, которая приходит координатору раньше req_1,
4. только после этого координатор получает заявку req_1.

Очевидно, что req_1 предшествует (по Лэмпорту) заявке req_2 , поскольку имеется цепочка причинно-следственных связей $req_1 \rightarrow m \rightarrow req_2$. Тем не менее req_1 будет обработана позже.

Рассмотрим более сложный алгоритм, не использующий координатора и сохраняющий причинно-следственные связи. Его автором является Л.Лэмпорт.

Этот алгоритм предполагает наличие у каждого узла своей очереди запросов Q. Узлы действуют по следующей схеме:

- узел, желающий войти в критическую секцию (КС), рассылает всем узлам (включая себя) сообщение REQUEST, содержащее отметку времени;
- при получении запроса REQUEST узел помещает его в свою очередь Q. Если узел не находится в КС, он посылает отправителю запроса сообщение ACK. В противном случае узел откладывает отправление подтверждения до выхода из КС;
- узел входит в КС, если 1) его запрос имеет наименьшую отметку времени среди всех запросов в его локальной очереди; 2) он получил подтверждения на свой запрос от всех остальных узлов системы;перед выходом из КС узел 1) удаляет свой запрос из своей очереди; 2) посылает соощение RELEASE всем остальным узлам;
- при получении сообщения RELEASE узел удаляет соответствующий запрос из своей очереди.

Отметим, что данный алгоритм использует намного больше служебных сообщений, чем централизованный алгоритм. На один цикл работы с разделяемым ресурсом требуется 3(N-1) сообщение, где N — количество узлов в системе. Существует модификация алгоритма Лэмпорта — алгоритм Рикарда -Агравала — который требует 2(N-1) сообщений.

4.7 Распределённые транзакции

Транзакции (набор операций, который или полностью выполняется, или откатывается до начального состояния) — важный элемент работы с базами данных. При переносе транзакционной семантики на реплицированные данные, возникает проблема координации. Транзакцию можно фиксировать только если все реплики смогли её выполнить, если же хотя бы одна реплика выполнить транзакцию не смогла — откат нужно делать на всех репликах, иначе данные станут несогласованными.

Одним из самых распространённых алгоритмов фиксации распределённых транзакций является алгоритм двухфазной фиксации (two-phase commit, 2PC). Этот алгоритм требует наличие узла-координатора, который руководит выполнением транзакции.

Алгоритм 2PC выполняется по следующей схеме:

Фаза 1: координатор связывается с участниками (репликами), запрашивая, может ли каждая из них быстро и гарантированно завершить транзакцию. Если данная реплика может выполнить данную транзакцию (её ресурсы не заблокированы под другую транзакцию), она блокирует свои ресурсы и отвечает OK. В противном случае она отвечает NO. После сбора откликов:

- если все реплики ответили OK, координатор принимает решение зафиксировать транзакцию;
- если хотя бы одна реплика ответила NO, координатор принимает решение откатить транзакцию.

Фаза 2: координатор связывается с репликами и приказывает им либо зафиксировать транзакцию, либо откатить её. Реплики выполняют решение координатора.

4.8 Общие протоколы консенсуса

Большинство задач координации в распределённых системах могут быть сведены к *задаче консенсуса*. Наиболее общая формулировка этой задачи следующая: все узлы должны договориться о значении некоторой переменной.

Алгоритм консенсуса должен удовлетворять трём требованиям:

- завершаемость — в конце концов каждый корректный узел принимает некоторое решение;
- согласие — все корректные узлы приходят к одному решению;
- истинность — принятое значение должно было быть предложено одним из узлов.

Свойство завершаемости является свойством живости, согласие и истинность — свойствами безопасности.

Наиболее известными общими протоколами консенсуса являются Paxos, Raft, ZAB. Отметим, что все эти алгоритмы не устойчивы к византийским отказам.

Как правило алгоритм консенсуса реализуется в виде некоторого сервиса, на основе которого строятся выбор лидера, взаимное исключение и т. д. Существует даже выражение «консенсус — сердце распределённой системы».

