Распределенные системы

Содержание

1 Формализм. Логические часы Лампорта (свойства и алгоритм) 5

2 Формализм. Векторные часы (свойства и алгоритм) 6

3 Формализм. Часы с прямой зависимостью (свойства и алгоритм) 7

4 Взаимное исключение в распределенной системе. Централизованный алгоритм. 8

5 Взаимное исключение в распределённой системе. Алгоритм Лампорта 9

6 Взаимное исключение в распределённой системе. Алгоритм Рикарда и Агравалы 10

7 Взаимное исключение в распределённой системе. Алгоритм обедающих философов. 11

8 Алгоритм на основе токена. 12

9 Взаимное исключение в распределённой системе. Алгоритмы на основе кворума (простое большинство, рушащиеся стены). 13

10 Согласованное глобальное состояние (согласованный срез). Алгоритм Чанди-Лампорта. Запоминание сообщений на стороне отправителя. 14

11 Согласованное глобальное состояние (согласованный срез). Алгоритм Чанди-Лампорта. Запоминание сообщений на стороне получателя. 15

12 Глобальные свойства. Стабильные и нестабильные предикаты. Слабый конъюнктивный предикат. Централизованный алгоритм. 16

13 Слабый конъюнктивный предикат. Распределенный алгоритм. 18

14 Диффундирующие вычисления, пример. Останов. Алгоритм Дейксты и Шолтена. 19

15 Локально-стабильные предикаты: согласованные интервалы, барьерная синхронизация (3 алгоритма). Применение для определения взаимной блокировки (deadlock). 20

16 Упорядочение сообщений. Определения, иерархия порядков. Алгоритм для FIFO. 21

17 Упорядочение сообщений. Определения, иерархия порядков. Алгоритм для причинно-согласованного порядка. 22

18 Упорядочение сообщений. Определения, иерархия порядков. Алгоритм для синхронного порядка. 23

19 Общий порядок (total order). Алгоритм Лампорта. 24

20 Общий порядок (total order). Алгоритм Скина. 25

21 Иерархия ошибок в распределенных системах. Отказ узла в асинхронной системе — невозможность консенсуса (доказательство Фишера-Линча-Патерсона). 26

22 Консенсус в распределенных системах. Применение консенсуса: выбор лидера, terminating reliable broadcast. 30

23 Синхронные системы. Алгоритм для консенсуса в случае отказа заданного числа узлов. 31

24 Синхронные системы. Проблема византийских генералов. Алгоритм для N >= 4, f = 1. Объяснить идею обобщения для f > 1. 32

25 Синхронные системы. Проблема византийских генералов. Невозможность решения при N = 3, f = 1. 33

26 Недетерминированные алгоритмы консенсуса. Алгоритм Бен-Ора. 34

27 Paxos. Алгоритм, его свойства. 36

28 Paxos. Общие принципы (RSM, концепции). Основные модификации (fast paxos и multi paxos). 39

29 Шардирование. Общий принцип. Статическое отображение, остаток от деления, расширяемое побитовое хеширование, битовый бор, постоянное число секций. 40

30 Шардирование. Общий принцип, хеширование рандеву, консистентное хеширование, Multi-Probe Consistent Hashing. 42

31 Шардирование. Общий принцип, JumpHash. 44

32 Транзакции в распределенных системах. ACID. 2 Phase Locking. 47

33 Транзакции в распределённых системах. ACID. 2 Phase Commit. 49

34 Raft. Алгоритм, его свойства. 50

35 CAP теорема (концепции, подходы, без доказательства). 53

36 Gossip. Eventual consistency. CRDT и дельта-CRDT, примеры со счетчиком, множеством. 54

37 Leader/Follower репликация. Общий принцип, реализация, синхронная и асинхронная репликация. 56

38 Leader/Follower репликация. Failover (для синхронного и асинхронного случаев), масштабирование репликации, снимки данных. 59

39 Слабые модели консистентности. Мотивация, чтение собственных записей. 62

40 Слабые модели консистентности. Мотивация, монотонное чтение. 64

41 Слабые модели консистентности. Мотивация, чтение согласованного префикса. 65

42 Multi-Leader репликация. Мотивация, схема с множеством дата-центров, конфликты при записи. Синхронное разрешение конфликтов, избегание конфликтов. 67

43 Multi-Leader репликация: векторные версии, объединение версий. Разрешение конфликтов при чтении и записи, параллельное разрешение конфликтов, изменение состава кластера. 69

44 Multi-Leader репликация. Использование деревьев Мёркла для синхронизации реплик, Sloppy Quorum, Hinted Handoff. 71

45 Multi-Leader репликация. Last Write Wins. 72

46 Кеширование. Cache-Aside и Cache-Through архитектуры, слабые модели консистентности. 75

47 MapReduce. Последовательная реализация, примеры решаемых задач. 77

48 MapReduce. Распределенная реализация. Мапперы и редьюсеры, локальность map, сбои узлов, избыточность. 78

49 MapReduce. Каскады MapReduce задач, Combiner-оптимизация, Map-only задачи. 80

50 Распределённое объединение. Использование границ и слияние отсортированных последовательностей. 81

51 Resilient Distributed Datasets. Мотивация, реализация, секционирование датасетов, материализация датасетов. 83

52 Распределённое машинное обучение. Разделение градиента, алгоритм с обменом градиентами, проблемы при масштабировании. 85

53 Распределённое машинное обучение. Quantization, Sparsification, Error Correction. 86

54 Распределённое машинное обучение. Схемы пересылки сообщений, обмен весами, послойное обучение, SwarmSGD. 88

55 Самостабилизация: взаимное исключение 90

56 Самостабилизация: поиск остовного дерева 91

1 Формализм. Логические часы Лампорта (свойства и алгоритм)

Кратко опишем используемые далее обозначения.

Обозначение	Объект
$P, Q, R, \ldots \in \mathbb{P}$	Процессы
$a, b, c, \ldots \in \mathbb{E}$	События в процессах `proc(e)` $\in \mathbb{P}$
$m \in \mathbb{M}$	Сообщения, `snd(m)`, `rcv(m)` $\in \mathbb{E}$.

Таблица 1: Общие обозначения

Определение. Отношение *Произошло-до* ($\rightarrow$) – минимальный *строгий частичный порядок* (транзитивное, антирефлексивное, антисимметричное отношение) на $\mathbb{E} \times \mathbb{E}$ такой, что

- Если e, f в одном процессе и e идет перед f, то $e \rightarrow f$.
- Если m – сообщение, то `snd(m)` $\rightarrow$ `rcv(m)`.

Определение. *Логические часы*. Определим функцию $C : \mathbb{E} \rightarrow N$ так, чтобы

$$\forall e, f \in \mathbb{E}\ e \rightarrow f \implies C(e) < C(f).$$

Алгоритм. (Логические часы Лампорта)

- Каждый процесс хранит счетчик.
- Перед посылкой процесс увеличивает счетчик на единицу.
- При посылке дополнительно посылается счетчик.
- Получатель обновляет свое время следующим образом:

$$C \leftarrow \max(C, C_r) + 1.$$

Свойства логических часов Лампорта:

- Время события не уникально.
- Являются логическими часами в смысле определения.

2 Формализм. Векторные часы (свойства и алгоритм)

Определение. *Векторные часы*. Определим функцию $VC: \mathbb{E} \to N^k$ так, чтобы

$$\forall e, f \in \mathbb{E}\ e \to f \iff VC(e) < VC(f).$$

Сравнение производится покомпонентно.

Алгоритм. (Векторное время)

- Каждый процесс хранит свой вектор-время (размер – число процессов).
- Перед посылкой сообщения процесс увеличивает свою компоненту на единицу.
- При приеме сообщение берется покомпонентный максимум:

$$VC \leftarrow \max(VC, VC_r).$$

- После получения сообщения процесс увеличивает свою компоненту на единицу.

Свойства векторного времени:

- Векторное время уникально для каждого события.
- Векторное время полностью передает отношение произошло-до.
-

$$\forall e, f \in \mathbb{E}\text{: } \texttt{proc(e)} = P_i, \texttt{ proc(f)} = P_j \Longrightarrow \left(e \to f \iff \begin{pmatrix} VC(e)_i \\ VC(e)_j \end{pmatrix} < \begin{pmatrix} VC(f)_i \\ VC(f)_j \end{pmatrix} \right).$$

3 Формализм. Часы с прямой зависимостью (свойства и алгоритм)

Определение.

$$e \to_d f \iff e < f \vee \exists m \in \mathbb{M}\colon\ e \leqslant \texttt{snd(m)} \wedge \texttt{rcv(m)} \leqslant f.$$

Определение. *Часы с прямой зависимостью*. Определим функцию $VC_d\colon \mathbb{E} \to N^k$ так, чтобы

$$\forall e, f \in \mathbb{E}\colon\ e \to_d f \iff VC_d(e) < VC_d(f).$$

Алгоритм. (Часы с прямой зависимостью)
Алгоритм полностью повторяет алгоритм для векторных часов, за исключением того, что посылается только та компонента времени, которая соотвествует процессу-отправителю.

4 Взаимное исключение в распределенной системе. Централизованный алгоритм.

Обозначение	Объект
CS_i	Критическая секция с номером
$\texttt{Enter}(CS_i)$	Вход в критическую секцию
$\texttt{Exit}(CS_i)$	Выход из критической секции

Таблица 2: Общие обозначения

Определение. *Взаимное исключение*. Основное требование

$$\texttt{Exit}(CS_i) \rightarrow \texttt{Enter}(CS_{i+1}).$$

Определение. *Требование прогресса:*

- Каждое желание процесса попасть в критическую секцию будет рано или поздно удовлетворено.
- Может быть гарантирован тот или иной уровень честности удовлетворения желания процессов о входе в критическую секцию.

Алгоритм. (Централизованный алгоритм)

- Вся работа контролируется выделенным координатором.
- Общение происходит по следующему протоколу:

Вид запроса	Действие
`request`	Запрос разрешения у координатора
`ok`	Одобрение координатором входа в секцию
`release`	Освобождение пользователем критической секции

Таблица 3: Виды запросов

- При входе в критическую секцию процесс шлёт запрос координатору, дожидается разрешения, затем входит в критическую секцию. При завершении работы процесс посылает координатору сообщения, что секция свободна. Данный алгоритм всегда требует 3 сообщения для работы с критической секцией.
- Не масштабируется из-за необходимости иметь выделенного координатора.

5 Взаимное исключение в распределённой системе. Алгоритм Лампорта

Вид запроса	Действие
`request`	От запрашивающего процесса ко всем другим
`ok`	Подтверждение получения (не даёт права входа в CS)
`release`	После выхода из критической секции

Таблица 4: Виды запросов алгоритма Лампорта

Алгоритм. (Лампорт)

- Координатор отсутствует, но у процессов есть приоритет.
- Сообщения request и release рассылаются всем другим процессам, всего $3n-3$ сообщения на CS.
- Используются логические часы Лампорта. Для установления порядка "кто раньше". Обязательно требуется порядок FIFO на сообщениях.
- Все процессы хранят у себя очередь запросов.
- В критическую секцию можно войти, если
  - Мой запрос первый в очереди, т.е. его время меньше времени остальных запросов (при равенстве времен порядок определяется по приоритету процесса).
  - Получен ok от всех других процессов, т.е. они знают о вашем запросе.
- Если процесс хочет войти в CS, то он посылает всем другим процессам request со своими часами. Ждёт от всех ok. Входит в критическую секцию, если можно. Иначе ждёт release от всех процессов, которые раньше в очереди.

6 Взаимное исключение в распределённой системе. Алгоритм Рикарда и Агравалы

Вид запроса	Действие
`request`	От запрашивающего процесса ко всем другим
`ok`	Разрешение о входе в критическую секцию

Таблица 5: Виды запросов алгоритма Рикарда и Агравалы

Алгоритм. (Алгоритм Рикарда и Агравалы)

- Оптимизация алгоритма Лампорта.
- Всего $2n - 2$ сообщений.
- Если процесс хочет войти в CS, то он шлет request всем остальным процессам. Если процесс получивший запрос не хочет войти в CS, либо его номер запроса (в часах) больше, то он отсылает разрешение ok. Процесс, который входит в CS, хранит в очереди какие ok-ответы он должен послать после выхода.

Замечание. В отличие от алгоритма Лампорта, порядок сообщений FIFO не требуется.

7 Взаимное исключение в распределённой системе. Алгоритм обедающих философов.

Определение. В частном случае ресурсы – вилки, процессы – философы, граф конфликтов – кольцо.

Теорема 7.1. В ориентированном графе без циклов всегда есть исток.

Теорема 7.2. Если у истока перевернуть все ребра, то граф останется ациклическим.

Алгоритм. (Алгоритм обедающих философов)

- Философ владеет вилкой, если ребро в графе конфликтов исходит из его вершины.
- Философ может принять пищу, если владеет обеими вилками, т.е. он исток.
- После еды вилки надо отдать (ленивый способ):
  - После еды вилки помечаются грязными.
  - Моем вилки и отдаём их по запросу, даже если сами хотим есть.
  - Чистые вилки не отдаём, если сами хотим есть. Ожидаем все вилки, едим, отдаем, если был запрос.

Алгоритм. (Обобщение алгоритма обедающих философов на произвольный граф)

- Взаимное исключение эквивалентно полному графу конфликтов (ребро между каждой парой процессов).
- При инициализации вилки раздаются в каком-то порядке (например, по порядку id процессов).

Замечание. (Результат)

- 0 сообщений на повторный заход в критическую секцию.
- В худшем случае $2n - 2$ сообщения.
- Количество сообщений пропорционально числу желающих попасть в критическую секцию.
- Отсутствие *deadlock*, т.к. поддерживается ацикличность графа.

8 Алгоритм на основе токена.

Определение. Токен – некоторый объект, который даёт владельцу право на вход в критическую секцию.

Алгоритм. (Алгоритм на основе токена)

- В система существует один токен для конкретного ресурса (критической секции).
- Все узлы в системе объединены в кольцо.
- Токен пересылается по кругу, и каждый процесс делает следующее:
  - Если нет желания войти в критическую секцию, то пересылаем токен дальше.
  - Если желание есть, то входим (т.к. у нас уникальное право). После завершения передаем токен дальше.

Замечание. Количество сообщений в системе стабильно, но необходимо ждать, пока токен дойдет до тебя.

9 Взаимное исключение в распределённой системе. Алгоритмы на основе кворума (простое большинство, рушащиеся стены).

Определение. *Кворум:*

- Семейство подмножеств множества процессов $Q \subset 2^{\mathbb{P}}$.
- Любые два кворума имеют непустое пересечение:

$$\forall A, B \in Q: \; A \cap B \neq \emptyset$$

Примеры. Варианты кворумов:

- Централизованный алгоритм как частный случай кворума.
- Простое большинство (больше половины процессов) и взвешенное большинство.
- Рушащиеся стены.

Определение. *Кворум «рушащиеся стены»*

- Процессы образуют квадратную матрицу (приблизительно).
- Кворумом назовем набор процессов, состоящий из некоторого столбца целиком и представителей всех остальных столбцов.
- Заметим, что пересечение любых двух таких множеств непусто.
- Размер порядка $2\sqrt{n}$.

Замечание. При пересечении кворумов потенциально возможен *deadlock*. Решением служит *иерархическая блокировка*.

10 Согласованное глобальное состояние (согласованный срез). Алгоритм Чанди-Лампорта. Запоминание сообщений на стороне отправителя.

Определение. *Срезом* называется любое $G \subseteq E$, удовлетворяющее условию

$$\forall e \in E, f \in G \; e < f \Longrightarrow e \in G.$$

Определение. Срез G называется *согласованным*, если

$$\forall e \in E, f \in G \; e \to f \Longrightarrow e \in G.$$

Алгоритм. (Чанди, Лампорт)

- Сначала все процессы помечаются как белые (w).
- Процесс-инициатор запоминает свое состояние, помечается красным (r) и посылает токен всем соседям.
- При получении сообщения w-процесс запоминает свое состояние и становится красным, после чего посылает токен всем соседям.
- Запомненные состояния образуют согласованный срез.

Замечание. Алгоритм работает корректно только в случае, когда соблюдается FIFO порядок на сообщениях.

