Распределенные системы

Содержание

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 на $\mathbb{E} \times \mathbb{E}$ такой, что

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

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

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

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:

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

Свойства векторного времени:

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

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

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)

- Весь процесс 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 узел шлёт запрос координатору, дожидается разрешения, затем входит в критическую секцию. При завершении работы узел 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 ко всем другим узлам
`ok`	Подтверждение получения (не даёт права входа в CS)
`release`	Освобождение узлом критической секции (всем узлам)

Таблица 4: Виды запросов алгоритма Лампорта

Алгоритм. (Алгоритм Лампорта)

- Координатор отсутствует, все узлы равны.
- Сообщения request и release рассылаются всем другим узлам, всего $3n - 3$ сообщения на CS.
- Используются логические часы лампорта. Для установления порядка "кто раньше". Обязательно требуется порядок FIFO на сообщениях.
- Все узлы 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 номеру узла, который посылается вместе с часами).
  - Получен ok от всех других узлов, т.е. они знают о вашем запросе.
- Если узел хочет войти в CS, то он посылает всем другим узлам request со своими часами и id. Ждёт от всех ok. Если других запросов не поступало, либо время нашего запроса меньше времени других запросов, то входим в критическую секцию. Иначе ждем release от всех узлов, которые раньше нас в очереди.

6 Взаимное исключение в распределённой системе. Алгоритм Рикарда и Агравалы

Вид запроса	Действие
`request`	От запрашивающего ко всем другим узлам
`ok`	После выхода из критической секции

Таблица 5: Виды запросов алгоритма Рикарда и Агравалы

Алгоритм. (Алгоритм Рикарда и Агравалы)

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

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.

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. Виды 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 любым двух таких множеств непусто, что удовлетворяет определению кворума.

Замечание. Не все кворумы тривиальны и плохо мастурбируются. Например, “рушащиеся стены” имею 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)

- Снача 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)
Сообщения делятся на 4 вида:

- 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 токен-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 он не может участвовать в wr-сообщениях.

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 маркер 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 тогда и только тогда, когда он верен на хотя бы 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 VC.
- При наступлении истинности локального предиката, отправляем сообщение координатору C (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 то 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 всгеда 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 $[F, G]$. Запомним состояние системы на срезе F (например, это может сделать координатор, если он используется). После этого каждый процесс будет помнить, менялось ли у него состояние (относительно блокировки). 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. Порядок называется *FIFO (First In First Out)*, если

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

То есть не бывает пар сообщений, упорядоченных неправильно.

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

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 получателем, 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.

Определение. Порядок называется *причинно-согласованным*, если

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

То есть не бывает пар сообщений, упорядоченных неправильно.

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

Алгоритм. (Централизованный алгоритм для причинно-согласованного порядка) Выберем координатора, который будет осуществлять передачу. Для корректности алгоритма достаточно, чтобы каналы до координатора имели FIFO порядок.

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.
- Сообщение от P_i к P_j с матрицей W обрабатывается, если выполнены все условия:
  - Соблюдается FIFO порядок: $W_{ji} = M_{ji} + 1$. Сообщение имеет ожидаемый номер.
  - Соблюдается причинная согласованность: $\forall k \neq j\colon\ M_{ki} \geqslant W_{ji}$. То есть посылающий процесс не знает о событиях, о которых не знает принимающий.
  - После обработки обновляем матрицу: $M = \max(M, W)$.
- При невыполнении хотя бы одного из условий сообщение кладется в очередь.

Замечание. Причинно-согласованный порядок сильнее, чем FIFO порядок.

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

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

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

Замечание. Синхронный порядок является самым сильным требованием о порядке.

Алгоритм. (Централизованный алгоритм для синхронного порядка)
Выберем координатора, который будет осуществлять передачу. Отличие от централизованного алгоритма для причинно-согласованного порядка заключается в том, что координатор дожидается подтвержедния, что сообщение доставлено и только после этого посылает новое. Для корректности алгоритма также достаточно, чтобы каналы до координатора имели FIFO порядок.

Алгоритм. (Распределенный алгоритм для синхронного порядка)
Алгоритм основан на иерархии процессов. Бывают “большие“ и “маленькие“ процессы.