Теорема FLP

Теорема Фишера-Линч-Патерсона (FLP) представляет собой утверждение о невозможности (impossibility result), которых достаточно много в теории распределённых вычислений. Ещё одним примером такого рода утверждений является уже упомянутая CAP-теорема Брюера. Ценность этих результатов состоит в том, чтобы предостеречь исследователей от попыток создания заведомо нереализуемых алгоритмов.

Теорема FLP формулируется следующим образом: *в асинхронной системе при возможности отказа хотя бы одного узла невозможно создать детерминированный алгоритм консенсуса, который обладал бы одновременно свойствами живости, безопасности и отказоустойчивости*.

Алгоритм ZAB

ZAB расшифровывается как ZooKeeper atomic broadcast protocol (протокол атомарной широковещательной передачи сообщений) и используется в сервисе координации ZooKeeper. Он был предложен в работе *B. Reed, F. Junqueira* A simple totally ordered broadcast protocol (2008).

Требования к протоколу широковещательной рассылки. Протокол ZAB использует лидера, который от имени клиента предлагает некоторое значение. Под клиентом здесь понимается пользователь распределённой системы, который даёт запросы на чтение или запись.

Время активности лидера называется *эпохой*. При смене лидера (в результате отказа старого) номер эпохи увеличивается на 1. Все остальные узлы называются *последователями*. Последователь подчиняется только командам лидера текущей эпохи. Лидер общается с остальными узлами кластера с помощью широковещательной рассылки.

Предполагается, что выполнены следующие условия:

- надёжная доставка**:** если сообщение m доставлено одним из узлов, то в конце концов оно будет доставлено всеми корректно работающими узлами;
- глобальное упорядочивание**:** если на одном из узлов сообщение a доставляется раньше сообщения b, то каждый узел, который доставляет a и b, доставляет a раньше b;
- сохранение причинно-следственных связей (каузальности): если сообщение a предшествует сообщению b и оба сообщения доставляются, то a должно доставляться раньше b.

Поскольку лидер может смениться, ZooKeeper добавляет следующее требование:

- если m — последнее сообщение, доставленное от лидера L, каждое сообщение, предложенное до m лидером L должно быть доставлено.

Перечисленные требования гарантируют корректность состояний реплик БД ZooKeeper:

1. надёжность доставки и глобальное упорядочивание гарантирует согласованность состояний реплик;
2. сохранение каузальности гарантирует, что состояние реплик корректно с точки зрения приложения, которое использует ZAB;

Описание алгоритма ZAB. В нормальном состоянии лидер рассылает (предлагает) другим узлам предложения по обновлению данных. Эти предложения могут либо приниматься, и тогда обновление фиксируется, либо отвергаться.

Работа кластера в этом случае идёт по следующей схеме:

- клиент подключается к случайному узлу кластера;
- при запросе на чтение отклик (считанные данные) поступает с этого узла;
- при запросе на запись:
  - если подключились к узлу-лидеру — запрос обрабатывается лидером,
  - если подключились к ведомому — запрос будет перенаправлен лидеру;
- для обработки запроса на запись лидер использует протокол двухфазной фиксации, чтобы обновить все копии данных; для достижения согласованности каждое обновление помечается идентификатором ZXID, который позволяет достичь глобального упорядочивания обновлений.

Выбор лидера. В случае, если один из узлов обнаружил отказ лидера, он инициирует выборы. Выборы может инициировать и сам лидер, если он не может должным образом взаимодействовать более чем с половиной серверов. При этом лидер как бы уступает лидерство узлу с лучшей сетевой средой.

После начала выборов все узлы переходят в состояние LOOKING (становятся кандидатами) и начинают обмен широковещательными сообщениями. В сообщение включается последняя эпоха лидера, о которой знает кандидат, ZXID последнего обновления, которое видел кандидат и идентификатор кандидата.

При голосовании выигрывает кандидат с более поздней эпохой. Если у двух кандидатов эпохи равны — сравниваются ZXID. Если и они равны — сравниваются идентификаторы.