Замечание. (Классификация сообщений)

- ww-сообщения. Их не надо сохранять, состояние их уже учитывает.
- rr-сообщения. Их не надо сохранять, они просто сами произойдут потом.
- wr-сообщения. Такие сообщения нужно обязательно сохранять для дальнейшего восстановления состояния системы.
- rw-сообщения. Таких не может быть по определению согласованного среза.

Алгоритм. (Запоминание сообщений на стороне отправителя)

- w-процесс обязательно отправляет подтверждение на каждое полученное сообщение.
- Процесс-отправитель сохраняет только те сообщения, на которые не успело прийти подтверждение.
- r-процесс не отправляет токен-подтверждение, поэтому wr-сообщения и только они не удалятся из буфера.
- Буфер готов тогда, когда процесс становится красным. После этого отправляемые сообщения не являются wr.

11 Согласованное глобальное состояние (согласованный срез). Алгоритм Чанди-Лампорта. Запоминание сообщений на стороне получателя.

Первую часть вопроса см. в предыдущем билете.

Алгоритм. (Запоминание сообщений на стороне получателя)
Процесс P запоминает все сообщения от процесса Q, пришедшие в отрезок времени после того, как P стал красным, до того, как Q пришлет токен из алгоритма Чанди-Лампорта.

12 Глобальные свойства. Стабильные и нестабильные предикаты. Слабый конъюнктивный предикат. Централизованный алгоритм.

Определение. *Глобальным предикатом* называется предикат, определенный над состоянием системы в целом. Под состоянием системы подразумевается согласованный срез.

Определение. Предикат $P(G)$ называется стабильным, если для любых согласованных срезов G, H выполняется:

$$G \subset H \wedge P(G) \Longrightarrow P(H).$$

Алгоритм. (Простой алгоритм для стабильных предикатов)
Строим согласованный срез при помощи алгоритма Чанди-Лампорта, проверяем на нём выполненность предиката. Если он верен, то будет верен и в дальнейшем.

Определение. *Локальным* называется предикат, зависящий от состояния только одного процесса.

Замечание. Если глобальный предикат является дизъюнкцией локальных, то его предельно просто проверять даже без построения каких-либо срезов.

Определение. Предикат называется *слабым конъюнктивным*, если он верен хотя бы на одном согласованном срезе.

Теорема 12.1. Срез согласован тогда и только тогда, когда векторные времена процессов на этом срезе попарно несравнимы.

Алгоритм. (Централизованный алгоритм для слабого конъюнктивного предиката)

- Каждый процесс отслеживает свое векторное время.
- При наступлении истинности локального предиката, отправляем сообщение координатору (делая при этом все необходимые манипуляции со временем).
- Координатор поддерживает *срез-кандидат* и очередь необработанных сообщений.
  - Для каждой компоненты среза-кандидата координатор хранит флажок. Красный – элемент не может быть частью согласованного среза. Зеленый – может. Начальное состояние – нулевой вектор, все флажки красные.
  - Обрабатываем сообщения только от красных процессов; от зеленых сообщения идут в очередь.
  - Сравниваем пришедший вектор попарно с другими процессами (достаточно сравнить только две соответствующие компоненты). Если нарушилась согласованность (новый вектор оказался больше), то делаем меньший процесс красным. После обработки делаем процесс зеленым.

– Как только все флажки стали зелеными, найден согласованный срез.

Теорема 12.2. (Корректность)

- Алгоритм никогда не пропустит согласованный срез. Действительно, пусть есть согласованный срез. В каком-то порядке процессы дойдут до момента истинности предиката, после чего пошлют сообщения координатору. Ни одно из этих сообщений не может сделать другой процесс красным (если он стал зеленым после обработки сообщения их этого среза), потому что срез согласован. Поэтому все процессы станут зелеными сразу после обработки соответствующих сообщений.

- Компонента согласованного среза становится зеленой и всегда будет такой оставаться.

13 Слабый конъюнктивный предикат. Распределенный алгоритм.

Первую часть вопроса см. в предыдущем билете.

Алгоритм. (Распределенный алгоритм для слабого конъюнктивного предиката)

- Каждый процесс имеет своего собственного координатора.
- Процессы шлют сообщения своим координаторам. Координаторы общаются между собой, пересылая друг другу срезы-кандидаты с флажками.
- Красные координаторы обрабатывают сообщения от своих процессов. После обработки, координатор становится зеленым. Если другой процесс был помечем красным, то соответствующее сообщение шлется нужному координатору.

14 Диффундирующие вычисления, пример. Останов. Алгоритм Дейксты и Шолтена.

Определение. *Диффундирующим* называется вычисление, для которого верно:

- Процессы бывают в двух состояниях: активный и пассивный.
- Получение сообщения делает процесс активным.
- Посылать сообщения могут только активные процессы.
- Активный процесс в любой момент может стать пассивным.
- Алгоритм начинается с одного активного процесса-инициатора.

Пример. Алгоритм Дейкстры – пример диффундирующего вычисления.

Определение. Диффундирующее вычисление завершилось, если все процессы пассивны и нет сообщений в пути.

Определение. *Проблема останова* – как процессу-инициатору узнать, когда алгоритм завершился?

Алгоритм. (Дейкстра, Шолтен. Останов диффундирующего вычисления)

- Все процессы будут выстраиваться в дерево.
- На все сообщения требуются подтверждения.
- Каждый процесс знает своего предка в дереве, число своих детей и разницу между числом отправленных сообщений, и сообщений, на которые было получено подтверждение.
- *Зеленым* назовем пассивный процесс без детей и неподтвержденных сообщений. В противном случае, процесс считается красным. Дерево состоит из красных процессов.
- При получении сообщения, зеленый процесс становится красным, делая родителем отправителя сообщения и высылая тому подтверждение. После получения подтверждения отправитель увеличивает счетчик детей.
- Аналогично, как только процесс становится зеленым, он удаляет себя из дерева, посылая предку соответствующее сообщение.
- Вычисление остановилось, как только корень дерева (то есть, инициатор), становится зеленым.

15 Локально-стабильные предикаты: согласованные интервалы, барьерная синхронизация (3 алгоритма). Применение для определения взаимной блокировки (deadlock).

Определение. Пара срезов $F, G \subseteq E$ называется *интервалом* $[F, G]$, если $F \subseteq G$.

Определение. Интервал $[F, G]$ называется согласованным, если

$$\forall e \in E, f \in F \; e \to f \Longrightarrow e \in G.$$

Замечание. Интервал $[G, G]$ согласован тогда и только тогда, когда G – согласованный срез.

Теорема 15.1. Интервал $[F, G]$ согласован тогда и только тогда, когда существует согласванный срез H такой, что $F \subseteq H \subseteq G$.

Определение. Интервал $[F, G]$ называется *барьерно-синхронизированным*, если

$$\forall f \in F, g \in E \setminus G \; f \to g.$$

Теорема 15.2. Любой барьерно-синхронизированный интервал согласован.

Алгоритм. (Алгоритмы построения барьерной синхронизации)

- Построение через координатора. Каждый процесс посылает координатору сообщение. Когда координатор получил сообщение от *всех*, он посылает всем сообщение. Срезы для интервала: по посылке сообщений процессами и по приему сообщений от координатора.
- Посылка каждый каждому.
- Посылка токена два раза по кругу.

Определение. *Локально-стабильным* называется стабильный предикат, определяемый группой процессов с неизменным состоянием.

Пример. Взаимная блокировка – пример локально-стабильного предиката. Для проверки такого предиката необходим согласованный срез. Для этого воспользуемся барьерной синхронизацией $[F, G]$. На срезе F процессы пошлют координатору информацию, ожидают ли они другой процесс. После этого каждый процесс будет помнить, менялось ли у него состояние (относительно блокировки). На срезе G процессы сообщат координатору, поменялось ли их состояние. Если на момент G состояние у процессов не менялось и на момент F была зафиксирована взаимная блокировка, то она в действительности есть.

16 Упорядочение сообщений. Определения, иерархия порядков. Алгоритм для FIFO.

Определение. Иерархия порядков сообщений.
Для обычных сообщений между парой процессов:

- Асинхронная передача (нет порядка).
- FIFO.
- Причинно-согласованный порядок.
- Синхронный порядок.

Для массовой рассылки сообщений:

- Общий порядок (total order).

Определение. Говорят, что соблюдается порядок *FIFO (First In First Out)*, если

$$\nexists m, n \in \mathbb{M}\colon\ \texttt{snd(m)} < \texttt{snd(n)} \wedge \texttt{rcv(n)} < \texttt{rcv(m)}.$$

Замечание. Под $<$ подразумевается отношение порядка в одном процессе.

Определение. Говорят, что соблюдается *причинно-согласованный порядок*, если

$$\nexists m, n \in \mathbb{M}\colon\ \texttt{snd(m)} \to \texttt{snd(n)} \wedge \texttt{rcv(n)} \to \texttt{rcv(m)}.$$

Замечание. Под $\to$ подразумевается отношение “произошло до“.

Определение. Порядок называется *синхронным*, если всем сообщениям можно сопоставить время $T(m)$ так, что $T(\texttt{snd}(m)) = T(\texttt{rcv}(m)) = T(m)$ и

$$\forall e, f \in \mathbb{E}\colon\ e \to f \Longrightarrow T(e) < T(f).$$

Определение. Пусть в системе сообщения рассылаются нескольким получателям. Обозначим $\texttt{rcv}_p(m)$ – события получения сообщения процессами $p \in \mathbb{P}$. Будем говорить, что соблюдается *общий порядок*, если

$$\nexists m, n \in \mathbb{M},\ p, q \in \mathbb{P}\colon\ \texttt{rcv}_p(m) < \texttt{rcv}_p(n) \wedge \texttt{rcv}_q(n) < \texttt{rcv}_q(m).$$

Замечание. Для случая, когда сообщения отправляются только одному процессу, это *общий порядок* всегда выполняется.

Алгоритм. (FIFO)
Алгоритм для восстановления FIFO порядка сообщений основан на их нумерации.
Рассмотрим взаимодействие двух процессов:

- Нумеруем все сообщения в порядке отправки.
- Получатель поддерживает номер ожидаемого сообщения.
- Получатель обрабатывает пришедшее сообщение, если его номер совпал с ожидаемым.
- Если номер сообщения не совпал с ожидаемым, то сообщение складывается в очередь и обрабатывается, когда его номер становится равным ожидаемому номеру.

17 Упорядочение сообщений. Определения, иерархия порядков. Алгоритм для причинно-согласованного порядка.

Первую часть вопроса см. в билете 16.

Алгоритм. (Централизованный алгоритм для причинно-согласованного порядка)

- Передача сообщений через координатора.
- Для корректности необходимо, чтобы каналы до координатора имели FIFO порядок сообщений.

Алгоритм. (Распределенный алгоритм для причинно-согласованного порядка)

- Используем матричные часы:
  - У каждого процесса хранится матрица M. M_{ij} – количество сообщений, посланных от процесса P_i к процессу P_j.
  - Перед посылкой сообщения от P_i к P_j обновляем $M_{ij} = M_{ij} + 1$ и шлем матрицу M вместе с сообщением.
- Если процесс P_i получил сообщение от P_j с матрицей W, то оно обработается, если соблюдаются условия:
  - FIFO порядок: $W_{ji} = M_{ji} + 1$. Сообщение имеет ожидаемый номер.
  - Причинная согласованность: $\forall k \neq j\colon\ M_{ki} \geqslant W_{ki}$. То есть посылающий процесс не знает о событиях, о которых не знает принимающий.
- При невыполнении хотя бы одного из условий сообщение кладется в очередь.
- После обработки обновляется матрица принимающего процесса через покомпонентный максимум: $M = \max(M, W)$.

18 Упорядочение сообщений. Определения, иерархия порядков. Алгоритм для синхронного порядка.

Первую часть вопроса см. в билете 16.

Пример. Нарушение синхронного порядка Перекрестные сообщения между двумя процессами.

Алгоритм. (Централизованный алгоритм для синхронного порядка)

- Передача сообщений через координатора.
- Координатор дожидается подтверждения, что сообщение доставлено и только после этого может послать новое.
- Для корректности необходимо, чтобы каналы до координатора имели FIFO порядок сообщений.

Алгоритм. (Распределенный алгоритм для синхронного порядка)
Алгоритм основан на иерархии процессов, они упорядочены по приоритету.

- Пусть у процесса P приоритет больше, чем у процесса Q.
- P шлёт сообщение Q с подтверждением получения. Пока подтвержение не получено, процесс пассивен.
- Q шлет сообщение P только после получения разрешения, то есть подтверждения возможности посылки. P между отправкой подтверждения и получением сообщения.
- Пассивный процесс не участвует в обработке и посылке сообщений, поэтому всегда можно выбрать момент передачи сообщения $T(m)$ в его промежутке пассивности.
- Не может произойти взаимная блокировка, потому что пассивным всегда становится процесс с бОльшим приоритетом.
- Независимые пары процессов могут общаться независимо.

19 Общий порядок (total order). Алгоритм Лампорта.

Первую часть вопроса см. в билете 16.

Замечание.

- *Broadcast* – всем другим процессам.
- *Multicast* – подмножеству процессов.

Алгоритм. (Централизованный алгоритм обеспечения общего порядка)
Пусть в системе соблюдается FIFO порядок сообщений. Если процесс хочет сделать рассылку сообщения, он сообщает об этом координатору, который в свою очередь рассылает сообщения в фиксированном порядке.

Замечание. Централизованный алгоритм также обеспечивает причинно-согласованный порядок.

Алгоритм. (Лампорт)
Обобщим алгоритм Лампорта для взаимной блокировки.

- Необходимо соблюдение FIFO порядка сообщений.
- Все multicast-сообщения придется заменить на broadcast.
- Перед отправкой сообщения процесс берет “билет”, соответствующий его логическому времени, и посылает `request` запрос всем другим процессам.
- Процессы отвечают ему `ok`. Отправитель начинает рассылку после `ok` от всех процессов.
- Порядок обработки сообщений определяется парой из билета и номера процесса.

20 Общий порядок (total order). Алгоритм Скина.

Первую часть вопроса см. в билете 16.

Алгоритм. (Скин)
Модифицируем алгоритм Лампорта. Для этого алгоритма не требуется FIFO порядок на сообщениях, и он умеет делать multicast-сообщения.

- Пусть процесс хочет сделать рассылку. В таком случае, он шлёт всем кому надо `request`, приписывая к нему свое логическое время в качестве *предварительного* билета.
- Все процессы, как и в алгоритме Лампорта, имеют очередь сообщений, приоритетную по билетам.
- Если обрабатываемое сообщение – запрос на рассылку, то автору отправляется `ok`.
- Как только процесс получает все `ok`, он отправляет настоящие сообщения, приписывая к ним текущее логическое время в качестве *финального* билета. Эти сообщения обрабатываются другими процессами в общем порядке, приоритетном по номеру билета.
- Сообщения из рассылки обрабатываются процессами как только доходят до начала очереди.

21 Иерархия ошибок в распределенных системах. Отказ узла в асинхронной системе — невозможность консенсуса (доказательство Фишера-Линча-Патерсона).

Определение. Иерархия ошибок и отказов в распределенных системах

- Отказ узла (самый простой).
- Отказ канала (равносильно отказу всех узлов).
- Ненадежная доставка (некоторые сообщения не доходят).
- Византийская ошибка (враги захватили узел).

Замечание. Стоит отметить, что именно они делают программирование распределенных систем довольно сложным, так как являются нормой. При частичном отказе системы остальные части должны продолжать работать.
Для корректного решения частичного отказа стоит сначала определить виды систем:

- Синхронные системы:
  - Время передачи сообщения ограничено сверху.
  - Можно разбить выполнение алгоритма на фазы.
- Асинхронные системы:
  - Время не ограничено.
  - Время передачи конечно, если нет отказов.

Определение. Рассмотрим свойства консенсуса в распределённой системе:

- Согласие – все процессы должны завершиться с одним и тем же решением.
- Нетривиальность – должны быть варианты исполнения, приводящие к разным решениям.
- Обоснованность – решение должно быть предложением одного из процессов.
- Завершение – протокол должен завершиться за конечное время.

Замечание. Достичь этих свойств без отказа легко:

- Каждый процесс шлет свое предложение всем остальным.
- Дожидается предложения от других процессов.
- Теперь из данных предложений, используя детерминированную функцию (max, min, etc), выбирает решение.