- Большой процесс шлет маленькому сообщение с подтвержением получения. Пока подтвержение не получено, процесс пассивен, не обрабатывает приходящие сообщения и не посылает новые.
- Маленький процесс шлет большому сообщение только после получения разрешения, то есть подтвержения возможности посылки. Большой процесс пассивен между отправкой подтверждения и получением сообщения.
- Пассивный процесс не участвует в пересылке сообщений, поэтому всегда можно выбрать момент передачи сообщения $T(m)$ в его промежутке пассивности.
- Не может произойти взаимная блокировка, потому что становится пассивным всегда более большой процесс.

Замечание. Независимые пары процессов могут общаться независимо.

19 Общий порядок (total order). Алгоритм Лампорта.

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 свойство всегда выполняется.

Алгоритм. (Централизованный алгоритм обеспечения общего порядка)
Пусть в системе соблюдается FIFO порядок сообщений. Тогда если процесс P 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 взаимной блокировки. Пусть в системе соблюдается FIFO порядок сообщений. Все multicast-сообщения придется заменить на broadcast. Процесс, собирающийся послать сообщения, берет “билет”, соответствующий его логическому времени, и посылает `request` запрос всем другим процессам. Те, в свою очередь, отвечают ему `ok`. После того, как был получен `ok` от всех процессов, отправитель начинает рассылку. Как и в алгоритме Лампорта для взаимного исключения, порядок обработки сообщений определяется парой из билета и номера процесса.

20 Общий порядок (total order). Алгоритм Скина.

Первую часть вопроса см. в предыдущем билете.

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

- Пусть процесс P хочет сделать рассылку. В таком случае, он шлет всем тем, 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` (аналогично алгоритму Лампорта).
- Как только отправитель 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 вершины 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

- 1) отказ узла (самый простой)
- 2) отказ канала (равносильно отказу всех узлов)
- 3) ненадежная доставка (некоторые сообщения недоходят)
- 4) Византийская ошибка (враги захватили узел и пытаются тебя поиметь)

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

- 1) Синхронные системы
- 1.1) время передачи сообщения ограничено сверху
- 1.2) можно разбить выполнение алгоритма на фазы
- 2) Асинхронные системы
- 2.1) время не ограничено
- 2.2) время передачи конечно, если нет отказов

Определение. Рассмотрим свойства консенсуса в распределнной системе:

- 1) Согласие — Все процессы должны завершиться с одним и тем же решением.
- 2) Нетривиальность — Должны быть варианты исполнения, приводящие к разным решениям.
- 3) Обоснованность — Решение должно быть предложением одного из процессов.
- 4) Завершение — Протокол должен завершиться за конечное время.

Замечание. Достичь этих свойств без отказа легко:

- 1) Каждый процесс шлет свое предложение вем остальных.
- 2) Дожидается предложение от других процессов.

- 3) Теперь из данных предложений, используя детерминированную функцию (max, min, etc), выбирает решение.

Этот алгоритм работает и в асинхронной системе.

Теорема 21.1. *КЛЮЧЕВАЯ ТЕОРЕМА КУРСА (Фишера-Линча-Патерсона)*
Не существует такого детерминированного алгоритма, который при любом исполнении за конечное время придет к консенсусу в асинхронной системе.

Доказательство. (от противного)
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 функции:

- 1) Указать ожидать получения сообщения (нет возможности указать время ожидания)
- 2) Отправить сообщение
- 3) Принять решение (можно только 1 раз это сделать, но при этом сообщать свое решение другим алгоритмам разрешено)

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

Определение. Шагом в такой конфигурации называется

- 1) обработка какого-то сообщения процессом
- 2) внутреннее действие этого процесса и посылка им от нуля до нескольких сообщений до тех пор, пока процесс не перейдет к ожиданию следующего сообщения.

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

- 1) может содержать сколько угодно входных данных
- 2) начальных конфигураций много (на каждый вариант входных данных)
- 3) 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* — процесс, который делает конечное число шагов 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 к 0, и цепочки, приводящие 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. (от противного)

- Если не существует такой конфигурации, то все конфигурации одновалентны (есть конфигурации, которые всегда приводят к 0, есть приводящие к 1)
- Возьмем начальное состояние процессов, приводящие к 0 и к 1, начнем по очереди их заменять, чтобы найти пару конфигураций разной валентности, отличающиеся начальным состоянием только одного процесса.
- Тогда пусть этот процесс откажет сразу же, тогда его начальное состояние ни на что не влияет, то есть можно получить и 0 и 1 противоречие.