Действия нового лидера. Новый лидер запрашивает у последователей информацию о последней эпохе, которую видел данный последователь. Номер новой эпохи равен максимуму из полученных эпох плюс единица. Новый номер эпохи рассылается всем последователям.

Затем каждый последователь присылает лидеру ZXID последнего зафиксированного на данном последователе обновления. Лидер присылает ему все недостающие обновления. После этого кластер ZooKeeper готов к работе с клиентами.

4.9 Вопросы для самопроверки

1. Каковы правила работы со скалярными часами Лэмпорта?
2. Сформулируйте теорему FLP
3. Для чего нужен механизм выбора лидера в распределённых системах? Приведите примеры соответствующих алгоритмов

5. Блокчейн

В 2008 году участник одной из емэйл-рассылок, посвящённых криптографии, по имени Сатоши Накамото опубликовал статью, которая называлась «A Peer-to-Peer Electronic Cash System». В ней он предложил децентрализованную систему платежей, которая могла бы функционировать без какой-либо доверенной третьей стороны (банка или другого финансового института). Эту систему он назвал Bitcoin, а технологию безопасного хранения данных, которая легла в её основу, *блокчейном*, т. е. цепочкой блоков.

Блокчейн представляет собой распределённую базу данных, которая в корне отличается от всех привычных СУБД. Она предназначена для хранения *транзакций* — записей о перемещении актива (аналог записей в бухгалтерской книге). Масштабы этой СУБД (в географическом смысле) очень велики — узлы находятся на разных континентах. Эти узлы принадлежат различным пользователям, не получающим разрешение на присоединение к системе от какого-либо администратора или организации.

В качестве актива, перемещения которого отслеживаются в блокчейне, в сети Bitcoin выступает одноимённая криптовалюта биткойн, для обозначения которой часто используется сокращение BTC. Мельчайшей долей биткойна является Сатоши (сокращённо Sat, во множественном числе Sats): 1 BTC = 10^8 Sats.

В связи с отсутствием какого-либо централизованного контроля за поведением пользователей, блокчейн вынужден опираться на алгоритмы консенсуса, обладающий устойчивостью к византийским ошибкам. Этот протокол называется proof-of-work, и он является самым известным на данный момент византийским протоколом консенсуса.

В рамках данного пособия блокчейн интересует нас поскольку является по сути первой успешной попыткой построения децентрализованной СУБД планетарного масштаба, способную противостоять злонамеренному поведению отдельных пользователей. В следующих параграфах рассмотрим основные идеи данной технологии.

5.1 Роли узлов

Сеть Bitcoin является одноранговой (пиринговой), в ней отсутствует какой-либо координатор. Каждый узел может выполнять одну или несколько ролей из следующего списка:

• полный узел — хранит полную копию блокчейна. Поскольку размер реестра транзакций Bitcoin на начало 2023 года составляет более 460 Гб, для выполнения этой роли узлу требуется вместительный жёсткий диск;

• лёгкий узел — как правило, это кошелёк Bitcoin, работающий на мобильном устройстве; не имеет полного реестра транзакций;

• майнер — создаёт (майнит) и добавляет новые блоки в блокчейн; благодаря ему транзакции записываются в реестр; узел данного типа должен располагать большими вычислительными ресурсами (GPU, ASIC) для выполнения процедуры майнинга (описание этой процедуры приводится ниже) .

5.2 Структура хранения данных

Блокчейн представляет собой односвязный список блоков. Каждый блок ссылается на предыдущий. Нулевой блок называется genesis block, т. е. блок зарождения.

В каждом блоке имеется заголовок и набор транзакций. Таким образом, блок — это часть общего реестра, в котором зафиксированы перемещения средств в сети Bitcoin. Отсюда и название класса технологий, к которому относится блокчейн — технологии распределённого

реестра (distributed ledger technologies, DLT).

Заголовок блока содержит следующие поля:

- номер версии протокола Bitcoin,
- хэш предыдущего блока,
- корень дерева Меркля (дерева хэшей, объединяющего транзакции данного блока) ,
- временная метка (примерное время создания блока),
- сложность (пороговое значение, используемое в майнинге),
- поле nonce (участвует в майнинге).