Замечание. Этот алгоритм работает и в асинхронной системе.

Теорема 21.1. (FLP) Не существует такого детерминированного алгоритма, который при любом исполнении за конечное время придет к консенсусу в асинхронной системе.

Доказательство. Допустим такой алгоритм существует, тогда рассмотрим его исполнении на множестве из 0 и 1. Модель системы для теоремы:

Определение. *Процесс* – это некоторый детерминированный автомат, который может делать 3 вещи:

- Ожидать сообщения (без ограничения по времени).
- Отправлять сообщения.
- Принять решение (можно только 1 раз это сделать, но при этом сообщать свое решение другим алгоритмам разрешено).

Определение. *Конфигурация* — состояние всех процессов и сообщения в пути (отправленные + неполученные).

Определение. *Шагом* в такой конфигурации называется:

- Обработка какого-то сообщения процессом.
- Внутреннее действие этого процесса и посылка им от нуля до нескольких сообщений до тех пор, пока процесс не перейдет к ожиданию следующего сообщения. Это называется событием, оно характеризуется процессом, прочитавшем сообщение, и самим сообщением.

Так как все операции детерминированы, можно нарисовать полное дерево переходов. Начальная конфигурация содержит начальные данные для каждого из процессов.

- Может содержать сколько угодно входных данных.
- Начальных конфигураций много (на каждый вариант входных данных).
- Каждый процесс может иметь свою программу.

Определение. *Исполнение* – бесконечная цепочка шагов от начального состояния, так как процессы продолжают выполняться и после принятия решения.

Определение. *Отказ* – ситуация, когда процесс делает конечное число действий в процессе исполнения.

Определение. *Надежная доставка* – любое сообщение неотказавшего процесса обрабатывается за конечное число шагов.

Определение. *Согласие и решение* – все процессы должны прийти к решению за конечное число шагов (кроме возможно отказавшего).

Определение. *Валентность*

- Конфигурация i-валентная, если все цепочки шагов приводят к решению i.

- Бивалентная – если есть цепочки, приводящие и к 0, и к 1.

Определение. *Коммутирующие события* — это цепочки с событиями на разных процессах, которые приводят к одной и той же конфигурации, при изменении порядка их исполнения.

Лемма 21.2. Существует начальная бивалентная конфигурация.

Доказательство.

- Предположим, такой конфигурации нет. Тогда все конфигурации одновалентны: есть конфигурации, которые всегда приводят только к 0, есть приводящие только к 1.
- При этом, обязательно должна быть как 0-валентная, так и 1-валентная конфигурации: иначе нарушается нетривиальность консенсуса.
- Возьмем начальные конфигурации, приводящие к 0 и к 1, начнем по очереди их заменять, чтобы найти пару конфигураций разной валентности, отличающиеся начальным состоянием только одного процесса. Так можно сделать, потому что любые два состояния можно соединить цепочкой, в которой каждое изменение применяется только к начальным данным одного процесса. Где-то в этой цепочки и будет смена валентности за 1 шаг.
- Тогда пусть этот процесс откажет сразу, тогда его начальное состояние ни на что не влияет. Получается, что состояние бивалентное. Противоречие.

■

Лемма 21.3. Для бивалентной конфигурации можно всегда найти следующую за ней бивалентную.

Доказательство.

- Рассмотрим конфигурацию G. Пусть e – произвольное событие (процесс p, сообщение m). Определим C как множество конфигураций, достижимых из G без применения события e, а D – как множество конфигураций, достижимых из C одним применением e: $D = e(C)$.
- Предположим, что в D нет бивалентных конфигураций.
- Покажем, что в D есть как 0-, так и 1-валентные конфигурации.
  - G бивалентна, поэтому из нее достижима i-валентная конфигурация E_i. Рассмотрим случаи:
    - $E_i \in D$. Тогда эта конфигурация нам и подходит.
    - $E_i \in C$. Тогда нам подходит конфигурация $e(E_i)$. (из i-валентной конфигурации достижимы только i-валентные).

* E_i Не лежит ни там, ни там. Значит, она достижима из G по какой-то цепочке, которая сначала проходит в C, затем применяет e, после чего применяет еще какие-то события. Возьмем F_i – конфигурацию, котороая получилась сразу на выходе из C после применения e. Она нам подойдет.

- Зафиксируем валентность $e(G)$. Найдем какую-нибудь D_j с другой валентностью. Тогда между ними (они упорядочены так же, как и соответствующие им C_i) найдется пара D_i, D_{i+1}, с разными валентностями. Ей соответсвует пара C_i, D_{i+1}. Поскольку мы предположили, что из G не достижимы бивалентные конфигурации, валентности C_k и D_k совпадают. Не умаляя общности, положим валентность C_i, D_i равной нулю, валентность C_{i+1}, D_{i+1} равной единице. Так же, заменим i на 0

- Пусть переход от C_0 к C_1 произошел при обработке сообщения f.

  – Если f и e произошли на разных процессах, то цепочки

  $$C_0 \to_f C_1 \to_e D_1,\ C_0 \to_e D_0 \to_f D_1,$$

  очевидно, коммутируют, из чего следует бивалентность C_0. Противоречие.

  – Пусть f и e произошли на одном процессе. Предположим, что в состоянии C_0 этот процесс отказал, после чего система пришла к консенсусу (то есть, i-валентной конфигурации, где процессы сделали выбор) применив цепочку σ. Обозначим $A = \sigma(C_0)$. Так как в состоянии A уже есть консенсус, A одновалентно. Положим $E_0 = \sigma(D_0), E_1 = \sigma(D_1)$. Рассмотрим так же альтернативные цепочки:

  $$C_0 \to_e D_0 \to_\sigma E_0,\ C_0 \to_f C_1 \to_e D_1 \to_\sigma E_1.$$

  Из того, что A одновалентно, следует, что E_0 и E_1 одновалентны, причем имеют такую же валетность, как и A, так как достижимы из A. Но $E_0 = \sigma(D_0)$ – 0-валентная, $E_1 = \sigma(D_1)$ – 1-валентная. Противоречие.

■

Значит всегда есть переход в бивалентное состояние, то есть существует бесконечное исполнение, посещающее только бивалентные конфигурации, то есть не приводящая к консенсусу. ■

22 Консенсус в распределенных системах. Применение консенсуса: выбор лидера, terminating reliable broadcast.

Определение. *TRB* – это гарантия получения сообщения всеми процессами (или все получат, или никто не получит).

Алгоритм. (TRB эквивалентен консенсусу)

$\Longrightarrow$ Каждый процесс делает TRB своего предложения и приходит к консенсусу, используя детерминированную функцию.

$\Longleftarrow$ Рассылаем сообщение в любом порядке. С помощью консенсуса на одном бите решаем нужно ли обрабатывать сообщение или нет.

Определение. *Выбор лидера* – это задача выбора из множества процессов одного лидера за конечное время.

Алгоритм. (Выбор лидера эквивалентен консенсусу)

$\Longrightarrow$ Выбираем лидера, его предложение и есть консенсус.

$\Longleftarrow$ Каждый процесс предлагает себя в качестве лидера, а алгоритм консенсуса определяет выбор лидера.

23 Синхронные системы. Алгоритм для консенсуса в случае отказа заданного числа узлов.

Из 21 билета. Нельзя прийти к консенсусу, если все 4 свойства системы верны.

Алгоритм. (Консенсус в синхронной сети с отказами узлов)

- Пусть могут отказать f узлов ($0 \leq f < N$). Если отказывают все, то кому работать.
- Делаем $f+1$ фазу базового алгоритма (рассылаем известные множества предложений, где одна фаза – это максимальное время доставки сообщения).
- На каждой фазе каждый процесс рассылает остальным множество известных ему предложений.
- Алгоритм корректен по принципу Дирихле. Хотя бы на одной из $f+1$ фаз ни один процесс не откажет, тогда сразу у всех процессов множество предложений станет одинаковым.
- Можно приходить к решению, применяя детерминированную функцию к этому множеству.

24 Синхронные системы. Проблема византийских генералов. Алгоритм для N >= 4, f = 1. Объяснить идею обобщения для f > 1.

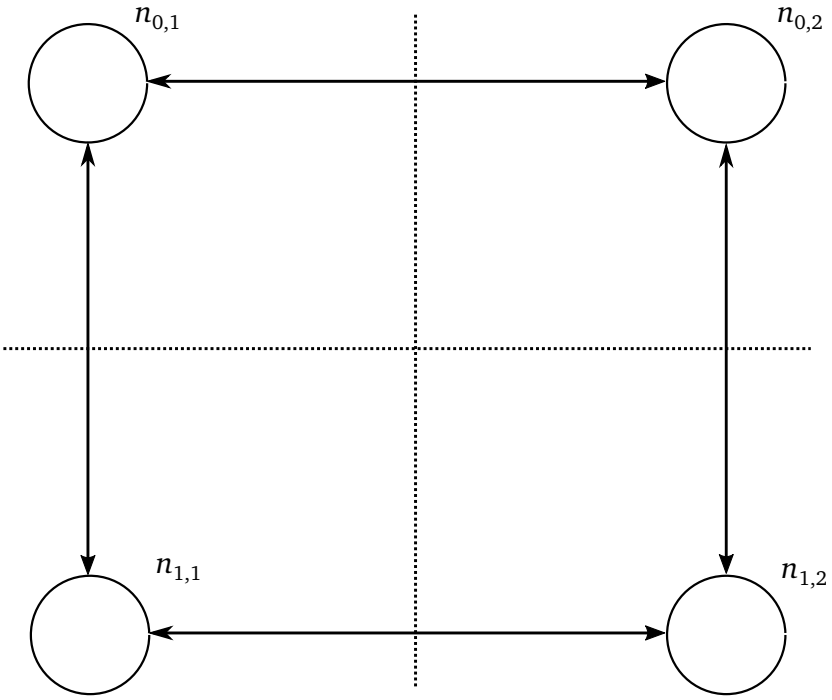
Определение. *Проблема византийских генералов* - прийти к консенсусу, штурмовать или не штурмовать крепость, но из N человек, есть f предадетелей.

Теорема 24.1. Решение *проблемы византийских генералов* возможно в синхронной системе только если $N > 3f$.

Алгоритм.

- Все процессы шлют свои предложения.
- Все процессы пересылают всю полученную информацию всем другим процессам.
- Если больше 1 генерала-предателя, то дополнительно пересылаем матрицу ответов, куб, гиперкуб и так далее (на каждого генерала по размерности).
- Теперь у каждого процесса есть матрица информации от каждого процесса.
- Для четырех процессов в матрице испорчена одна строка и столбец. Так как матрица 3 на 3 (без диагонали) в каждой строчке можно определить истинное значение предложение процесса, просто посчитав самое частое значение в строке.
- То есть три несбойных процесса имеют одни и те же 4 числа и могут прийти к консенсусу.
- Предатель не может помешать прийти к консенсусу, но может повлиять на то какое решение будет принято.

25 Синхронные системы. Проблема византийских генералов. Невозможность решения при N = 3, f = 1.



Теорема 25.1. (Невозможность решения задачи византийских генералов при $N \leqslant 3f$)

- Докажем теорему для случая $N = 3$, $f = 1$ и обоснованного консенсуса.
- Предположим, существует алгоритм, способный решить эту задачу. Возьмем четыре узла, соединим их как на рисунке, и запустим, дав на вход $n_{0,1}$, $n_{0,2}$ нули как предлагаемые значения, и единицы $n_{1,1}$, $n_{1,2}$.
- Рассмотрим узлы $n_{0,1}$, $n_{0,2}$. Представим себе, что $n_{1,1}$ и $n_{1,2}$ – на самом деле один узел, посылающий странные сообщения. Тогда $n_{0,1}$, $n_{0,2}$ из обоснованности должны прийти к консенсусу 0. Аналогично, $n_{1,1}$, $n_{1,2}$ придут к консенсусу 1.
- Если мы теперь посмотрим на эту же самую систему с точки зрения $n_{0,1}$, $n_{1,1}$ и $n_{0,2}$, $n_{1,2}$ (все аналогично, просто мы теперь считаем, что процессы кооперируют по-другому), то получим, что в обеих парах консенсус не достигнут.

26 Недетерминированные алгоритмы консенсуса. Алгоритм Бен-Ора.

Замечание. Невозможность построения алгоритма консенсуса при возможности отказа узла доказывается только в случае выполнения следующих свойств (теорема FLP, билет 21):

- Система асинхронная.
- Алгоритм детерминированный.
- Конечное время достижения консенсуса.

Избавимся от второго требования.

Замечание. К недетерминированным алгоритмам консенсуса предъявим требования:

- Консенсус достигается с вероятностью 1.
- Порядок исполнения операций выбирает “противник”.

Алгоритм. (Бен-Ор)
Алгоритм для бинарного консенсуса в системе с N процессами, отказать могут $f < N/2$.

- Будет множество раундов. Каждый раунд состоит из двух фаз.
- На каждой фазе процесс будет слать N сообщений и ждать $N - f$ ответов.
- В первой фазе процесс рассылает свое предпочтение: $(1, k, p)$. Здесь k – номер раунда, единица означает первую фазу, p – препочтение.
  - Процесс считает голоса, пришедшие от других процессов. Если какое-то значение набрало больше $N/2$ голосов, то оно *ратифицирует*.
  - Во второй фазе процесс шлет сообщения $(2, k, v)$ – где v – ратифицированное значение или ?, если его нет.
  - После того, как процесс ратифицировал или получил ратификацию во второй фазе, он меняет свое предпочтение на v.
  - Получив больше f ратификаций процесс принимает решение v, продолжая при этом исполняться.
  - Не получив ратификации, процесс меняет свое предпочтение на случайное.

Лемма 26.1. В одном раунде процессы не могут ратифицировать разные значения.

Лемма 26.2. Если процесс принял решение v, то в следующем раунде все процессы начнут с предпочтением v.

Доказательство.

- Чтобы принять решение, процесс получил минимум $f + 1$ сообщений вида $(2, k, v)$. Подобных сообщений с другим v быть не могло по предыдущей лемме.
- Чтобы начать раунд с другим предпочтением процесс должен был получить $N - f$ сообщений вида $(2, k, ?)$.
- Эти сообщения, очевидно, посланы разными узлами. Но тогда

$$(N - f) + (f + 1) = N + 1 > f.$$

Противоречие.

■

Замечание. (Об алгоритме Бен-Ора)

- Чтобы алгоритм все-таки заканчивался, нужно рассылать еще третий тип сообщения “решение”. Для корректности это не обязательно: за конечное число шагов все равно все примут решение.
- Система асинхронная, то есть сообщения не обязаны приходить раунд за раундом. Но поскольку мы ждем $N - f$ сообщений в каждой фазе, алгоритм получается “почти асинхронный”.
- Даже если сильный противник знает все о состоянии системы, вероятность завершения алгоритма за конечное число шагов равна единице.
- Ожидаемое время достижения консенсуса $\mathcal{O}(2^N)$, так как на каждом раунде все процессы начнут с одинаковым предпочтением с вероятностью, не меньшей 2^{-N}.

27 Paxos. Алгоритм, его свойства.

Алгоритм. (Paxos [Лампорт, 1989])

- Это – первый практически применимый алгоритм асинхронного консенсуса.
- Каждый процесс выбирает значение из множества предложенных.
- Алгоритм гарантирует согласие при любых отказах и при произвольных задержках сообщений.
- По теореме FLP, алгоритм не может гарантировать завершения за конечное время. Но в случаях, когда ошибки случаются нечасто, консенсус достигается за конечное число шагов.

В системе будет три вида процессов.

- Решаем задачу однократного консенсуса.
- Есть множество предлагающих процессов (proposers). Например, они пытаются выполнить какую-то операцию RSM, и предлагают свою в качестве следующей.
- Принимающих решение процессов (acceptors) в системе будет несколько. Если сделать один принимающий процесс, то система будет слишком уязвимой к отказам.
- Есть также множество узнающих процессов (learners), которые могут совсем не совпадать ли с предлагающими, ни с принимающими решение.

Будем строить алгоритм на основе кворума.