■

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

Доказательство. (рассмотрим что у нас есть)

- Дана конфигурация G, e — произвольное событие (процесс p, сообщение m). C — множество конфигураций из G без e. D — множество конфигураций, где e — это последнее событие.
- Допустим в D, нет бивалентных конфигураций.
- Очевидно, в ней есть и 0-валентные и 1-валентные, иначе G не бивалентна.
- Рассмотрим соседние конфигурации Ci и Ci + 1, отличающиеся одним сообщением, но приводящие при получении сообщения e в Di и Di + 1 разные по валентности конфигурации.
- 1 случай) Если процессы коммутируют, то противоречие.

- 2 случай) Иначе, рассмотрим 6 положений. Просто конфигурация C, если сначала произошло событие e, если произошло сначала f, а потом e и 3 соответствующих, если процесс отказал после выполнения операции.
- Тогда с одной стороны, если он умер после получения сообщения e, то остальные процессы придут к решению 0, а в случае f - 1. Но тогда, что будет, если процесс не будет просто долго принимать сообщения. Когда все процессы приняли решение, он вдруг проснулся и обработал сообщение. Противоречие.

■

Значит всегда есть переход в бивалентное состояние, значит можно бесконечно гонять конфигурации, но так и не решить что делать. ■

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. (Попробуем победить отказы в случае синхронной системы.)

- Пусть могут отказать f узлов ($0 \leq f \leq N$). Если отказывают все, то кому работать.
- Делаем $f+1$ фазу базового алгоритма (рассылаем известные множества предложений, где одна фаза — это максимальное время доставки сообщения).
- Первый раз множество состоит только из своего предложения, затем из всех полученных и своего. Затем уже скомбинированная информация (например, первый узел не успел отослать свое предложение и умер, а второй доотправил его третьему и третий комбинирует информацию о предложениях) отправляется еще раз и так $f+1$ раз.
- Доказательство корректности при отказе узла (по Дирихле) следует из определения алгоритма.
- За $f+1$ фазу в одной фазе нет отказов, и значит все живые процессы корректно передадут свои и соседские предложения. Происходит синхронизация, то есть множества предположений совпадут, а значит и дальше они не поменяются.

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 4х 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 (смотрите картинку в презентации).
- 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.

Алгоритм. (Консенсус не возможен при 1 предателе на 3 процесса. Доказательство от противного)

- Запутим алгоритм в 4-х копиях. Двум подадим на вход 0, двум даем 1 и соединим их в прямоугольник.
- Тогда верхние процессы считают что нижние предатели и консенсус на 1, а нижние наоборот, что верхние предатели и консенсус в 1.
- Противоречие, так как 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*.
- 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.
    * Участвует в выборе другого кандидата.
    * 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 получение новой записи, информация о предыдущей не совпадает, то *follower* должен выкинуть некорректную запись, уменьшить индекс и повторить операцию заново, пока не останется корректный префикс. Затем получить хвост. (Можно оптимизировать, но смысла нет из-за редкости ошибок).

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 index.

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

Утверждение 34.1. *Raft* гарантирует *safety*.

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.

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

Gossip и базовые определения читайте в предыдущем билете.

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. *Множества:*

- *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:

  – Недетерменированные 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 рассылкает ее на все реплики.
- Реплики применяют операцию локально и отвечают лидеру.
- Лидер дожидается подтвержения от всех реплик, что они применили операцию.
- После этого лидер подтвержает 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 же 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 накаждой 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 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.
- `group :: [[(Key, Value)]] -> [(Key, Value)]`, результаты `map` объединяются по ключу.
- `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. (Распределенный подсчет встречаемости слов)

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

Пример. (Распределенный подсчет обратного индекса слов по документам)

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

39 MapReduce. Распределенная реализация. Маппе-ры 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 (напр. `hash(key) % R`).
- Один узел является *мастером* и координирует работу всех узлов.

Определение. *Узел-маппер в распределенной реализации*.

- Каждый узел хранит в памяти *R* корзин пар ключ-значение, каждая из которых впоследствии попадает на свой определенный редьюсер.
- При переполнении памяти корзины сбрасываются с оперативной памяти на диск, в отдельный файл.
- *Мастер* с определенной периодичностью уведомляет мастер о статусе выполнения задачи.

Определение. *Узел-редьюсер в распределенной реализации*.