Хэш предыдущего блока выполняет роль ссылки на предыдущий блок. Таким образом, блокчейн представляет собой односвязный список с обратной связью.

Здесь мы сталкиваемся с одним из многочисленных механизмов безопасности, реализованных в технологии блокчейн. При попытке что-либо изменить в *i*-м блоке (записать транзакцию, выгодную злоумышленнику) происходит изменение хэша блока, и он уже больше не совпадает с хэшем, записанным в следующем блоке. Таким образом, необходимо заново создать (намайнить) *i*+1-й блок. Его хэш также изменится, нужно будет заново майнить *i*+2-й блок и так до конца цепочки. Поскольку майнинг требует значительных затрат электроэнергии, подмена транзакций становится невыгодной.

5.3 Майнинг

Майнинг представляет собой узел создания нового блока. Опишем его на примере. Предположим, на данный момент в блокчейне 99 блоков. Каждый майнер набирает порцию транзакций из специального буфера (mempool) и создаёт заготовку для блока номер 100. Поле nonce в заголовке нового блока устанавливается равным 0, после чего вычисляется хэш блока. Если полученный хэш окажется больше некоторого порогового значения, поле nonce увеличивается на 1 и вновь вычисляется хэш. Эта процедура выполняется параллельно всеми майнерами, они соревнуются в поиске значения nonce, которое обеспечит хэш блока меньший порога. Если эта цель достигнута, блок считается сформированным, и он рассылается остальным майнерам для проверки.

Майнер, получивший новый блок с номером 100, прекращает майнинг и проверяет блок на подлинность. Проверка проходит по нескольким критериям, самое важное установить, действительно была выполнена работа по поиску nonce (отсюда и название протокола — proof-of-work). Если проверка проходит успешно, майнер добавляет блок в свою копию блокчейна и переходит к вычислению блока номер 101.

Выигравший гонку за 100-й блок майнер получает награду за блок, а также комиссии за все транзакции, вошедшие в данный блок. В этом заключается ещё один, нетехнический механизм защиты блокчейна от мошенников. Если майнер будет вести себя нечестно (записывать транзакции, осуществляющие мошеннические действия), он скомпрометирует всю сеть, и пользователи начнут покидать её. Отток пользователей приведёт к падению прибыли майнера. Таким образом, майнеру по чисто меркантильным соображениям выгодно вести себя честно.

5.4 Транзакции

Транзакция (запись о перемещении средств) представляет собой структуру данных, в которой содержится описание её входов и выходов. Описание входа представляет собой номер выхода некоторой другой транзакции и идентификатор этой транзакции. Другими словами, вход — это ссылка на выход другой транзакции. В описании каждого выхода

присутствует объём переводимой криптовалюты и скрипт, необходимый для узла верификации права транзакции, ссылающейся на этот выход, тратить соответствующую сумму.

У пользователя сети Bitcoin отсутствует баланс счёта в привычном его понимании. Есть только набор выходов транзакций, направленных на его адреса. Те выходы, на которые не ссылается ни один вход ни одной другой транзакции, называются UTXO (unspent transaction output). Таким образом, баланс пользователя — это совокупность его UTXO.

Возможность задания транзакций с несколькими входами и выходами оправданно. Допустим, у пользователя есть два UTXO по 0,5 BTC. Он может перевести 1 BTC с помощью транзакции с двумя входами, ссылающимися на эти UTXO. И наоборот, при наличии UTXO объёмом в 1 BTC можно перевести по 0,5 BTC на два различных адреса — для этого нужна транзакция с двумя выходами.

Существует особый тип транзакций, не имеющий входов. Это так называемые coinbase-транзакции. В списке транзакций каждого блока блокчейна первой располагается транзакция этого типа. Она предназначена для перевода награды майнеру за создание нового блока, и это единственный способ эмитирования криптовалюты в сети Bitcoin.