- Кворум и множество предлагающих процессов заранее зафиксированы.
- Можно использовать любой кворум.
- Кворумы используются потому, что в такой схеме отказ какого-либо процесса не остановит работу.
- Предполагается, что отказы временные.

Среди множества принимающих процессов будет лидер. Предлагающие процессы должны знать (заранее фиксированное) множество принимающих процессов и кто из них лидер (лидер будет меняться).

- Выбор лидера – тоже задача консенсуса. Поэтому выбирать его будем за конечное время без гарантии того, что лидер получится один.
- Алгоритм все равно будет гарантировать согласие. Но гарантии завершения не будет до тех пор, пока лидеров несколько.

Основа алгоритма.

- Для прихода к консенсусу алгоритм делает один или несколько ранудов голосования.
- Раунд голосования инициируется лидером. Все предложения высылаются ему, он же их ставит на голосование.
- Раундов может быть несколько только если лидер не один.
- Несколько голосований может происходить одновременно. Вся структура алгоритма построена так, чтобы согласие все равно было обеспечено.
- Каждое голосование имеет свой уникльный номер. При отсутствии прогресса, лидер может пересоздать голосование с новым номером.

1-я фаза голосования.

1a Подготовка. Лидер инициирует голосование и рассылает кворуму принимающих сообщение $(1a, k)$. k – номер голосования.

1b Обещание. Получив сообщение $(1a, k)$, принимающий обещает не принимать предложения с меньшим номером. Далее он отвечает

- $(1b, k, ack, k', v')$, где (k', v') – информация о принятом предложнии с максимальным номером $k' < k$, или $k' = 0$, если ничего еще не было принято.
- $(1b, k'', nack)$, если уже было дано другое обещание с $k'' > k$. Лидер ответит $(1a, k'')$.

2-я фаза голосования.

2a Запрос. Лидер, получив обещания (ab, k, ack, k', v') от кворума принимающих, предлагает значение.

- Берет значение v' для наибольшего k', полученного от принимающих, или предлагает свое значение, если все $k' = 0$.
- Посылает $(2a, k, v)$ кворуму принимающих.
- На второй фазе можно использовать другой кворум.

2b Подтверждение. Если принимающий получает запрос $(2a, k, v)$, и он не давал обещания для $k' > k$, то он принимает предложение (k, v) и посылает сообщение $(2b, k, v)$ всем узнающим.

2c Узнающий, получив сообщение $(2b, k, v)$ от кворума принимающих, узнает о том, что принято значение v.

Теорема 27.1. (Корректность Paxos)
Если есть два принятых предложения (k, v), (k', v'), то $v = v'$.

Доказательство. Предположим противное, и $v \neq v'$. Без потери общности, будем считать, что $k < k'$ и k' такой наименьший.

- Внутри одного голосования не может быть принято два разных значения, потому что лидер этого голосования выставил ровно одно значение.
- Несогласованность может быть вызвана только разными голосованиями (с разными лидерами).
- Предположим, что есть два параллельно идущих голосования. Поскольку мы используем кворум, есть хотя бы один принимающий, который знает про оба голосования. Именно этот принимающий и не даст принять разные значения.

■

28 Paxos. Общие принципы (RSM, концепции). Основные модификации (fast paxos и multi paxos).

Первую часть вопроса см. в предыдущем билете.

Определение. *Replicated State Machine*. Пусть есть некоторое состояние, которое нужно хранить и менять, и которое нужно защитить от сбоя узла. Для надежности можно держать несколько копий этого состояния на разных машинах.

- Если операции не коммутируют между собой, то независимо применять изменения на разных узлах нельзя.
- Поэтому, нужно приходить к консенсусу по вопросу упорядочивания операций. В том числе для этого используется Paxos.

Замечание. (Модификации Paxos)

- **Multi Paxos**. Заметим, что лидер может проделать первую фазу сразу для нескольких голосований. После этого можно быстро делать вторую фазу для всех запусков. Между предлагающим и узнающим 3 передачи сообщений.
- **Fast Paxos**. Можно еще сильнее сократить задержку. Можно предлагать значение сразу принимающим, в случае, если предлагающий знает, кто лидер. Получается задержка в 2 сообщения, при отсутствии коллизий. По количеству сообщений может получиться хуже, потому что нужно посылать сразу кворуму принимающих.
- **Dynamic Paxos**. Модификация с изменяемым набором серверов. Смена списков принимающих становится одной из операций для RSM. Основные проблемы находятся на стыке кворумов, нужно чтобы кворумы старых и новых процессов были согласны.
- **Cheap Paxos**. Экономим сообщения. Будем посылать не всем принимающим, а только $f+1$ процессу. Остальные будут запасными, и использоваться только в случае отказов.
- **Stoppable, Byzantine Paxos**.

29 Шардирование. Общий принцип. Статическое отображение, остаток от деления, расширяемое побитовое хеширование, битовый бор, постоянное число секций.

Определение. *Шардирование.*

- Узлы распределенной системы хранят непересекающиеся подмножества данных.
- Пока что, система не поддерживает запросы, относящиеся к данным, расположенным сразу на нескольких серверах.
- Рассматриваем простейшие запросы по ключу (`get`, `set`, `cas`).
- Клиенты должны знать, как понять, на каком сервере хранится ключ. Хранить это отображение в явном виде не получится, так как его придется хранить на отдельном сервере.

Алгоритм. (Шардирование статическим отображением)
Зафиксируем множество узлов, построим отображение ключей на эти узлы. Это отображение меняться не будет, поэтому пусть каждый процесс знает его. *Плюсы*:

- Легко реализовать.

Минусы:

- Неравномерное распределение ключей.
- Фиксированное множество узлов.

Алгоритм. (Остаток от деления)
Построим такое отображение ключей в номера узлов:

$$\texttt{node\_id} \leftarrow \texttt{hash}(key) \mod N,$$

где N – число узлов. *Плюсы*:

- Переменное число узлов.
- Простота реализации.

Минусы:

- $\Theta(N)$ перемещений данных при добавлении или удалении узла.

Алгоритм. (Расширяемое побитовое хеширование)
Пусть число серверов всегда равно 2^m (m меняется). Тогда сделаем отображение, в котором номером сервера для конкретного ключа будет число, полученное из первых m бит его хеша. При добавлении узла, докупается столько же узлов, сколько было. При этом каждый сервер отдаст примерно половину своих данных новому серверу.

Алгоритм. (Битовый бор)
Отображение будет построено на боре, алфавит которого состоит из нуля и единицы. Листья бора соответствуют узлам, на которых хранятся ключи с префиксом хеша, равным строке от корня бора до листа.

- При добавлении узла, расщепляем переполненный лист на два, передавая на новый узел половину данных.
- При удалении узла:
  - Если брат – тоже лист, просто передаем свои ключи ему.
  - Если брат – не лист, то заменяем все поддерево родителя одним листом.

Минусы:

- При добавлении или удалении узла происходят скачки нагрузки.

Алгоритм. (Постоянное число секций)

- Заводим S секции, не меняем их число.
- Выбираем способ отображения ключа в номер секции.
- Храним на главном сервере информацию о том, где какая секция лежит.
- При добавлении или удалении узла перемещаем данные секциями.

Замечание. (О секциях)

- Секций должно быть не очень много, чтобы информацию от отображении секций на сервера можно было бы поместить на один сервер. При этом их должно быть на несколько порядков больше числа серверов, чтобы в дальнейшем можно было маштабироваться горизонтально.
- Можно делать балансировку нагрузки, определяя загруженные сервера, горячие данные и т.п. Для этого тоже полезно, чтобы секций было много.

30 Шардирование. Общий принцип, хеширование рандеву, консистентное хеширование, Multi-Probe Consistent Hashing.

Первую часть вопроса см. в предыдущем билете.

Алгоритм. (Rendezvouz hashing)
Зафиксируем число K. На основе "хорошей" хеш-функции построим следующее отображение:

$$\texttt{node\_id} \leftarrow \operatorname{argmax}_{i=0}^{K-1}(h(key \mid i)).$$

- При добавлении узла каждый узел перемещает только те ключи, которые должны перейти на новый узел:

$$h(key \mid \texttt{new\_node\_id}) > h(key \mid \texttt{cur\_node\_id}).$$

- При удалении узла перемещаются только ключи с этого узла.
- Поиск узла по ключу осуществляется за $\mathcal{O}(K)$.

Алгоритм. (Consistent Hashing)
Рассмотрим возможные значения хеш-функции как точки кольца. Разместим на этом кольце все сервера. Ключ будет лежать на том сервере, который находится ближе всего по часовой стрелке.

- При добавлении узла, ключи перемещаются только на новый узел.
- При удалении узла, ключи перемещаются только с него.

Замечание. (Детали реализации Consistent Hashing)

- Список узлов можно хранить в дереве поиска или в отсортированном массиве.
- Перемещаются только непрерывные отрезки ключей (по хешам). На каждом узле можно хранить отображение из хешей в списки ключей.

Замечание. (О Consistent Hashing)

- Возможно неравномерное распределение ключей, из-за случайного выбора точек для узлов.
- При удалении узла все его ключи перемещаются на единственный узел.

Алгоритм. (Consistent Hashing: vnodes)
Пусть каждому физическому узлу соответствует несколько виртуальных:

$$h(\texttt{node} \mid 0),\ h(\texttt{node} \mid 1),\ \ldots.$$

Чем больше виртуальных копий, тем равномернее распределение ключей по узлам и больше нагрузки на память и время.

Алгоритм. (Multi-Probe Consistent Hashing)
Пусть каждому узлу соответствует только одна точка на круге. Теперь будем много раз проецировать ключ на круг, аналогично тому, как мы делали в vnodes. Берем ближайшую точку, соответствующую узлу. Тратится меньше памяти, но больше времени. (Слабый проигрыш по времени, но сильный выигрыш по памяти).

31 Шардирование. Общий принцип, JumpHash.

Первую часть вопроса см. в предыдущем билете.

Алгоритм. (JumpHash)
Пусть в системе N узлов.

- Обозначим за
$$0 \leqslant ch(k,N) < N$$
номер узла, на который должен попасть ключ k.
- При добавлении узла каждый ключ с вероятностью $(N+1)^{-1}$ переходит на новый узел.
- Все это происходит при предположении, что узлы только добавляются. Это разумно в системе, где число данных в основном растет. При отказе узла его место (номер) получает новый работающий узел.

Лемма 31.1. (О равномерности распределения узлов)
Пусть $\xi_N = ch(k,N)$ – случайная величина, номер узла, на котором лежит ключ k. Тогда $P(\xi_N = i) = \frac{1}{N}$.

Доказательство.

- База индукции. Очевидно, что $P(\xi_1 = 0) = 1$.
- Переход. Если $i = N$:
$$P(\xi_{N+1} = N) = \sum_{i=0}^{N-1} P(\xi_N = i) \cdot \frac{1}{N+1} = \sum_{i=0}^{N-1} \frac{1}{N} \cdot \frac{1}{N+1} = \frac{1}{N+1}.$$
Если $i \neq N$:
$$P(\xi_{N+1} = i) = P(\xi_N = i) \cdot \left(1 - \frac{1}{N+1}\right) = \frac{1}{N} \cdot \frac{N}{N+1} = \frac{1}{N+1}.$$

■

Алгоритм. (Наивная реализация) Напишем простую реализацию алгоритма, которая для заданного узла k эмулирует его жизнь при количестве серверов от 1 до n:

```
fun jumpHash(key: Key, n: Int) -> Int {
    random.set_seed(hash(key))
    result = 0
    for (i in 1 until n)
        if (random.uniform(0, 1) < 1/(i + 1))
            result = i
    return result
}
```

Алгоритм. (Хорошая реализация)
Заметим, что “прыжки” происходят редко, то есть достаточно часто

$$ch(k, j+1) = ch(k, j).$$

Будем вычислять только точки прыжков, то есть точки, в которых

$$ch(k, j+1) = j.$$

Предположим, что мы знаем точку последнего прыжка b:

$$ch(k, b+1) = b.$$

Тогда поставим задачу найти ближайшую справа точку, в которой произойдет прыжок:

$$ch(k, j+1) \neq ch(k, b+1),\ j \to \min_{j>b}.$$

Эквивалентная этой задача ставится так: найти максимальное j, в котором еще не произошел прыжок:

$$ch(k, j+1) = ch(k, b+1),\ j \to \max_{j>b}.$$

Лемма 31.2. $P(ch(k,n) = ch(k,m)) = \frac{m}{n}$, если $n \geqslant m$.

Доказательство.

- Если $n = m$, то
$$P(ch(k,n) = ch(k,n)) = 1 = \frac{n}{n} = \frac{m}{n}.$$
- Если $n > m$. Тогда прыжков не должно быть на шагах от $m+1$ до n. Вероятность того, что на $m+k$-м шаге не произойдет прыжок, равна $1 - \frac{1}{m+k} = \frac{m+k-1}{m+k}$. Получаем вероятность:
$$P(ch(k,n) = ch(k,m)) = \frac{m}{m+1} \cdot \frac{m+1}{m+2} \cdot \ldots \cdot \frac{n-1}{n} = \frac{m}{n}.$$

■

Алгоритм. (Хороший алгоритм, продолжение)
Пусть в точке $i \geqslant b+1$ еще не произошел прыжок. Тогда понятно, что $j \geqslant i$:

$$P(j \geqslant i) = P(ch(k,i) = ch(k,b+1)) = \frac{b+1}{i}.$$

Воспользуемся этим равенством, чтобы сделать более эффективный алгоритм. Сгенерируем случайное число $r \sim U(0,1)$. Тогда

$$j \geqslant i \iff r \leqslant \frac{b+1}{i},$$

что эквивалентно

$$j \geqslant i \iff i \leqslant \frac{b+1}{r}.$$

Выберем самую точную нижнюю границу на j:

$$j = \max_{i \leqslant \frac{b+1}{r}} i = \left\lfloor \frac{b+1}{r} \right\rfloor.$$

Это и будет очередная точка прыжка. Напишем код, который симулирует жизнь ключа:

```
fun jumpHash(key: Key, n: Int) -> Int {
    random.set_seed(hash(key))
    b = -1 // Last jump point
    j = 0 // Next jump point
    while (j < n) {
        b = j
        r = random.uniform(0, 1)
        j = floor((b + 1) / r)
    }
    return b
}
```

Лемма 31.3. (О времени работы JumpHash)
Математическое ожидание времени работы JumpHash составляет $\mathcal{O}(\log N)$.

Доказательство. Мы совершаем прыжки только вперед, причем каждый узел посещается не более одного раза.

$$\mathbb{E}[T(N)] = \sum_{i=1}^{N-1} \mathbb{E}[\xi_i] = \sum_{i=1}^{N-1} i^{-1} = \mathcal{O}(\log N).$$

■

Замечание.

- JumpHash использует $\mathcal{O}(1)$ памяти.
- JumpHash очень хорошо распределяет нагрузку.

32 Транзакции в распределенных системах. ACID. 2 Phase Locking.

Определение. *Транзакция* это единица работы над множеством элементов, хранящихся в базе данных.

Определение. *ACID:*

- *Atomicity* (атомарность) — все изменения или ничего.
- *Consistency* (согласованность) — перевод системы в согласованное состояние в конце транзакции.
- *Isolation* (изолированность) — параллельные транзакции не должны влиять друг на друга, а выполняться как будто бы последовательно.
- *Durability* (надежность) — завершенные транзакции сохраняются даже в случае сбоев и перезапуска системы.

Алгоритм. *Подходы к сохранению Atomicity:*

- **Подход 1**. Храним "собственную версию" данных в рамках транзакции (shadow copy):
  - Не делаем изменения основной копии до завершения (commit) транзакции.
  - Откидываем свою копию её если транзакция откатывается.
  - Получается *Redo log* — журнал изменений которые надо применить только в случае завершения транзакции.
- **Подход 2**. Храним «журнал отката»:
  - Вносим изменения в основную копию.
  - *Undo log* — запоминаем журнал по которому можно отменить (undo) все произведённые в транзакции изменения.
  - Если надо транзакцию откатить, то применяем undo log чтобы отменить внесённые изменения.

Алгоритм. *Подходы к сохранению Durability:*

- Либо все изменения исходных данных записаны в энергонезависимую (non-volatile) память.
- Либо *redo log* записан в энергонезависимую память (более популярно, т.к. это последовательный журнал, который проще писать на диск).