- Узлы читают предназначенные им корзины с маппров до тех пор, пока не будут собраны все значения.
- Сортирует и группирует собранные данные по ключу.
- Вызывает `reduce` на собранных значениях и записывает результат в итоговый файл.

Обраотка 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, задача 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.

Обраотка сбоев редьюсеров.

- Сбой редьюсера определяется пингом.
- При сбое редьюсера до завершения свертки и репликации результирующего файла, задача полностью перезапускается на другом узле. При этом потребуется заново собрать данные с соответствующих корзин на каждом маппере.
- В противном случае, перезапуск задачи не требуется

Обраотка сбоев мастера.

- 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 перезапускать задачу, не повторяя уже выполненных вычислений.

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.

Оптимизация: избыточность.

- Любую подзадачу можно запускать как гонку на нескольких узлах одновременно, т.е. взять результат из узла, который завершит выполнение раньше.
- Гонку следует запускать на медленных узлах, так 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.

40 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()
```

Утверждение 40.1. В общем случае, задачи распределенного вычисления можно решать, составив ацикличный граф подзадач MapReduce. Порядок вызовов определеяется топологической сортировкой. Независимые друг от друга задачи можно решать параллельно.

Оптимизация: Combiner.

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

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.

41 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 должна 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 распределенной сортировки: 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
```

k должно быть достаточно малым, чтобы данные не переполнили единственный узел, но достаточно большим, чтобы выборка была репрезентативной. Тогда в качестве a_i достаточно взять $\frac{ik}{n}$ порядковую статистику.

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

Мотивация.

- Удобно производить вычисления, последовательно выполняя `map`, `filter` и `flatMap`. Однако, каждая операция требует работу с дисокм для сохранения промежуточных данных, которые не нужны в конечном результате.
- В итеративных алгоритмах (напр. локальные оптимизаторы, графовые алгоритмы, алгоритмы связанные с Марковскими цепями) на каждом шаге данные преобразуются и сохраняются, что понижает производительность из-за активной работы с диском.

Пример. (Ленивые вычисления: 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))
```

Пример. (Ленивые вычисления: оптимальная реализация)

```
result = []

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

Замечание. Данная идея реализована в:

- Java streams,
- Haskell stream fusion,
- Python generators.

Алгоритм. (Последовательная реализация)

- Результат каждой операции — ленивый контейнер.

- Каждый контейнер хранит только операцию, с помощью которой он был получен, и список других контейнеров, из которых он был получен.

Алгоритм. (Распределенная реализация: Resilient Distributed Datasets)

- Датасет шардируется по строкам на несколько секций. В зависимости от выбора, по чему производится шардирование (ключ или значение), некоторые операции смогут производиться локально, без использования сети (напр. `groupByKey` при шардировании по ключу).
- После операции получаем из родительских секций производные секции. В каждой производной секции хранится, с помощью какой операции и из каких родительских датасетов она была посчитана.
- В случае сбоя, утраченную секцию можно пересчитать, поскольку известно, с помощью каких операций и из каких секций она была посчитана. По возможности, это можно сделать параллельно.
- В отдельных случаях промежуточные вычисления выполняются энергично (датасет материализуется):
  - если требуется совершить несколько запросов,
  - если из датасета будет построено более одного производных,
  - если требуется создать контрольную точку для ускорения восстановления после сбоя,
  - если операция этого требует (напр. сортировка).

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

Замечание. Алгоритм работает в доверенных сетях и не приспособлен для работы с «византийскими» процессами.

43 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 с увеличению доли времени исполнения, затрачиваемого на пересылку градиента. Способы борьбы с этим рассматриваются в следующих билетах.

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

Алгоритм. (1-Bit Quantization)

- Посчитаем средние значения среди положительных и отрицательных компонент градиента, $\mathbb{E}_+$ и $\mathbb{E}_-$ соответственно.
- Заменим все положительные компоненты градиента на $\mathbb{E}_+$, а отрицательные на $\mathbb{E}_-$.
- Для каждой компоненты градиента будем посылать один бит, отвечающий за знак исходного значения.
- Размер такого градиента будет $2 \cdot 32 + |W|$ бит вместо $32 \cdot |W|$ бит.

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, спопоставив 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.

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

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

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

    .

– 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 оставшихся 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: $\nabla_{i+1} = \nabla_{i+1} + err_i$.

45 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.

46 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 46.1. *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.

■

47 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.