Транзакции (за исключением coinbase-транзакций) генерируются при переводе некоторой суммы с одного счёта на другой. Новая транзакция распространяется по сети Bitcoin от узла к узлу. При попадании на узел-майнер транзакция проходит верификацию и помещается в буфер mempool. При создании нового блока майнер берёт транзакции из своего mempool.

5.5 Правило наиболее длинной цепочки

В сети Bitcoin возможна ситуация, когда два майнера создают два корректных блока с одинаковым номером практически одновременно. В результате в блокчейне возникает развилка. Это можно трактовать как нарушение согласия между узлами о том, какой блок будет следующим в блокчейне. Для разрешения этой конфликтной ситуации применяется правило *наиболее длинной цепочки*.

Состоит оно в следующем. При создании нового блока майнер обязан сделать его наследником последнего блока самой длинной цепочки. Если же обе ветки имеют одинаковую длину, новый блок майнится на тот из конечных блоков, который был получен первым данным майнером.

В конце концов одна из веток станет длиннее и будет объявлена верным продолжением блокчейна. Вторая ветка будет отсечена, блоки расформированы, транзакции снова попадут в сеть блокчейна (mempool-ы узлов).

Злоумышленник может попытаться добавить в свою копию блокчейна блок с мошеннической транзакцией и попробовать вырастить поверх него длинную цепочку, чтобы по правилам блокчейна она была бы признана верной. Однако этот вид атаки требует наличия у мошенника большого объёма вычислительных ресурсов.

5.6 Безопасность лёгкого узла

Легкий узел — это тип узла, который не имеет полной копии реестра. Эти узлы также называют SPV-узлами, где SPV расшифровывается как simplified payment verification (упрощенная проверка платежей). Такой узел обычно выполняет функции кошелька и размещается на каком-либо мобильном устройстве. Он хранит только часть информации из блокчейна, касающуюся транзакций, относящихся к данному узлу. Эту информацию он получает от полных узлов.

В целях безопасности лёгкий узел скрывает список счетов Bitcoin, которые его интересуют. Для этого при синхронизации с полным узлом он пересылает не сам список, а

фильтр Блума, заполненный по этому списку.

Фильтр Блума — это вероятностная структура данных, которая используется для проверки того, является ли элемент членом некоторого множества. Он представляет собой совокупность битового массива длины m (изначально заполненного нулями) и набора из n хэш-функций. Хэш-функции подбираются так, чтобы хэшем любой входной строки было бы число в диапазоне от 0 до m-1.

Каждый элемент множества хэшируется всеми хэш-функциями, в результате получаются числа $i_1,\ldots,i_n$. Биты фильтра с номерами $i_1,\ldots,i_n$ устанавливаются равными единице. Можно сказать, что происходит узел обучения фильтра на элементах данного множества.

Проверка принадлежности некоторого элемента данному множеству происходит следующим образом. Элемент также хэшируется всеми хэш-функциями фильтра, соответствующие позиции битового массива проверяются на наличие в них единиц. Если хотя бы в одной из этих позиций стоит ноль — элемент *однозначно* не принадлежит множеству; если же во всех позициях единицы — элемент *возможно* принадлежит множеству.

Лёгкий узел, для сокрытия интереса к определённым биткоин-адресам, посылает полному узлу фильтр Блума, обученный на этих адресах. Полный узел присылает SPV-узлу транзакции, удовлетворяющие полученному фильтру Блума, а также некоторую информацию, подтверждающую подлинность этих транзакций. Сюда входят, в частности, *проверочные цепочки* (authentication path) из соответствующих блоков.

Здесь стоит пояснить, что транзакции в каждом блоке организованы в так называемое дерево Меркля (хэш-дерево), корень которого хранится в заголовке. Дерево является бинарным и состоит из узлов, содержащих хэши их потомков. Сами транзакции хранятся в листьях дерева. Для проверки принадлежности некоторой транзакции определённому блоку нам достаточно иметь небольшую часть дерева Меркля, которая и называется проверочной цепочкой. SPV-узел выполняет такую проверку для всех полученных транзакций, выполняя последовательность хэшированний, результат которых должен совпасть с корнем дерева Меркля соответствующего блока.