Определение. Максимальный *уровень изоляции* (isolation) level называется *сериализуемостью* (serializability) — все транзакции можно переупорядочить в последовательную историю исполнения, так чтобы никакие две транзакции не выполнялись параллельно.

Определение. *2-Phase Locking:*

- Каждая транзакция состоит из 2-х последовательных фаз — фаза получения блокировок и фаза отпускания блокировок.
- Блокировки могут браться и отпускаться в любом порядке в соответствующих фазах, при условии что каждая операция над элементом данных происходит после получения соответствующей ему блокировки и до её отпускания.

Замечание. 2PL исполнение гарантирует сериализуемость транзакции.

Замечание. Блокировка может быть решена локально каждым узлом (распределённые алгоритмы блокировки не нужны!).

Пример.

- Участник *P* решил что транзакция завершилась успешно (commit) и сохранил все изменения перед опусканием блокировок, сделав их видимыми другим участникам.
- Участник *Q* решил что транзакция завершилась неуспешно (rollback) и отменил все изменения перед опусканием блокировок.
- *Нарушена атомарность* транзакции. Способы решения в следующем билете.

33 Транзакции в распределённых системах. ACID. 2 Phase Commit.

Про транзакции написано в предыдущем билете.

Определение. *2 Phase Commit:*

- Централизованный алгоритм завершения транзакции, т.е. у каждой транзакции есть выделенный *transaction coordinator*.
- **Фаза 1.** Запрос (request):
  - Координатор спрашивает каждого участника о готовности к завершения транзакции.
  - Участник может ответить *yes* только, если он может обеспечить завершение даже в случае сбоя (т.е. он всё записал) и все данные корректны, иначе *no*.
  - Транзакцию можно завершить только, если все участники ответили *yes*.
- **Фаза 2.** Завершение:
  - Координатор принимает решение commit/abort и записывает его.
  - Координатор доводит до участников решение.

Замечание. *Ошибки:*

- Transaction Commit = Consensus, поэтому к нему применим результат *FLP*.
- При отказе узлов или связи *2PC* не сможет завершиться, до восстановления узлов/связи.

Замечание. Много полезных картинок в конце презентации к лекции 7.

34 Raft. Алгоритм, его свойства.

Замечание. *Paxos* обладает недостатками, его трудно реализовать на практике:

- Очень сложен в понимании.
- Построен на “однократном консенсусе”
- Проблемы с практической реализацией:
  - Нужен multi-Paxos.
  - Нужен выбор лидера.
  - Нужен общий журнал.

Определение. *Дизайн Raft*:

- *Понятность* (минимум состояний и недетерминизма).
- Подзадачи:
  - *Leader election*.
  - *Log replication*.
  - *Safety* (согласованность в консенсусе).
- Гарантии:
  - *Election Safety* (не более одного лидера).
  - *Leader Append-Only* (лидер только добавляет).
  - *Log Matching* (все записи в журналах совпадают).
  - *Leader Completes* (committed записи будут у будущих лидеров).
  - *State Machine Safety* (однозначный выбор операции).

Алгоритм. *Выбор лидера:*

- Весь процесс работы Raft разбит на термы, в начале каждого терма происходит выбор лидера. Термы нумеруются последовательными числами. Каждый узел помнит максимальный номер. В каждом терме не более одного лидера.
- **Состояния** узлов:
  - *Leader* — обрабатывает все запросы.
  - *Follower* — все узлы, кроме лидера.
  - *Candidate* — узлы претендующие на лидерство (роль существует только на этапе выборов).

- **Переходы** состояний:
  - Все *followers* следят за *heartbeats* (регулярные сообщения) от лидера. Если лидер не сообщает о себе определённое время, то узел инициирует новые выборы. Для того, чтобы все узлы одновременно не ломились на выборы используют технику рандомизированных таймаутов.
  - *Candidate* занимается одой из следующих вещей:
    * Ожидает большинство голосов за себя.
    * Ожидает выбора другого лидера.
    * Ожидает таймаут.
  - *Leader* работает, пока жив его терм, т.е. до тех пор пока он не обнаружит терм с большим номером в системе.

Определение. *Журнал* представляет из себя последовательность пронумерованных ячеек (log index), которые хранят номер терма и операцию.

Алгоритм. *Репликация журнала:*

- *Committed* — записи, которые подтверждены большинством.
- *Leader* регулярно рассылает всем *AppendEntries*:
  - *leader id* и пачка записей (*log index, term, data*).
  - Информация (*log index, term*) предыдущей записи.
  - *Follower* добавляет запись в журнал только в случае, когда информация о предыдущей записи совпадает.

Замечание. Возможны следующие расхождения в журналах у *followers*:

- Отсутствие каких-то записей (не успели получить обновления).
- Неподтверждённые записи (например, какой-то умерший лидер, который не успел зафиксировать записи большинством).
- Оба вида расхождения вместе (корректный префикс записей и странный набор в хвосте).

Алгоритм. *Согласование журналов*:

- Храним для каждого *follower* next index, т.е. номер записи с которой нужно слать обновления.
- Если при получении новой записи информация о предыдущей совпадает, то *follower* применяет новую запись.
- Если при получении новой записи информация о предыдущей не совпадает, то *follower* сообщает *leader* об уменьшении своего next index, таким образом *leader* найдет общий префикс и после этого начнет добавлять записи.

Определение. *Safety механизм*:

- *Committed* записи в журнале не должны перезаписываться.
- *Election restriction* — не отдаём голос лидеру, если наш журнал более *свежий*. Для проверки сначала сравниваем *term* последней записи, в случае равенства смотрим на *log index*.

Замечание. Только записи из текущего *term leader* считаются *committed* при записи в большинство узлов. (Пример проблемы подробно рассказан на 8 лекции).

Теорема 34.1. *Raft* гарантирует *safety*.

Доказательство.

- Докажем от противного.
- Пусть *leader* $S1$ терма T закоммитил запись, но она не попала в журнал *leader* $S5$ терма U $(T < U)$.
- Значит существует процесс $S3$, в журнале которого была эта запись, но он отдал голос за лидерство $S5$ в терме U. Процесс $S3$ обязательно существует, так как *commited* запись лежит в журнале большинства процессов и лидер имеет голоса большинства процессов, значит их пересечение не пусто.
- Процесс $S3$ нарушил *election restriction*, получено противоречие.

■

35 CAP теорема (концепции, подходы, без доказательства).

Определение. Распределённой системе хранения необходимо три свойства **CAP**:

- *Consistency* — все клиенты видят одинаковые данные (atomic/strong consistency).
- *Availability* — система работает несмотря на сбои узлов (запрос к неотказавшему узлу должен получить ответ).
- *Partition tolerance* — система работает несмотря на обрыв связи между разными частями системы (partition).

Теорема 35.1. Можно иметь только два из трёх свойств **CAP**.

Определение. *Gossip protocol* — принцип построения системы, при котором узлы распространяют информацию друг другу по мере возможности. В том числе то, что они узнали от других узлов — слухи.

Замечание. *Gossip protocols* могут обеспечить только *eventual consistency*. Это означает, что при отсутствии сбоев через какое-то время все узлы системы будут знать согласованное состояние системы.

Примеры. *Варианты систем*:

- **CA**: тривиальные системы, например, с координатором.
- **CP**: Paxos, Raft и другие.
- **AP**: Gossip protocols.
- В некоторых случаях системы принято делить на два вида:
  - $ACID =$ **CA** — Atomicity, Consistency, Isolation, Durability.
  - $BASE =$ **AP** — Basically Available, Soft state, Eventual consistency.

Замечание. Основная проблема *Gossip* протоколов это разрешение конфликтов после сбоя (например, разрыв связи). Пути решения проблемы:

- Приложение само решает как разрешить конфликт.
- $CRDT$ (см. следующий билет).

36 Gossip. Eventual consistency. CRDT и дельта-CRDT, примеры со счетчиком, множеством.

Gossip и базовые определения читайте в предыдущем билете.

Замечание. *CRDT* позволяет *Gossip* системе однозначно восстановить результат конфликтующих операций.

Определение. *CRDT* оперирует состоянием системы. *Операция* в системе задается состоянием x, в которое система переходит при применении этой операции к начальному состоянию s_0.

Определение. Операция *объединения* (merge) для состояний после нескольких операций:

- x — состояние после применения операции x_{op}.
- y — состояние после применения операции y_{op}.
- $x \sqcup y$ — состояние после объединения состояний x и y.

Определение. Множество состояний системы образует *полурешетку* — полугруппа с коммутативной и идемпотентной операцией *объединения* состояний:

- *Коммутативность*: $x \sqcup y = y \sqcup x$
  - Порядок операций не важен.
  - Не нужен *total order*, т.е. консенсус по поводу того в каком порядке операции происходили на разных узлах.
- *Идемпотентность*: $x \sqcup x = x$
  - Повторное применение операции не меняет состояние.
  - Поэтому не нужна *reliable* доставка, то есть *exactly once delivery*.
- *Полугруппа (ассоциативность)*: $x \sqcup y \sqcup z = x \sqcup (y \sqcup z)$
  - Можно объединять операции в любом порядке.

Пример. *CRDT* — Увеличивающийся счётчик. Вариант с *операциями*:

- *Операция*: Добавить x к значению счётчика.
- *Состояние*: Множество операций.
- В данном случае $\sqcup = \cup$, следовательно объединение *коммутативно*.
- По множеству операций возможно восстановить значение.
- Нужно чтобы у всех участников было общее мнение о множестве проведённых операций. В случае сбоя множества могут быть разными. Используем *gossip* для восстановления.

- *Идемпотентность* получим, добавив уникальной идентификатор каждой операции.

Замечание. Предыдущий пример не масштабируется по числу операций.

Пример. *CRDT* — Увеличивающийся счётчик. Вариант с *состоянием*:

- *Операция*: Добавить x к значению счётчика.
- *Физическое состояние*: Вектор размером в число узлов. (каждый узел увеличивает свою компоненту).
- *Логическое состояние*: сумма элементов вектора
- *Объединение*: Покомпонентный max. Это *коммутативная* и *идемпотентная* операция. Но важно, что счётчик только растет.
- Рассылаем через *gossip* текущее известное состояние. Размер состояния фиксирован, но надо пересылать $\mathcal{O}(n)$ значений.

Определение. *δ-CRDT* — это *CRDT* на основе состояний, где пересылается не все целиком, а только отличие состояния от предыдущего.

Пример.

- Счётчик: если узел i увеличивает счётчик, то пересылаем не весь вектор, а только отображение $\{i \to x_i\}$.
- Операция *merge* это объединение отображений и покомпонентный максимум.
- Каждому соседу надо посылать только изменения по сравнению с предыдущим посланным сообщением, т.е. только значения изменившихся значений в отображении.
- *Итого:* в стабильной системе одна операция вызывает распространение сообщения размером $\mathcal{O}(1)$.

Примеры.

- *Счётчик вверх и вниз*: два счётчика для увеличения и уменьшения.
- *Растущее множество*: аналогично счётчику, операцией слияния будет объединение множеств (можно передавать дельты изменений, вместо самих множеств).
- *Множество с операций удаления*: разделим на множество добавленных и удалённых элементов. Но множество удаленных элементов растет вечно. На практике удалённые элементы выкидывают через определённый промежуток времени.

Замечание. *CRDT* можно композировать между собой для представления более сложных объектов.

37 Leader/Follower репликация. Общий принцип, реализация, синхронная и асинхронная репликация.

Определение. *Leader/Follower* репликация. На каждом узле хранятся одни и те же данные.

- Один узел является лидером, принимает запросы на чтение и запись.
- Остальные узлы являются репликами, принимают запросы только на чтение.

Преимущества:

- Можно читать из любой копии ⇒ увеличиваем пропускную способность на чтение (но не на запись).
- Можно читать из ближайшей копии ⇒ уменьшаем задержку чтения (но не записи).
- Храним данные в нескольких копиях ⇒ увеличиваем надежность, так как если одна реплика недоступна, можем читать данные из любой другой.

Замечание. Так как лидер жестко выбран заранее, нет необходимости постоянно приходить к консенсусу по поводу его выбора.

Определение. *CDN (Content Delivery Network)* — системы, основанные на Leader/Follower репликации. Используются в случаях, когда нужно часто читать данные, при этом запись происходит редко.

Алгоритм. (Реализация Leader/Follower) Клиент посылает запрос на изменение. Лидер должен разослать эти измения репликам. Существует несколько вариантов репликации:

- **Репликация на уровне команд**.

  Лидер в текстовом виде рассылает исполненные команды.

  Преимущества:

  – Пересылаем небольшой объем данных, так как сколько бы данных не затрагивала команда, мы пересылаем только ее текстовый вид.

  Проблемы:

  – Недетерминированные команды (рандом, текущее время). Решение: можно пересылать не функцию, а результат ее выполнения.

  – Не гарантируется одинаковый порядок исполнения команд. Решение: использование журналов.

- **Репликация на уровне журналов**.

  Каждая реплика ведет журнал изменений. Лидер рассылает записи из журнала. Запись в жернале имеет вид (O, F, X, Y) — объект, поле, старое и новове значения.

  Преимущества:

  – Определенный порядок.
  – Полный детерминизм.
  – Не пересылаем записи откаченных транзакций.
  – Записи из журналов имеют более простой формат, а значит, применять их проще, чем команды.

  Недостатки:

  – Большой объем пересылаемых данных. Но в реальной жизни преимущества перевешивают этот недостаток.

Определение. *Синхронная репликация:*

- Лидер применяет операцию локально и затем рассылает ее на все реплики.
- Реплики применяют операцию локально и отвечают лидеру.
- Лидер дожидается подтверждения от всех реплик, что они применили операцию.
- После этого лидер подтверждает применение операции у себя и сообщает клиенту о завершении операции.
- **Актуальность данных**
  – Если лидер сообщил клиенту о завершении операции, то эта операция применена на всех репликах.
  – Таким образом, каждая реплика содержит актуальные данные, и ее состояние совпадает с состоянием лидера.
  – Чтение из любой реплики даст один и тот же результат.
  – Консистентность ⇒ жертвуем доступностью.
- **Недоступные узлы**
  – При недоступности любого узла система не может обрабатывать запросы на запись.
  – Но при этом может обрабатывать запросы на чтение и получать актуальные данные на любом доступном узле.

Определение. *Асинхронная репликация:*

- Лидер применяет операцию локально и сразу сообщает клиенту о завершении операции.
- После этого лидер рассылает операцию репликам.
- Не ждем пока реплики получат изменения и подтвердят их.
- Можно пересылать не каждый запрос, а группировать их.
- **Недоступные узлы**
  - Если недоступна реплика, то система всё равно может обрабатывать любые запросы.
  - Если лидер упал, мы не можем произвести замену без потерь, так как потенциально на каждой реплике нет какого-то суффикса журнала.
  - Можем ждать, пока лидер поднимется. До этого момента система не сможет обслуживать запросы на запись.
  - А можем произвести замену с минимальной потерей данных.

38 Leader/Follower репликация. Failover (для синхронного и асинхронного случаев), масштабирование репликации, снимки данных.

Определения см. в предыдущем билете.

Определение. *Failover* — процесс замены упавшего лидера на новый.

Алгоритм. (Замена лидера, синхронная репликация)

- Можем заменить лидера на любую реплику, так как каждая реплика содержит актуальные данные.
- В системе ни в какой момент времени не должно быть двух лидеров.
- Поэтому сначала нужно выключить старого лидера (например, изолировать на уровне сети).
- Только после отключения старого лидера, включаем новый.
- **Незавершенная операция**:
  - Если лидер успел отреплицировать последнюю операцию хотя бы на один узел, то какой-то клиент мог бы успеть прочитать результат выполнения операции с этого узла. Такой узел станет новым лидером и отреплицирует операцию на остальные узлы.
  - Если лидер не успел отреплицировать последнюю операцию ни на один узел, то ни один клиент не мог успеть прочитать результат выполнения операции. Также клиент, который послал запрос на применение операции, не успел получить подтверждение о завершении операции. Таким образом, эту операцию можно просто забыть.

