Распределенные системы

Содержание

1 Формализм. Логические часы Лампорта (свойства и алгоритм) 3

2 Формализм. Векторные часы (свойства и алгоритм) 4

3 Формализм. Часы с прямой зависимостью (свойства и алгоритм) 5

4 Взаимное исключение в распределенной системе. Централизованный алгоритм. 6

5 Взаимное исключение в распределённой системе. Алгоритм Лампорта 7

6 Взаимное исключение в распределённой системе. Алгоритм Рикарда и Агравалы 8

7 Взаимное исключение в распределённой системе. Алгоритм обедающих философов. 9

8 Алгоритм на основе токена. 10

9 Взаимное исключение в распределённой системе. Алгоритмы на основе кворума (простое большинство, рушащиеся стены). 11

10 Согласованное глобальное состояние (согласованный срез). Алгоритм Чанди-Лампорта. Запоминание сообщений на стороне отправителя. 12

11 Согласованное глобальное состояние (согласованный срез). Алгоритм Чанди-Лампорта. Запоминание сообщений на стороне получателя. 13

12 Глобальные свойства. Стабильные и нестабильные предикаты. Слабый конъюнктивный предикат. Централизованный алгоритм. 14

13 Слабый конъюнктивный предикат. Распределенный алгоритм. 16

14 Диффундирующие вычисления, пример. Останов. Алгоритм Дейксты и Шолтена. 17

15 Локально-стабильные предикаты: согласованные интервалы, барьерная синхронизация (3 алгоритма). Применение для определения взаимной блокировки (deadlock). 18

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

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

18 Недетерминированные алгоритмы консенсуса. Алгоритм Бен-Ора. 21

19 Шардирование. Общий принцип. Статическое отображение, остаток от деления, расширяемое побитовое хеширование, битовый бор, постоянное число секций. 23

20 Шардирование. Общий принцип, хеширование рандеву, консистентное хеширование, Multi-Probe Consistent Hashing. 25

21 Шардирование. Общий принцип, JumpHash. 27

22 Транзакции в распределенных системах. ACID. 2 Phase Locking. 30

23 Транзакции в распределённых системах. ACID. 2 Phase Commit. 32

24 Raft. Алгоритм, его свойства. 33

25 Самостабилизация: взаимное исключение 36

26 Самостабилизация: поиск остовного дерева 37

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 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.

17 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.

18 Недетерминированные алгоритмы консенсуса. Алгоритм Бен-Ора.

Замечание. Невозможность построения алгоритма консенсуса при наличии ошибок доказывается только в случае выполнения следующих свойств:

- Система асинхронная.
- Алгоритм детерминированный.

Избавимся от второго требования.

Замечание. К недетерминированным алгоритмам консенсуса предъявим требования:

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

Алгоритм. (Бен-Ор)
Пусть в системе N процессов, отказать могут только f.

- 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 18.1. В одном раунде процессы не могут ратифицировать разные значения.

Лемма 18.2. 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.
- Время работы алгоритма объясняется примерно так: на каждом раунде все процессы начнут с одинаковым предпочтением с вероятностью, не меньшей 2^{-N}.

19 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. (Наивный подход)
Построим такое отображение ключей в номера узлов:

$$\texttt{nodeid} \leftarrow \texttt{hash}(k) \mod N,$$

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.

Алгоритм. (Постоянное число секций)

- Заводим S секции, не меняем их число.
- Выбираем способ отображения влюча 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)

- Секий 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.

20 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:

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

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

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

- 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).

21 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 21.1. (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 21.2. $P(ch(k,n) = ch(k,m)) = \frac{m}{n}$, 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 $r\ U(0,1)$. 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)
    }
}
```

Лемма 21.3. (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.

22 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.

23 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.

24 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 24.1. *Raft* гарантирует *safety*.

25 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 25.1. *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.

■

26 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.