5.7 Вопросы для самопроверки

- Опишите структуру блока
- Как достигается консенсус в Bitcoin (майнинг, правило самой длинной цепочки)?
- Какие средства применяются для обеспечения безопасности лёгкого узла?

6. Распределённая обработка данных

6.1 Большие данные

Вести обработку данных с помощью распределённой системы (кластера) целесообразно только в случае, когда задача не решается с помощью отдельного сервера. Это может быть в случае большого объёма данных, высокой скорости их поступления и т. д. В таких ситуациях говорят о *больших данных* (Big Data).

В литературе популярно определение больших данных через 3V:

- объём (volume): к настоящему времени объём накопленных данных во многих организациях исчисляется сотнями терабайтов;
- скорость (velocity): посты в социальных сетях, биржевая информация и многие другие категории данных генерируются с большой скоростью и должны быть

обработанны оперативно;
- неоднородность (variety): массивы данных, с которыми приходится иметь дело аналитикам, отличаются не только большим объёмом, но и разнообразием форматов хранения.

Существует также определение через 7V. К предыдущим трём V добавляются:
- изменчивость (variability): информация может менять своё значение при изменении контекста (например, слова естественного языка с течением времени) ;
- достоверность (veracity): подчёркивается необходимость предварительной очистки данных; точность результатов анализа зависит от того насколько тщательно отбрасывались ошибочные данные на этапе предобработки;
- визуализация (visualization): результатом анализа должно стать некоторое представление, которое без труда может быть воспринято человеком, например, графическое представление;
- ценность (value): смысл обработки больших данных состоит в извлечении некоторой пользы, т. е. превращении накопленного массива данных в нечто ценное для заказчика.

6.2 Виды обработки данных

Выделяют следующие основные виды обработки данных:
- OLTP — обработка в бухгалтерском стиле; большое количество запросов на чтение и запись, каждый из которых затрагивает лишь несколько строк; время отклика на запрос минимально;
- OLAP — анализируются большие объёмы данных; сеанс обработки может длиться несколько часов; сами сеансы запускаются с некоторой периодичностью по расписанию;
- потоковая — анализ некоторого окна (порции) из бесконечного потока данных.

6.3 Архитектура кластера Hadoop

Hadoop — фреймворк с открытым кодом, предназначенный для работы с большими объёмами данных на кластерах с большим количеством узлов. Разрабатывался для свободной поисковой машины Nutch программистами Дугом Каттингом и Майком Кафареллой. Начало разработки относится к 2005-му году, в 2006-м Дуг Каттинг стал сотрудником Yahoo. В 2008м году Hadoop стал проектом верхнего уровня инкубатора проектов Apache Software Foundation.

Изначально Hadoop задумывался как инструмент аналитической обработки данных с помощью технологии MapReduce (см. п.6.4). В более поздних версиях появилась возможность проводить другие типы обработки (вычисления на графах с помощью парадигмы Pregel, анализ потоков данных с использованием технологии Storm). Таким образом, Hadoop превратился в универсальное решение для распределённой обработки данных.

Кластер Hadoop состоит из набора серверов, размещённых в стойках (racks). Стойки связаны коммуникационной системой. Основные компоненты Hadoop:
- файловая система HDFS — обеспечивает хранение (чтение/запись) данных на узлах кластера;
- менеджер ресурсов YARN — управление ресурсами (выделение ресурсов под задачи), планирование задач;

- сервис координации ZooKeeper;
- библиотека MapReduce

Кластер строится по принципу ведущий — ведомые. На главном узле запускаются основной узел HDFS (NameNode) и основной узел YARN (Resource Manager)

Файловая система HDFS предназначена для хранения больших файлов. Файл разбивается на блоки, каждый из которых хранится в виде файла в локальной ФС узла. Размер блока по умолчанию равен 128 Мб. Каждый блок реплицируется. По умолчанию используется следующая схема репликации: две копии блока располагаются на серверах одной стойки, третья копия — на сервере в другой стойке.

Перечислим функции основных узлов HDFS:

- NameNode — хранение метаинформации (расположение файловых блоков на узлах кластера);
- DataNode — хранение блоков данных.