Алгоритм. (Замена лидера, асинхронная репликация)

- Невозможно произвести замену без потерь, так как потенциально на каждой реплике нет какого-то суффикса журнала.
- Можно ждать восстановления лидера (и не обслуживать запросы на запись) или произвести замену с минимальной потерей данных.
- Выбираем реплику с самым длинным журналом и делаем её новым лидером. Так мы потеряем меньше всего данных.
- В системе всё ещё не может быть одновременно двух лидеров, поэтому сначала выключаем старый, а затем назначаем нового.
- Нарушена *Durability*: лидер сказал клиенту, что его транзакция завершилась успешно, но данные его транзакции были потеряны.

- Когда прежний лидер восстановится, произойдет объединение логов для обеспечения *Eventual consistency*.

Алгоритм. (Объединение логов)

- После восстановления старого лидера в его журнале есть суффикс, которого нет у нового лидера.
- У нового лидера также есть суффикс, которого нет у старого.
- **Решение конфликтов**

  Часть суффикса старого лога может быть без проблем применена на новом лидере. Но некоторые записи будут конфликтовать с суффиксом нового лога. Возможные решения:

  - Забыть конфликтующую запись из старого лога.
  - Использовать *CRDT*. Но записи из суффикса лога нового лидера иногда не имеют смысла без записей из суффикса старого лидера, поэтому такие системы сложно проектировать.

Замечание. Чем больше реплик, тем больше времени надо потратить лидеру, чтобы переслать на каждую весь журнал. Это ограничивает масштабируемость.

Способы решения проблемы масштабируемости репликации:

- *Каскады репликации*.
  - Реплики составляют древовидную структуру, каждая реплика имеет свой уровень.
  - Реплика, получив изменение, пересылает его репликам на следующем уровне.
  - Таким образом, лидер тратит относительно немного времени на рассылку изменения репликам.
- *Gossip*.
  - Лидер пересылает журнал подмножеству реплик.
  - Дальше реплики сами рассылают друг другу недостающие фрагменты.
  - На каждой реплике применяем только записи из последовательного префикса, до первой “дырки“.

Пример. (Асинхронная репликация на практике)

- Снимки данных
  - Подключаем к лидеру новую реплику в асинхронном режиме.
  - Передаём на новую реплику префикс лога.
  - Как только получили необходимые данные, отключаем реплику.

  - Таким образом, получили снимок данных, содержащий префикс всех операций.

- Применение снимков для инициализации новых реплик.
  - При подключении новой реплики не хотим передавать на неё весь журнал, с первой операции.
  - Инициализируем новую реплику снимком данных, отражающим операции $[1 - N]$.
  - Начнём передавать на неё журнал с $N + 1$-ой записи.

- Разные реплики могут работать в разных режимах. Например, некоторые реплики можно сделать синхронными, а некоторые асинхронными.

- Имеем хотя бы одну синхронную реплику на случай падения лидера.

39 Слабые модели консистентности. Мотивация, чтение собственных записей.

Замечание. Репликация с точки зрения *CAP*-теоремы:

- Синхронная репликация — это *C*-протокол.
- Асинхронная репликация (без *Failover*) не является ни *C*-, ни *A*-протоколом.
  - Доступности нет, потому что в случае падения лидера не можем обслуживать запросы на запись.
  - Но зато есть частичная доступность, так как продолжаем принимать запросы на чтение.
  - Консистентности нет, потому что возможны нелиаризуемые исполнения.
  - Зато есть *Eventual Consistency*, так как в отсутствие записей каждая реплика в конечном итоге получит весь журнал.

Пример. Нелинеаризуемое исполнение Лидер ответил ОК на запись, клиент прочитал не то значение с реплики, потому что репликация записи ещё не произошла.

Определение. *Слабые модели консистентности:*

- Не запрещают все нелинеаризуемые исполнения, так как это приведет к потере доступности.
- Но запрещают некоторые исполенения, которые могут отрицательно повлиять на пользовательский опыт.
- Какие именно исполнения запретить и как их избегать зависит от самой модели.

Пример. (Чтение собственных записей)
Хотим, чтобы если клиент сделал какое-то изменение, то при дальнейших запросах на чтение он видел результат этого изменения. Есть два подхода к реализации такой слабой модели консистентности:

- *Чтение из лидера:*
  - Представим, что в нашей системе пользователи могут менять только свои данные (таких данных мало) и читать чужие данные (их много).
  - Тогда будем читать свои данные с лидера, а чужие с реплик.
  - Таким образом, мы всегда будем читать актуальные собственные записи.
  - Примеры таких систем: сайты с объявлениями, профили в социальных сетях.
- *Запоминание номера записи:*

– После того, как клиент отправляет запрос на запись, он получает подтверждение и номер этой записи в журнале лидера, которой соответствует исполененная транзакция.
– Клиент запоминает полученный номер у себя.
– Вместе с запросом на чтение клиент запрашивает у реплики номер последней записи в ее журнале.
– Реплика в ответ посылает номер своей последней записи в журнале и ответ на запрос.
– Если номер, полученный от реплики, меньше сохраненного у клиента, то это значит, что запись не успела отреплицироваться. Поэтому делаем запрос на чтение следующей реплике.
– Если номер, полученный от реплики, не меньше сохраненного у клиента, то возьмем ее ответ на запрос и больше не будем спрашивать другие реплик.

40 Слабые модели консистентности. Мотивация, монотонное чтение.

Первую часть вопроса см. в предыдущем билете.

Пример. (Монотонное чтение)
Запись может пропадать при повторном чтении (например, если запросы попадают к разным репликам). Поэтому хотим, чтобы если клиент уже один раз получил результат какого-то изменения, то при дальнейших чтениях он будет продолжать получать этот же результат, а не данные до изменения. Есть два подхода к реализации такой слабой модели консистентности:

- *Запоминание номера последней записи:*
  - Аналогично чтению собственных записей.
  - При **каждом** запросе узнаём N — номер последней записи в журнале.
  - Читаем только из реплики, на которую N-ая запись уже отреплицировалась.
  - Помимо монотонного чтения, такая модель обеспечивает чтение собственных записей.
- *Чтение из одной реплики:*
  - Будем производить все чтения из одной и той же реплики.
  - Выбираем ближайшую к клиенту группу реплик — дата-центр.
  - Внутри дата-центра определяем номер реплики по хешу.
  - Проблема: если мы меняем количество реплик в дата-центре, то часть клиентов должна будет начать читать с новой реплики, и они могут столкнуться с немонотонным чтением. Решение: Для уменьшения числа таких клиентов используем консистентное хеширование.
  - Проблема: после переезда клиент будет делать запросы в другой датацентр и опять возможно немонотонное чтение. В реальной жизни: скорость передвижения клиента значительно меньше скорости передачи данных, поэтому данные успеют отреплицироваться на реплики нового датацентра.
  - Проблема: реплика, с которой читал клиент упала, и клиент начинает читать из другой реплики, опять может возникнуть немонотонное чтение. Решение: аналогично изменению числа реплик, обеспечиваем монотонное чтение только в отсутствие сбоев.

41 Слабые модели консистентности. Мотивация, чтение согласованного префикса.

Первую часть вопроса см. в билете 39.

Пример. (Чтение согласованного префикса)
Если в системе есть шардирование, то мы хотим запретить ситуацию, когда клиент читает измененные данные, а вот данные, от которых они зависят, будут прочитаны не измененными. Например, клиент видит зависимую сущность, но не видит сущность, от которой она зависит. При отсутствии шардирования получаем гарантию автоматически. Есть несколько подходов к реализации такой слабой модели консистентности:

- *Настройка схемы шардирования:*
  - Если сущность Y может зависеть от сущности X — будем хранить их в одной секции.
  - Например, можно хранить все сообщения из одного чата в одной и той же секции (секционирование по $hash(chat\_id)$).
  - Но сделать так можно не всегда.
- *Специфичные для приложения методы:*
  - Например, просто не будем отображать комментарий, если не можем отобразить исходное сообщение.
  - Такое могло произойти, если сообщение еще не успело отреплицироваться на реплику, с которой читаем.
  - Как только исходное сообщение будет получено, отобразим и само сообщение, и комментарии.
- *Версионирование записей*

Алгоритм. (Версионирование записей)

- Каждая запись в логе хранит:
  - Порядковый номер записи в логе текущего шарда.
  - Порядковые номера записей в логах других шардов, от которых зависит наша запись.
- Хотим читать с остальных шардов только те записи, которые были не раньше записи, от которой мы зависим.

Возможная реализация алгоритма:

```
versions[1..N] <- null
while True:
    for i <- 1..N if versions[i] == null:
        versions[i] = fetchShard(i)
    invalidations = False
    for i <- 1..N, j <- 1..N, if i /= j and
        versions[i] /= null and versions[j] /= null:

        if versions[i].dependencies[j] > versions[j].v:
            invalidations = True
            versions[j] = null
    if not validations:
        return versions
```

Проблемы такого подхода:

- Завершаемость алгоритма не гарантируется.
- Одна из более поздних записей лога может соответствовать удалению сущности, от которой мы зависим. Решения:
  - Можно никогда не удалять данные, только помечать как удалённые.
  - Можно использовать персистентные структуры данных и запрашивать нужную версию (никто так не делает).

42 Multi-Leader репликация. Мотивация, схема с множеством дата-центров, конфликты при записи. Синхронное разрешение конфликтов, избегание конфликтов.

Определение. *Multi-Leader репликация:*

- Можем писать в любого из лидеров ⇒ уменьшаем задержку записи.
- Несколько лидеров принимают запросы на запись ⇒ увеличиваем пропускную способность на запись.
- Можем пережить падение лидера и даже после сбоя сможем обслуживать запросы как на чтение, так и на запись ⇒ увеличиваем надежность.
- **Общая схема:**
  - Несколько дата-центров, в каждом — свой лидер.
  - У каждого лидера несколько реплик, подчинённых ему по схеме Leader/Follower.
  - Лидер принимает запросы на чтение и запись, реплики — только на чтение.
- **Конфликты при записи:**
  - Параллельные конфликтующие записи являются проблемой, даже если нет аппаратных сбоев.
  - В случае аппаратных сбоев ситуация становится ещё сложнее, так как реплики не могут синхронизироваться и обнаружить конфликт.

Алгоритм. (Синхронное разрешение конфликтов)

- Не говорим пользователю, что его операция завершена, пока она не отреплицируется на всех лидеров или хотя бы на кворум.
- Решение идейно эквивалентно Paxos/Raft/etc.
- Низкая доступность: изолированный лидер не сможет принимать запросы на запись, но по-другому быть не может вследствие CAP-теоремы.
- Высокая задержка записи: пользователь должен ждать, пока запись отреплицируется на кворум лидеров. Решение:
  - Возвращать пользователю ответ сразу, как только хотя бы один лидер обслужил запись.
  - Тогда задержка, обусловленная репликацией на других лидеров, будет скрыта от пользователя.

Алгоритм. (Избегание конфликтов)

- Если параллельные изменения объекта X могут привести к конфликту, будем направлять все запросы на изменение объекта X через одного лидера.
- На практике часто помогает уменьшить количество конфликтов.
- Есть много случаев, когда такая стратегия не применима.
- Проблемы:
  - Пользователь может переехать, и запросы на изменение его объявлений должны пойти через нового ближайшего лидера.
  - Не всегда все запросы на изменение объекта X можно направлять через одного лидера.

Алгоритм. (Использование CRDT)

- Конфликтов не бывает в принципе: применение операций в любом порядке приводит к одному и тому же результату.
- С помощью Gossip когда-нибудь на каждый узел отреплицируется весь набор операций.
- Значит, итоговое значение на каждом узле будет одинаковым.

43 Multi-Leader репликация: векторные версии, объединение версий. Разрешение конфликтов при чтении и записи, параллельное разрешение конфликтов, изменение состава кластера.

Алгоритм. (Векторные версии)

- Храним на каждой реплике key-value базу данных.
- На каждой реплике для каждой записи ключ-значение храним вектор версий.
- Реплики обмениваются локальными значениями вместе с версиями.
- k-ая компонента вектора показывает, сколько обновлений этого ключа мы видели с реплики k.
- При изменении значения увеличиваем локальную компоненту вектора на 1.
- Если A *happens-before* B, то A может быть удалена.

Алгоритм. (Параллельные обновления)

- Параллельность определяется так же, как и в векторных часах.
- Мы сами не можем определить, какое значение правильное (так как нет *happens-before*), тогда сохраним оба значения и пусть приложение само решит, что делать.
- Новая версия – покомпонентный максимум из двух несравнимых векторов.

Замечание. Приложение может само решать конфликты. Пример: объединение множеств, так элементы не будут потеряны, но могут остаться удаленные.

Замечание. Некоторые конфликты невозможно разрешить на уровне приложения. Например, запись числа в поле объекта.

Алгоритм. (Разрешение конфликтов при чтении)

- Приложение читает данные и понимает, что версии разошлись.
- Решает конфликт, получает итоговое значение.
- Записывает его в базу.
- Увеличивая локальную версию той реплики, куда новая версия будет записана.
- Результирующему значению будут предшествовать все конфликтующие записи.

- Результирующее значение заменит любое из них.

Алгоритм. (Разрешение конфликтов при чтении на разных узлах)

- Конфликт может быть одновременно решен на нескольких узнал, но тогда получаем несравнимые вектора.
- Вероятность такого мала, так как конфликт решается при чтении клиентом.
- Используем детерменированное разрешение конфликтов.
- Как обычно, либо можем сохранить два значение и дать приложению решить конфликт.
- Вероятность такого конфликта мала, так как конфликт решается приложением при чтении, то нужно чтобы с двух разных узлов два приложения параллельно прочитали конфликтные значения, решили этот конфликт и сделали параллельную запись на два разных узла.

Алгоритм. (Разрешение конфликтов при записи)

- Есть конфликты, которые сама реплика может решить при получении конфликтующей версии.
- Обычно так решаются очень простые конфликты, которые не требуют вмешательства сложной логики на стороне приложения.
- Например, параллельная запись в разные поля объекта.

Алгоритм. (Изменение состава кластера)

- Вместо вектора будем хранить отображение.
- Ключ – имя узла. Значение – версия, которая была увидена с узла.
- Если же в отображении нет нужного нам ключа, то представим, что он там есть, и значение 0.

Алгоритм. (Удаление старых версий)

- Мотивация: векторные часы при увеличении количества реплик увеличиваются в размерах.
- Хотим удалять старые значения, которые соответствуют выведенным из эксплуатации узлам.
- Будем на каждом из узлов помнить, когда мы получали в последний раз сообщение от каждой из реплик.
- Из векторных часов можно удалить ключ, который соответствует реплике, от которой дольше всего не было сообщений, так как с наибольшей вероятностью эта реплика уже не существует.
- Есть возможность потерять отношение *happens-before*.

44 Multi-Leader репликация. Использование деревьев Мёркла для синхронизации реплик, Sloppy Quorum, Hinted Handoff.

Алгоритм. (Деревья Мёркла)

- Построение дерева Меркла: в листьях значение $h(key_i \mid value_i)$, в остальных вершинах хеш от конкатенации значений в вершинах-детях. Значение в корне называется *digest*.
- Реплики сравнивают *digest*, чтобы узнать о наличии расхождений.
- Рекурсивно спускаясь по уровням дерева, реплики найдут различия в листовых вершинах. Причем им не нужно пересылать поддеревья, в которых нет отличий.

Алгоритм. (Sloppy Quorum)

- Если читаем с единственной реплики, велик шанс прочесть устаревшее значение (например, она может быть медленная).
- Будем читать с R реплик и брать самое свежее среди них значение, так уменьшается вероятность прочитать устаревшее значение. Если прочитали несколько параллельных версий, то решаем конфликт.
- Если пишем на одну реплику, то она может быть медленнее, рискуем потерять *Durability* при ее падении.
- При записи используем W реплик, так запись быстрее отреплицируется. Клиенту сообщаем, что запись удалась только после записи на W реплик. Также можем делать эти W записей транзакционно.
- Пусть N – количество узлов в кластере, тогда:
  $R + W > N$ – каждая подтверждённая запись будет прочитана
  $R + W <= N$ – улучшается *Durability* записи, уменьшается вероятность прочитать старое значение, но можем не прочесть сделанную запись.