Менеджер ресурсов YARN используется в Hadoop начиная с версии 2.0, которая была выпущена в 2013 году. Его появление позволило разделить функции управления ресурсами и контроля вычислительных узлов, которые в первой версии Hadoop возлагались на один узел (JobTracker), что отрицательно сказывалось на производительности. Основными компонентами YARN являются:

- ResourceManager (диспетчер ресурсов) — главный узел, который распределяет ресурсы (дисковое пространство, ОЗУ, процессор) среди всех запущенных приложений;
- ApplicationMaster (менеджер приложения) — узел, который отвечает за отдельное приложение, получая для него ресурсы у менеджера ресурсов;
- NodeManager (менеджер узла) — агент, который запускается на каждом узле; он отслеживает ресурсы данного узла, отсылает отчёты о состоянии узла главному узлу.

Кратко опишем схему взаимодействия узлов при выполнении задачи MapReduce:

- клиент ставит задачу MapReduce на выполнение;
- диспетчер ресурсов координирует распределение вычислительных ресурсов в кластере;
- менеджеры узлов запускают и контролируют контейнеры (порции ресурсов) на узлах кластера;
- менеджер приложения координирует узлы map и reduce данной задачи. узлы задачи выполняются в контейнерах, которые планируются диспетчером ресурсов и управляются менеджерами узлов;
- распределенная файловая система HDFS предоставляет задачам совместный доступ к файлам.

6.4 Технология программирования MapReduce

На данный момент MapReduce является одной из важнейших технологий распределённых аналитических вычислений. Она была представлена в статье «MapReduce: Simplified Data Processing on Large Clusters», опубликованной в 2004 году. Её авторами были два сотрудника компании Google.

Технология MapReduce скрывает от пользователя все низкоуровневые детали процесса вычислений (коммуникации между узлами кластера, обеспечение отказоустойчивости и многое другое). Программист описывает задачу в виде двух функций: map() и reduce(). Функция map() для каждой входной пары ключ-значение создаёт набор промежуточных пар. Затем библиотека MapReduce автоматически (программист этот этап не контролирует)

группирует все промежуточные значения, связанные с одним и тем же промежуточным ключом K, и передает их функции reduce(). Функция reduce(), принимает промежуточный ключ K вместе с набором значений для этого ключа. Она аггрегирует значения вместе, и создает ноль или одно выходное значение.

Подсчёт слов (word count). Проиллюстрируем сказанное на примере алгоритма подсчёта уникальных слов в коллекции текстовых документов. Этот классический пример приводится практически во всех публикациях по MapReduce (включая оригинальную статью) и является своего рода «Hello world» в мире MapReduce.

Формулировка задачи проста: дан корпус текстов; подсчитать количество вхождений каждого уникального слова во всех имеющихся текстах.

Процесс вычислений при решении этой задачи происходит в несколько этапов:

1. Разбиение входных данных на порции (splitting). Каждая порция поступает своему процессу, выполняющему функцию map().
2. Отображение (mapping). Каждый процесс, выполняющий map(), разбивает строки на слова и создаёт набор пар ключ-значение, где ключ — слово, значение — единица.
3. Сортировка и объединение по ключу (shuffling). Пары с одинаковыми ключами объединяются — получается пара с данным ключом и массивом, содержащим все значения.
   Например:
   < «dog», 1 > , <«dog», 1>, <«dog», 1> → <«dog», [1,1,1]>
   На этом этапе также может происходить локальная свёртка(combine):
   <«dog», [1,1,1]> → <«dog», 3>.
   Цель локальной свёртки — уменьшения объёма информации, пересылаемой процессам reduce().
4. Пересылка промежуточных пар ключ-значение соответствующим процессам reduce() (распределение может происходить по первой букве ключа, по хэшу ключа или иному принципу).
5. Свёртка (reducing). По входным парам ключ-значение генерируются пары — результаты вычислений. Принцип свёртки в данном алгоритме (и в большинстве других) совпадает с принципом локальной свёртки combine.