Алгоритм. (Hinted Handoff) Иногда, из-за недоступности, мы не можем сделать запись на W узлов, тогда сделаем запись на узлы, которые не должны хранить наши данные (например, узлы, являющиеся репликами другой партиции). Эти узлы знают, что они хранят записи не принадлежащие им, и они отреплицируют эти данные на нужные узлы после их поднятия.

45 Multi-Leader репликация. Last Write Wins.

Один из способов решения конфликта — оставлять на каждой реплике только последнюю (в некотором смысле) запись. Нельзя на каждой реплике сохранять последнюю пришедшую запись, потому что тогда, они даже *eventually* не придут к согласованности.

- Физические часы — каждая запись будет иметь метку времени — физическое время той реплики, которая приняла эту запись, но из-за плохо синхронизированных часов может отбросить последнюю запись, хотя последнюю надо оставлять.
- Логические часы — векторные часы, основанные на отношении “произошло до“, но среди параллельных записей нет той, которая произошла до, и брать любую из них тоже нельзя.

Определение. Метод *Last Write Wins* не работает при выборе любой из параллельных записей.
c happens-before b, $a \parallel b$, $a \parallel c$.
Приоритет по убыванию: c, a, b.

- При конфликте берем покомпонентый максимум векторных версий:
  $a + c = c, a + c \parallel b$
  $(a + c) + b = a + c = c$
  $a + b = a, c \to (a + b)$
  $(a + b) + c = a + b = a$

- При конфликте берем версию победителя:
  $a + c = c, (a + c) \to b$
  $(a + c) + b = b$
  $a + b = a, (a + b) \parallel c$
  $(a + b) + c = c$
- Поэтому держать только актуальное значение на реплике, и заменять при необходимости на нужное не получится.

Алгоритм. (Решение конфликтов)

- Для каждого ключа на каждой реплики храним N значений — последнее полученное с каждой из N реплик значение по соответствующему ключу.
- Когда хотим получить актуальное значение, строим множество W, в котором оставим только параллельные операции, а те которое связаны хоть с кем-то отношением “произошло до“ выкинем, среди этого множеставва оставим одно значение произвольным образом, например по timestamp’у.

Замечание. Серьезная проблема — понимать, когда и какую из старых операций можно забыть.

Пример. Даже если в операции не упоминается явно ключ — не значит, что можно забыть.

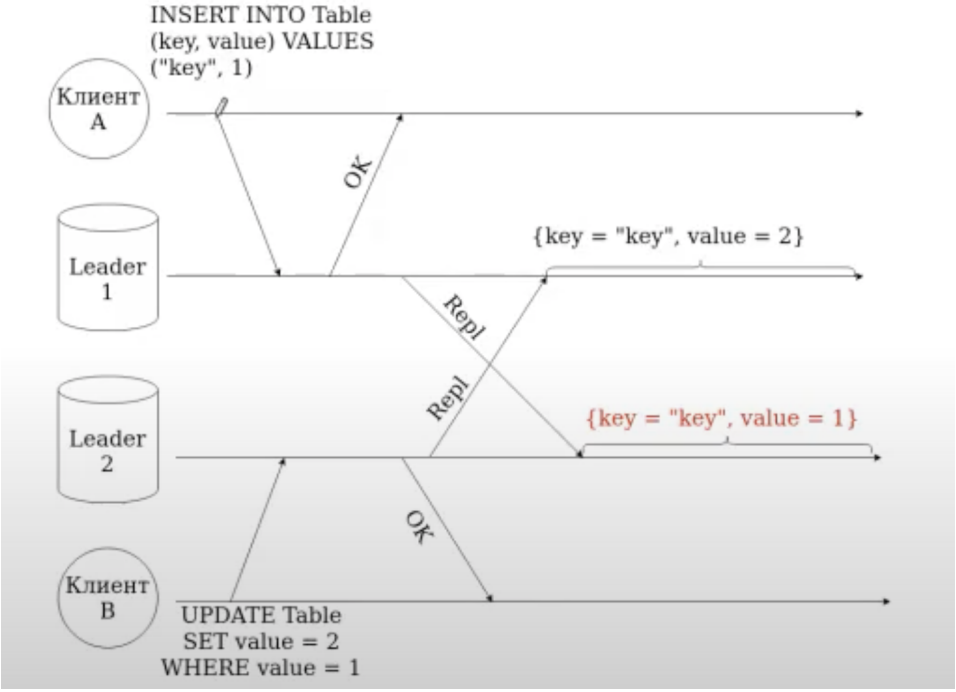


Рис. 1: Пример для РСУБД

Пример. Инкрементор:

- Состояние — единственное число.
- Система выполняет запросы вида “увеличить число на k“.
- Если не храним предка пришедшей записи, то мы не можем сказать, на сколько увеличил третий узел.
- Храним старое состояние, пока имеется возможность получить его непосредственного потомка.
- Выкидываем старое, когда по каждой компоненте вектора V получили “следующее“ состояние.

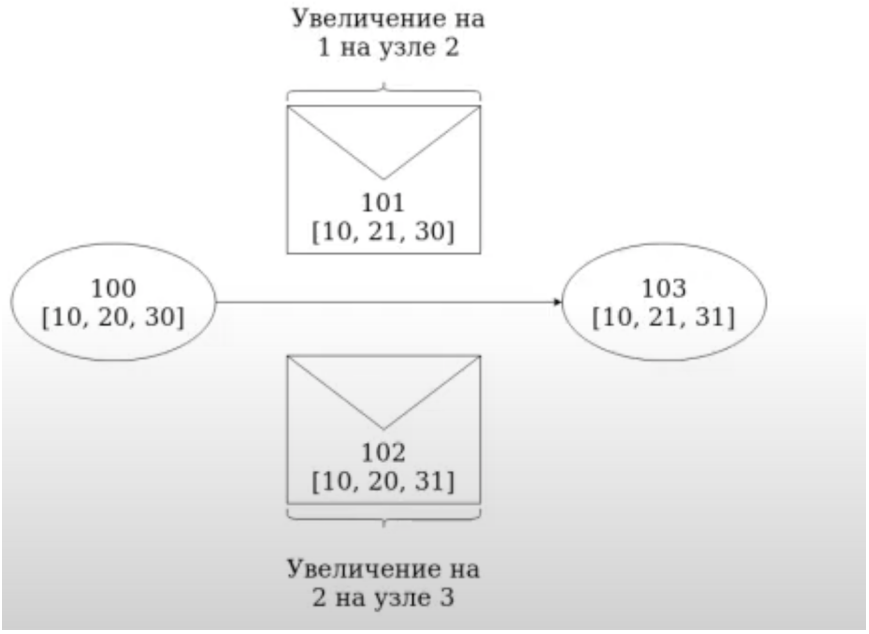


Рис. 2: Параллельная запись

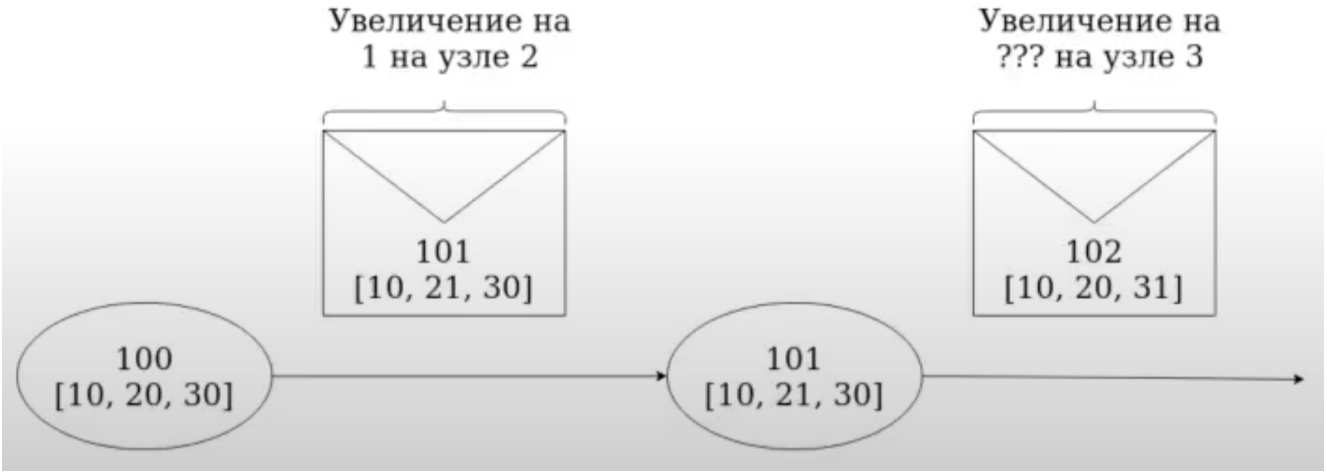


Рис. 3: Последовательная запись

46 Кеширование. Cache-Aside и Cache-Through архитектуры, слабые модели консистентности.

Главная мотивация – уменьшение задержки ответов и сглаживание нагрузки. Нагрузка на базу данных будет распределена более равномерно, потому что популярные объекты будут часто запрашиваться из кеша. Задержка при чтении из кеша маленькая, поскольку:

- Кеш обычно держит все данные в RAM.
- Кеш не поддерживает персистентность.
- Кеш не поддерживает сложные индексы (почти все кеш - просто хеш-таблица).

Определение. *Cache-Aside* – кеш и СУБД не связаны напрямую, клиент делает запросы в оба хранилища. Пример: MySQL + Memcached.

Алгоритм. (Работа с Cache-Aside)

- Синхронные запросы: сначала запрос в кеш, если там нет, то запрос в базу данных, сохранить в кеш. Время на запрос при отсутствии значения в кеше: $2 * RTT(cache) + RTT(DB)$.
- Асинхронные запросы: запрос отправляется в кеш и СУБД. Время на запрос при отсутствии значения в кеше: $RTT(cache) + RTT(DB)$. Если значение есть в кеше, то СУБД работает впустую.
- Если у клиента произошел сбой между записью в СУБД и кеш, то в кеше останется старое значение. Решение - добавим TTL, чтобы вытеснять устаревшие значения.

Определение. *Cache-Through* – клиент обращается к кеш + СУБД как к единому целому.

Алгоритм. (Работа с Cache-Through)

- Запрос сначала попадает в кеш, если ответ есть, то он возвращается клиенту.
- Иначе запрос попадает в основное хранилище.
- Ответ попадает в кеш, сохраняется в нём, возвращается клиенту.
- Время на запрос при отсутствии значения в кеше: $RTT(cache) + RTT(DB)$.

Алгоритм. (Кеширование и слабые модели консистентности)

- Можно думать о кешах как о асинхронных репликах, в которых есть все те же проблемы и такие же способы их решения: чтение собственных изменений - читаем данные которые мы можем изменить из СУБД, без использования кеша, монотонное чтение - читаем из одного и того же кеша.

- Можно ли записывать в кеш - это снизит задержку на запись (в СУБД будем писать только когда кеш переполнится), но жертвуем Durability (кеш подтвержил клиенту запись, но упал и значения из RAM'а потерялись).
- Если кешей, поддерживающих запись, несколько - будем думать о таких системах как о Multi-Leader Replication.

Алгоритм. (Определение несуществования объекта в СУБД с помощью кеша)

- Будем кешировать подотрезок, отсортированного отрезка, который хранится в СУБД, по этому подотрезку можно понять отсутствие некоторых элементов в СУБД.
- Если запрашиваемый элемент лежит в границах подотрезка, то достаточно проверить наличие.
- Иначе запрос напрявляется в основное хранилище.

47 MapReduce. Последовательная реализация, примеры решаемых задач.

Определение. *Модель MapReduce*. Универсальная модель для распределенных вычислений. Решает задачи, которые можно представить через следующие операции:

- `map :: Document -> [(Key, Value)]`, обработка каждого документа независимо.
- `reduce :: [Value] -> Result`, свертка сгруппированных значений.

Алгоритм. (Последовательная реализация)

```
fun mapReduce(docs: [Document],
              mapper: Document -> [(Key, Value)],
              reducer: [Value] -> Result) -> {Key: Result}:
    kvs = {}
    for doc in docs:
        for key, value in mapper(doc):
            if key not in kvs:
                kvs[key] = []
            kvs[key].append(value)
    return kvs.map { key, values -> key, reducer(values) }
```

Пример. (Подсчёт встречаемости слов)

- Каждый документ разбивается на слова.
- `map` для каждого слова возвращает пару из этого слова и 1.
- `reduce` суммирует значения.

Пример. (Построение обратного индекса документов)

- Каждый документ разбивается на слова.
- `map` для каждого слова возвращает пару из этого слова и идентификатора документа.
- `reduce` создает по списку идентификаторов множество уникальных.

48 MapReduce. Распределенная реализация. Мапперы и редьюсеры, локальность map, сбои узлов, избыточность.

Алгоритм. (Распределенная реализация)

- *M узлов-мапперов* параллельно выполняют операцию `map` для разных документов.
- *R узлов-редьюсеров* параллельно выполняют операцию `reduce` для разных ключей. При этом все значения, соответствующие одному ключу, должны обрабатываться на одном узле (например, `hash(key) % R`).
- Один узел является *мастером* и координирует работу всех узлов.

Определение. *Маппер в распределенной реализации*.

- Каждый узел хранит в памяти R корзин, которые содержат пары ключ-значение.
- Корзины впоследствии попадают на свой определенный редьюсер.
- При переполнении памяти, корзины сбрасываются с оперативной памяти на диск, каждая корзина в отдельный файл.
- *Маппер* с определенной периодичностью уведомляет мастера о статусе выполнения задачи.

Определение. *Редьюсер в распределенной реализации*.

- Каждый узел читает предназначенные ему корзины с мапперов до тех пор, пока не будут собраны все значения.
- Далее сортирует и группирует собранные данные по ключу.
- Вызывает `reduce` на собранных значениях и записывает результат в итоговый файл.

Алгоритм. (Обработка сбоев мапперов)

- Сбой маппера определяется утратой сообщений о статусе выполнения задачи.
- Если узел передал все корзины соответствующим редьюсерам, то его задачу можно не перезапускать.
- Если какой-либо редьюсер не получил с упавшего маппера хотя бы одну корзину, подзадача упавшего маппера перезапускается на новом узле, а полученные раннее корзины от старого узла инвалидируются (т.к. алгоритм маппера может быть недетерминированным).

- **Замечание от автора.** Если после сбоя маппера, отправившего все корзины, также дал сбой редьюсер, задачу также придется перезапустить на другом узле, т.к. новому узлу-редьюсеру заново потребуются утраченные корзины. С другой стороны, корзины можно реплицировать.

Алгоритм. (Обработка сбоев редьюсеров)

- Сбой редьюсера определяется пингом.
- При сбое редьюсера до завершения свертки и репликации результирующего файла, его подзадача полностью перезапускается на другом узле. При этом потребуется заново собрать данные с соответствующих корзин на каждом маппере.
- В противном случае, если редьюсер успел завершить задачу и отреплицировал файл с результатом, то перезапуск его подзадачи не требуется.

Алгоритм. (Обработка сбоев мастера)

- В общем случае необходимо перезапустить всю задачу с нуля.
- Для ускорения восстановления мастер может с определенной периодичностью делать снимки состояния задачи: статус подзадач map и reduce, местоположение файлов с результатами и др.
- Используя информацию со снимков, можно перезапустить задачу, не повторяя уже выполненных вычислений.

Алгоритм. (Оптимизация: локальность map)

- По возможности следует запускать `map` на узлах, где непосредственно расположен документ для сокращения нагрузки на сеть. Так как файлы реплицируются, для каждого документа будет несколько кандидатов.
- Если все кандидаты заняты, следует отдать предпочтение узлам, расположенным как можно ближе к тем, на которых расположены реплики документов. Это требуется для максимального ускорения передачи файлов.

Алгоритм. (Оптимизация: избыточность)

- Любую подзадачу можно запускать как гонку на нескольких узлах одновременно, т.е. взять результат из узла, который завершит выполнение раньше. Но запускать все задачи даже в двух копиях дорого.
- Можно запустить копию подзадачи, которая попала на медленный узел, таким образом ускорить общее время исполнения, так как медленные задачи не будут тормозить весь процесс.
- При завершении $\approx 95\%$ подзадач, можно запустить гонку по оставшимся, поскольку это будут самые медленные подзадачи, задерживающие выполнение задачи в целом. Конкретный процент задач можно настраивать.

49 MapReduce. Каскады MapReduce задач, Combiner-оптимизация, Map-only задачи.

Некоторые задачи нельзя решить за одно исполнение MapReduce.

Пример. (Для каждого интервала $[1000k, 1000(k+1))$ найти число слов с числом вхождений, лежащим в данном интервале.)

- С помощью MapReduce посчитать количество вхождений для каждого слова.
- С помощью MapReduce по результату предыдущего решить поставленную задачу:

```
fun mapper(doc: [(String, Int)] -> [(Int, String)]):
    for word, count in doc:
        yield count // 1000, word

fun reducer(words: [String]): Int:
    return set(words).size()
```

Определение. *Каскад* — ориентированный ациклический граф MapReduce задач, принимающих на вход результат выполнения предыдущих задач. Порядок вызовов определяется топологической сортировкой. Независимые друг от друга задачи можно решать параллельно.

Алгоритм. (Оптимизация: Combiner)

- Результатом `map` может быть большое число пар, что может быть избыточным.
- Перед передачей корзин редьюсеру, маппер может выполнить свертку по данным своего узла локально, чтобы сократить объем передаваемой информации.

Замечание. *Combiner* может не совпадать с *Reducer*.

Пример. (Подсчет среднего значения по каждому ключу)

```
fun mapper(doc: [(String, Float)]) -> [(String, Float)]:
    for group, value in doc:
        yield group, value

fun combiner(values: [Float]) -> (Float, Int):
    return (sum(values), len(values))

fun reducer(values: [(Float, Int)]) -> Float:
    return sum(values[0]) / sum(values[1])
```

Определение. *Map-only задачи* — задачи MapReduce, в которых не требуется свертка.

Пример. (Поиск по большой коллекции документов)

- Выполним поиск по каждому документу независимо, весь результат сопоставим одному фиктивному ключу.
- Объединим результаты с каждого маппера, минуя обработку редьюсерами.

50 Распределённое объединение. Использование границ и слияние отсортированных последовательностей.

Для сортировки методом слияния достаточно реализовать процедуру слияния — объединения отсортированных последовательностей.

Алгоритм. (Распределенное объединение)

- На каждом из R узлов лежит по одному файлу, каждый из которых содержит данные, отсортированные по ключу. Задача заключается в их слиянии.
- Узел, производящий слияние, делает это по стандартному алгоритму с использованием кучи.
- Так как результат может заполнить всю память на объединяющем узле, при переполнении задача продолжается на другом узле.
- Узлы, хранящие сливаемые последовательности, должны уметь отдавать очередной элемент по запросу, а также создавать контрольные точки и откатываться к ним. Это требуется для корректного возобновления слияния после падения узла слияния. Контрольные точки должны создаваться при переполнении очередного узла слияния.

Оптимизация: блочное чтение.

- Чтение по одному ключу с узлов неэффективно из-за накладных расходов на передачу данных по сети.
- Узел, производящий слияние, может получать блок ключей и кешировать его. Если блок от какого-либо узла исчерпывается, запрашивается новый, а также создается новая контрольная точка.

Алгоритм. (Распределенная сортировка)
Рассмотрим другой подход к распределенной сортировке: распределим ключи равномерно по нескольким узлам так, чтобы на каждом лежали только те, которые попадают в некоторый интервал. Затем будет достаточно отсортировать ключи на каждом узле локально и выдать в порядке интервалов.

Пусть m — общее число ключей, n — число узлов, на которых будет производиться локальная сортировка. Найдем такие границы интервалов:

$$a_1 < a_2 < \ldots < a_{n-1}$$

Что распределение ключей по этим границам:

$$r(x) = \begin{cases} 1, & x < a_1 \\ 2, & a_1 \le x < a_2 \\ \ldots & \\ n, & x \ge a_{n-1} \end{cases}$$

Будет равномерным:

$$P(r(x) = 1) \approx P(r(x) = 2) \approx \ldots \approx P(r(x) = n)$$

Для этого узнаем распределение ключей, взяв выборку из k ключей:

```
fun mapper(doc: [Key]) -> [(String, Key)]:
    for key in doc:
        if random.uniform(0, 1) < k / m:
            yield "sample", key
```

51 Resilient Distributed Datasets. Мотивация, реализация, секционирование датасетов, материализация датасетов.

Мотивация.

- В итеративных алгоритмах удобно производить вычисления, последовательно выполняя `map`, `filter` и `flatMap`. Однако, каждая операция требует работу с диском для сохранения промежуточных данных, которые не нужны в конечном результате.

Пример. (Ленивые вычисления)
Хотим посчитать:

```
result = datasource.fetch()
                   .map(...)
                   .filter(...)
                   .flatMap(...)
                   .collect(...)
```

Обозначим `f` — `map`, `p` — `filter`, `g` — `flatMap`.
Плохая реализация:

```
data = datasource.fetch()

mapped = []
for elem in data:
    mapped.add(f(elem))

filtered = []
for elem in mapped:
    filtered.add(elem) if p(elem)

result = []
for elem in filtered:
    result.addAll(g(elem))
```

Если считать результат на каждом шаге, то на диске аллоцируется много ненужных коллекций (`mapped, filtered`).
Оптимальная реализация:

```
result = []

for elem in datasources.fetch():
    mapped = f(elem)
    if p(mapped):
        result.addAll(g(mapped))
```

Аллоцировали на диске только одну коллекцию — `result`.

Алгоритм. (Последовательная реализация)

- Результат каждой операции — ленивый контейнер.

- Каждый контейнер хранит, с помощью какой операции и из каких контейнеров он был получен.

Алгоритм. (Распределенная реализация: Resilient Distributed Datasets)

- Датасет шардируется по строкам на несколько секций.
- Для каждой секции известно, с помощью какой операции и из каких секций родительского датасета она была посчитана.
- Если шардирование выполнено не по ключу, то для завершения операции может потребоваться пересылать данные по сети (например, для операции `groupByKey`).
- Если шардирование выполнено по ключу, то пересылать данные по сети не требуется.
- В случае сбоя, утраченную секцию можно пересчитать, поскольку известна операция и секции родителя, к которым ее необходимо применить. По возможности, это можно сделать параллельно.
- В отдельных случаях датасет нужно в явном виде посчитать и сохранить, то есть *материализовать*:
  - если требуется совершить несколько запросов к датасету,
  - если из датасета будет построено более одного производных,
  - если требуется создать контрольную точку для ускорения восстановления после сбоя,
  - если операция этого требует (например, сортировка).

  Материализацию можно производить как на диск, так и в ОЗУ, в зависимости от целей пользователя. Также можно материализовать отдельные секции.

52 Распределённое машинное обучение. Разделение градиента, алгоритм с обменом градиентами, проблемы при масштабировании.

Алгоритм. (Разбиение градиента)

- Датасет разбивается на части.

  $$D_i \subset D$$

  Определим функцию потерь на части датасета.

  $$\mathbb{L}_{D_i} = \sum_{x,y \in D_i} L(x, y, \hat{y})$$

  Тогда значение функции потерь на всем датасете есть сумма значений функции потерь на частях датасета.

  $$\mathbb{L} = \sum_{x,y \in D} L(x, y, \hat{y}) = \sum_i \sum_{x,y \in D_i} L(x, y, \hat{y}) = \sum_i \mathbb{L}_{D_i}$$

  Получаем, что значение частных производных по параметру есть сумма частных производных по параметру на каждой части датасета.

  $$\frac{\partial \mathbb{L}}{\partial W} = \frac{\partial \left(\sum_i \mathbb{L}_{D_i} \right)}{\partial W} = \sum_i \frac{\partial \mathbb{L}_{D_i}}{\partial W}$$

- Каждый узел считает градиент на своей части датасета, после чего рассылает его остальным узлам.
- На каждом узле полученные градиенты суммируются с собственными.
- Итоговый градиент используется в оптимизаторе.

Замечание. Данный алгоритм работает в синхронной сети, без сбоев узлов.

Масштабирование.

- Для рассылки градиента каждый узел посылает $N(N-1)|W|$ байт.
- Оценка сложности вычисления градиента и обновления весов: $\mathcal{O}\left(|W|\frac{|D|}{N}\right)$.
- Оценка сложности пересылки градиента: $\mathcal{O}(|W|N)$.

Замечание. При использовании данного подхода увеличение числа узлов приводит к увеличению доли времени исполнения, затрачиваемого на пересылку градиента. Способы борьбы с этим рассматриваются в следующих билетах.

53 Распределённое машинное обучение. Quantization, Sparsification, Error Correction.

Алгоритм. (1-Bit Quantization)

- Посчитаем средние значения среди положительных и отрицательных компонент градиента, обозначим за $\mathbb{E}_+$ и $\mathbb{E}_-$ соответственно.
- Для каждой компоненты градиента будем посылать один бит, отвечающий за знак исходного значения.
- Также будем посылать два числа: $\mathbb{E}_+$ и $\mathbb{E}_-$.
- На принимающей стороне заменим все положительные компоненты градиента на $\mathbb{E}_+$, а отрицательные на $\mathbb{E}_-$.
- Размер такого пересылаемого градиента будет $2 \cdot 32 + |W|$ бит вместо $32 \cdot |W|$ бит.
- После таких преобразований потеряли в точности (слабо) и в скорости сходимости (сильно).

Алгоритм. (Stochastic Quantization)

- Нормируем все компоненты градиента на отрезок $[-1, 1]$.

$$\hat{W}_i = \frac{W_i}{\sqrt{\sum_{j=1}^{|W|} W_j^2}} \in [-1, 1]$$

- k — число бит, которыми будет кодироваться одна компонента. Разобьем отрезок $[-1, 1]$ на $2^k - 1$ равных подотрезков, сопоставив их границам числа от 0 до $2^k - 1$.
- Для кодирования очередной компоненты градиента будем делать следующее:
  - Выбираем подотрезок, на который попадает отнормированное значение;
  - Пусть значение равно w, и оно попадает на подотрезок $[x, y]$. Подбросим нечестную монетку f и выберем одну из границ. Вероятность выпадения каждой границы линейно зависит от расстояния w до противоположной.

    $$\mathbb{P}(f(w) = \mathcal{E}) = \begin{cases} \frac{y-w}{y-x}, & \mathcal{E} = x \\ \frac{w-x}{y-x}, & \mathcal{E} = y \end{cases}$$

    В зависимости от выбранной границы, кодируем компоненту числом от 0 до $2^k - 1$, соответствующим границе;

– Преобразование получилось несмещенным, повышает точность и скорость сходимости.

$$\mathbb{E}(f(w)) = x \cdot \frac{y-w}{y-x} + y \cdot \frac{w-x}{y-x} = w$$

.

- Посылаем другим узлам k чисел — закодированный градиент, а так же нормировочную константу.
- Размер такого градиента будет $32 + k \cdot |W|$ бит вместо $32 \cdot |W|$ бит.

Алгоритм. (Sparsification)

- В градиенте остаются только k наибольших по модулю компоненты, остальные заменяются на нули.
- При пересылке отправляются только значения ненулевых компонент с их номерами.
- Размер такого градиента будет $32 \cdot k + k \cdot \log|W|$ бит вместо $32 \cdot |W|$ бит.
- Данный подход совместим с предыдущими алгоритмами.

Алгоритм. (Error correction)

- Пусть на шаге i был градиент ∇_i.
- После преобразований вышеописанных алгоритмов, получим преобразованный градиент $\tilde{\nabla}_i$.
- Сохраним локально ошибку $err_i = \nabla_i - \tilde{\nabla}_i$.
- На следующем шаге, скорректируем градиент с учетом этой ошибки: $\nabla_{i+1} = \nabla_{i+1} + err_i$.

54 Распределённое машинное обучение. Схемы пересылки сообщений, обмен весами, послойное обучение, SwarmSGD.

Схема пересылки сообщений: через мастер-узел.

- Каждый узел посылает свой градиент мастер-узлу, который просуммирует их локально.
- Затем каждый узел получает от мастер-узла сумму.
- Таким образом, суммирование произошло один раз. При этом число пересланных каждым узлом (кроме мастер-узла) бит равно $\mathcal{O}(|W|)$ вместо $\mathcal{O}(|W|N)$. В системе будет послано $2(N-1)$ сообщений вместо $N(N-1)$.

Схема пересылки сообщений: по кругу.

- Сначала градиент пересылается от 1 узла ко 2 узлу, от 2 узла к 3, и так далее. При прохождении через каждый узел к пересылаемому градиенту прибавляется локальный градиент узла.
- Когда градиент обрабатывает узел N, в нем просуммированы все локальные градиенты каждого узла. Он рассылается всем узлам обратно (напр. от N к $N-1$, от $N-1$ к $N-2$ и так далее).
- В системе будет послано $2(N-1)$ сообщений вместо $N(N-1)$.
- Алгоритм полностью децентрализован.

Схема пересылки сообщений: Gossip.

- Каждый узел на каждом шаге посылает свой градиент случайному подмножеству других узлов.
- При получении информации от соседних узлов — пересылает ее.
- Узел ждет, пока не получит градиент от всех остальных.

Алгоритм. (Обмен весами)

- Каждый узел независимо учит модель K ходов, причем только на своих локальных данных.
- Затем все узлы обмениваются **весами** между собой и усредняют.
- При таком подходе посылается в K раз меньше данных. Но при слишком большом значении параметра скорость схождения падает.

Алгоритм. (SwarmSGD)

- Случайно выбираются два узла, которые независимо параллельно учат модель на своих локальных данных K ходов.

- Затем узлы обмениваются весами между собой и усредняют.
- При таком подходе нет глобального усреднения весов, но алгоритм сходится.

Алгоритм. (Послойное обучение)

- Слои нейронной сети разбиваются на N групп, за каждую группу отвечает один узел.
- В течение K шагов каждый узел учит исключительно свою группу слоев. При этом градиенты по более глубоким слоям также считаются, но не записываются, а в градиенте обнуляются.
- Затем узлы пересылают друг другу обновленные веса нейронов своих групп.

55 Самостабилизация: взаимное исключение

Определение. *Самостабилизация:*

- Легальное состояние остаётся легальным.
- Из любого состояния попадём в легальное через конечное число шагов.

Пример. Существует алгоритм самостабилизации для *взаимного исключения*, где ровно один процесс имеет привилегию.

Определение. *Состояние системы:*

- N машин расположены в кольце. Каждая имеет K состояний ($K \geqslant N$). Если состояние меняется, информация отправляется по часовой стрелке дальше.
- Машина имеет привилегию, если:
  - Для первой: $S = L$. Состояние первой машины равно состонию машины слева (влево это против часовой стрелки).
  - Для остальных: $S \neq L$.

Определение. *Правила перехода между состояниями:*

- Для первой: $S \neq L \Longrightarrow S := (S + 1) \mod K$.
- Для остальных: $S \neq L \Longrightarrow S := L$.

Теорема 55.1. *Данный алгоритм самостабилизируется.*

Доказательство.

- Очевидно, что легальное состояние остаётся легальным.
- Легальное состояние достигается через конечное число ходов из любого состояния:
  - Как минимум одна машина имеет привилегию и может ходить. Действительно, если все состояния одинаковые, то первая машина может ходить. Иначе есть две пары процессов-соседей с разными состояниями, и в хотя бы одной из этих пар найдется процесс с привилегией.
  - Первая машина сдвинется через через $\mathcal{O}(N^2)$. В худшем случае привилегия будет передаваться с последних двух процессов в кольце, постепенно добавляя первые процессы.
  - Рано или поздно первая машина будет иметь уникальный S (т.к. $K \geqslant N$ и только первая машина имеет право на изменение состояния).
  - После этого через $\mathcal{O}(N^2)$ система стабилизируется.

■

56 Самостабилизация: поиск остовного дерева

Про самостабилизацию в предыдущем билете.

Определение. *Состояние узла в системе:*

- `dist` – расстояние до корня.
- `parent` – предок в дереве.

Алгоритм.

- Для корня фиксированно: $\texttt{dist} := 0$, $\texttt{parent} := -1$.
- Остальные процессы периодически совершают следующее:
  - Найти соседа с минимальным $\texttt{dist}_j$.
  - $\texttt{dist} := \texttt{dist}_j + 1$.
  - $\texttt{parent} := j$.

Упражнение. Доказательство корректности.