Данный алгоритм представлен в [7] в виде лаконичной программы на псевдокоде :

```
map(String key, String value):
// key: название документа, value: текст
 for each word w in value:
       EmitIntermediate(w, "1");

reduce(String key, Iterator values):
// key: слово,  values: список счётчиков
 int result = 0;
 for each v in values:
       result += Int(v);
 Emit(AsString(result));
```

Здесь в функции map() в цикле перебираются слова в полученном куске текста. Для каждого слова создаётся (EmitIntermidiate) пара ключ-значение, где ключом является очередное слово, а значением единица. Функция reduce() получает на вход сгруппированные данные, в которых ключом является слово, а значением — массив чисел. Для каждой такой группы осуществляется суммирование элементов массива, после чего порождается (Emit) строка-значение.

6.5 Вопросы для самопроверки

1. Какие существуют определения понятия “большие данные”?
2. Из каких основных компонент состоит кластер Hadoop?
3. Сформулируйте основные идеи модели программирования MapReduce. Приведите пример алгоритма, соответствующего данной модели.

7. Заключение

Данное пособие изначально задумывалось как краткое введение в проблематику распределённой обработки данных. В связи с этим за пределами изложения осталось много достаточно важных тем, касающихся как теоретических основ данной области, так и конкретных инструментов, необходимых для работы с большими массивами информации. Читателям, желающим углубить полученные знания, необходимо обратиться к специализированной литературе, большáя часть которой имеется в открытом доступе.

В первую очередь хотелось бы порекомендовать замечательные книги М.Клеппмана [2] и А.Петрова [3]. Они обе совмещают фундаментальный подход с практической направленностью изложения. Для лучшего понимания отдельных тем (скалярные часы, банковская задача) стоит обратиться к методическому пособию [4] и оригинальной статье Лэмпорта [5].

При изучении технологий распределённого реестра нельзя обойти стороной статью С.Накамото [6] , в которой впервые была представлен блокчейн. Более обширное и углубленное изложение можно найти в ставшей классической книге А. Антонопулоса [7].

Технология MapReduce кратко описана в оригинальной работе сотрудников Google [8]. Реализациям различных алгоритмов с помощью этой технологии посвящена книга [9].

8. Литература

1. *Э. Таненбаум, М. ван Стеен* Распределенные системы: принципы и парадигмы. ДМК-Пресс (2021)
2. *М.Клеппман* Высоконагруженные приложения. Программирование, масштабирование, поддержка. Питер (2018)
3. *А. Петров* Распределенные данные. Алгоритмы работы современных систем хранения информации. Питер (2021)
4. *М. Косяков* Введение в распределенные вычисления. Изд-во ИТМО (2014)
5. *L. Lamport* Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM (1978)
6. *S. Nakamoto* A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf (2008)
7. *A.M.Antonopoulos* Mastering Bitcoin: Unlocking Digital Cryptocurrencies. O'Reilly Media, Inc. (2014)
8. *J.Dean, S. Ghemawat* MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM. (2004)
9. *D.Miner, A.Shook* MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems. O'Reilly Media, Inc. (2012)

Учебное издание

Никольский Илья Михайлович

РАСПРЕДЕЛЕННАЯ ОБРАБОТКА ДАННЫХ

Учебно-методическое пособие

Электронное издание сетевого распространения
Художественное оформление *Ю. Н. Симоненко*
Макет утвержден 25.08.2023. Изд. № 12504

Издательство Московского университета

119991, Москва, ГСП-1, Ленинские горы, д. 1, стр. 15
Тел.: (495) 939-32-91; e-mail: secretary@msupress.com
http: // msupress.com. Отдел реализации:
тел.: (495) 939-33-23; e-mail: zakaz@msupress.com

Пособие посвящено обработке больших объемов данных, хранимых распределенно на узлах вычислительной системы. Рассмотрены основные подходы к репликации и секционированию данных. Большое внимание уделяется методам синхронизации компонент распределенных систем, изложены основы технологии блокчейн, а также основная технология распределенной обработки данных MapReduce.

Издательство Московского университета

ISBN 978-5-19-011913-8